Efficient Implementation of Sequential Loops in Dataflow Computation

Boon Seong Ang

NE43-205, Laboratory for Computer Science,
Massachusetts Institute of Technology, Cambridge, MA 02139.
hahaha@abp.lcs.mit.edu

Abstract

The implementation of sequential loops in dataflow computation had traditionally not received very much attention as it was assumed that most loops would be executed in parallel. This assumption was valid for earlier dataflow machines such as the MIT Tagged Token Dataflow Architecture (TTDA)[2], Sigma-1[9] but not for the newest generation of dataflow machines including Monsoon[6], EM-4[11] and Epsilon-2[7]. On the latter machines, sequential loops use less memory, and can execute in fewer instructions, albeit with lower parallelism then the parallel versions. This characterisation of sequential and parallel loops suggests that programs should have parallel outer loops and sequential inner loops. The run time of sequential loops therefore become significant in the overall run time. We also found that previous implementations of sequential loops can incur fairly high overheads. In this paper, we present two new ways of implementing sequential loops that have lower overhead then previous methods. We studied this problem in the context of compiling Id[14, 15] for Monsoon.

1 Introduction

Loops executing on dataflow machines can be classified into three categories according to the amount of inter-iteration parallelism allowed. At one extreme is an *unbounded* loop, where there is no artificial bound on the number of concurrent iterations. Such a loop is compiled into a recursive procedure which together with non-strict procedure call convention lead to uncontrolled unraveling of the loop. Next, we have what is often referred to as *k-bounded* loops[5], where the number of iterations executing concurrently is bounded to a value k which can be computed at run time. Finally, a *sequential* loop is one where one iteration of the loop has to complete before the next one executes.

The implementation of sequential loops in dataflow computation had traditionally not received much attention as it was assumed that most loops would execute in parallel, predominantly as k-bounded loops. This assumption was valid for earlier dataflow machines such as the MIT Tagged Token Dataflow Architecture (TTDA)[2] and Sigma-1[9] where the per-iteration cycle cost of executing a loop is the same whether the loop is k-bounded or sequential. The consumption of resources, in the form of token "tags" is also the same for both cases. For these machines, it is natural to execute loops in parallel to reap full benefit of parallel processing.

With the newest generation of dataflow machines (e.g. Monsoon[17, 6], EM-4[11] and the Epsilon-2[7]) the situation is different. These machines use *frame memory* on each processor to implement token-matching(see Section 2.1). This has two consequences for loops: (i) every *concurrent* iteration needs its own frame; (ii) sending data between two iterations that do not share (in a time multiplexed way) the same frame requires communication through the interconnection network. The former means that memory usage is lower for sequential loops while the latter suggests that they can execute in fewer cycles. Additional saving comes from fewer frame allocations and deallocations. The benefits come, however, at the expense of reduced parallelism. Thus there is a place for both k-bounded and sequential loops. Outer loops should be k-bounded for parallelism, while inner loops should be made sequential for efficiency reasons. Such a strategy places great important on the efficiency of sequential loops as the run time of inner loops often dominates the overall run time of programs.

Much as sequential loops can potentially be cheap, past implementations[8, 5] have not achieved the goal of keeping the costs down. In this work, we present two new ways of implementing sequential loops that incur less overhead then previous methods. We studied this problem in the context of compiling Id[14, 15], a non-strict, inherently parallel language, for Monsoon[6, 17]. The two implementations use slightly different execution models which are discussed in the paper.

We start by providing some background in Section 2 and proceed to describe some of the past approaches to implementing sequential loops in Section 3. In Section 4, we present a new implementation, the *self-gating* sequential loop schema, which implements a sequential loop in the pure dataflow model. We improve upon this schema by adopting a mixed dataflow-von Neumann execution model to produce the *frame-based-variable* sequential loop schema. This is presented in Section 5. Run time statistics of Id code compiled using the different implementation strategies are presented in Section 6. Conclusion follows in Section 7.

2 Background

This section provides a brief introduction to dataflow computation with emphasis on explaining loops. Dataflow computation orders instruction execution according to the availability of data. A dataflow computer continuously processes *tokens* that deliver data to the instructions. On Monsoon, each token contains: (1) a *value*; (2) an *ip*, pointer to the destination instruction; (3) a *port*, specifying left or right input; and (4) an *fp*, the frame pointer. An instruction is executed during the processing of a token if that token delivers the last piece of input data needed by the instruction. When an instruction executes, zero or more tokens are produced. Program execution starts with the injection of tokens carrying the arguments and ends when there is no more token to be processed.

2.1 Matching and Frames

Central to this execution model is keeping track of which input tokens have arrived at each instruction, and the values carried on these tokens. The process of grouping together all the input operands of a particular instruction, which then enables the instruction for execution is called *matching*. We will further assume that there are at most two inputs to each instruction[1].

A matching mechanism needs to have the following two features: (1) a way of finding out whether the other operand has arrived and if so, fetch that operand; (2) a place for storing the first operand that arrives. In order to support re-entrant code, the mechanism must distinguish between tokens belonging to different invocations of the same piece of code.

On Monsoon, matching is achieved by using *frame* memory which is local to each processing element (PE). Each instruction that requires matching[2] is assigned a unique slot of frame memory which has two fields: (1) a *presence bits* field indicating the presence or absence of each operand; and (2) a *data* field. The former keeps track of whether any one of the operands has arrived, while the latter provides the space for storing the operand that arrives first.

In compiling Id code for Monsoon, we group instructions belonging to the same procedure into a *code block*. The matching slots of instructions belonging to the same code block are assigned contiguous frame memory locations which are collectively called a *frame*. During the processing of a token, the *fp* (base pointer to a frame) on the token and the offset encoded in the destination instruction allows the matching frame slot to be determined. Matching then proceeds according to the state of the presence bits at the frame slot.

Monsoon supports re-entrant procedure invocation by allocating a new frame for each procedure call[3]. By so doing, tokens from different invocations will have different *fp*'s and hence matching under the scheme described above will be able to distinguish them.

2.2 Loops in Id

We start by introducing the syntax with an example that computes $\sum_{i=1}^{n} i$ when called with n. The code is shown in Figure 1. This procedure starts by initializing s to 0 and then iterates through a loop n times. During each iteration, it "updates" s by adding the current value of i to the value of s. Updating a variable presents a problem in Id because

[1] The exception to this is a signal-tree which we will see later.

[2] Not all instructions need matching. Unary operations, for example, have only one input and do not require matching.

```
def sum_1_to n = {s = 0;
                  in
                     {for i <- 1 to n do
                         (next s) = i+s;
                      finally s}};
```

Figure 1: An Id procedure that computes $\sum_{i=1}^{n} i$ when called with n.

Id is basically a functional language. In our example, s is a *nextified variable* of the for loop with an assignment made to (next s) during each iteration of the loop. In the same iteration, s and (next s) are different identifiers. However, the (next s) computed in an iteration automatically becomes the value of s in the next iteration. The scope of a nextified variable extends to the finally part of the loop. Thus the for loop in the example returns the value of (next s) computed during the last iteration of the loop.

3 Past Implementations of Sequential Loops

Implementing sequential loops on Monsoon turns out to be very tricky because by using only one frame to execute the entire loop, the *fp* is the *same* for tokens belonging to different iterations. Loops in general have to be re-entrant but the default way of allowing re-entrant code on Monsoon requires using a new frame for each iteration. With only one frame, unless the compiler compiles in mechanisms to separate tokens belonging to different iterations of the loop, tokens from different iterations destined for the same instruction can erroneously meet under the hardware matching scheme. We call such an event, a *conflict*.

In this section, we describe past attempts at solving this problem. We begin with an "obvious" implementation that uses a barrier at the end of each iteration. This is, unfortunately, incorrect. We next look at a straight forward fix which we call the brute-force 2-barrier approach. Finally, we look at a schema found in Culler's PhD thesis[5] which was the previous best schema.

3.1 An Incorrect, One-barrier Sequential Loop Schema

A first attempt at solving the conflict problem has the compiler insert a *barrier* at the end of each iteration. The barrier separates the current iteration from the next one and ensures that the next iteration does not begin execution until the current one has completed. Such a barrier can be implemented as shown at the bottom of Figure 2.

Figure 2 is the dataflow diagram for the code in Figure 3 which computes the n^{th} Fibonacci number. We can discern four regions in the dataflow diagram. At the top is a shaded oval blob labeled "Loop Predicate", which computes a boolean value indicating whether another iteration should be executed. Below this blob is a row of ovals, each with a "T" and an "F" output. Each oval is a switch that directs the value coming in on top to either the "T" or "F" output depending on the boolean value that comes in by the side. Next comes a big grey region labeled "loop body" where the bulk of the computation of each iteration is carried out. At the bottom is the barrier which consists of two parts: (1) A *signal tree* (the iteration termination signal tree) that collects the next values of the nextified variables.

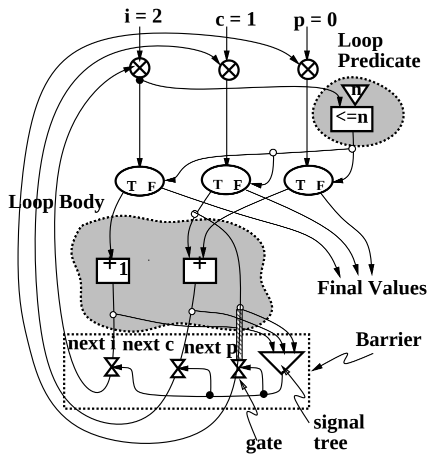

Figure 2: An incorrect implementation of the sequential loop in `fib` using the one-barrier approach. Under certain execution order, two tokens may end up on the value-input arc (shaded) of `(next p)`'s gate at the same time.

```
def fib n = if (n==0) then 0
            else
               {p = 0;  c = 1;
                in
                 {for i <- 2 to n do
                     (next c) = c+p;
                     (next p) = c;
                  finally c}};
```

Figure 3: `fib` computes the n^{th} Fibonacci number.

This produces an output token when all the inputs have arrived. The actual value on the token is unimportant as the token is only used to signal an event. (2) A row of *gate* instructions (represented as bow-ties), one for the `next` value of each nextified variable. Each gate, triggered by the output of the signal tree, is placed just before the `next` value is "circulated back" to the next iteration. A gate instruction takes a value input at the top, and a trigger input on the side. When both input tokens have arrived, a gate produces an output token carrying the same value as that on the value input. Computation that needs the output of a gate instruction is therefore guaranteed not to occur before the trigger is available.

This set of signal tree and gates implements a barrier as no value can cross the array of gates until every value has arrived at the barrier. Because the `next` values of nextified variables are the only input tokens to the next iteration, the next iteration cannot start until the current one finishes. There is one last symbol used in the dataflow diagram, a circle with a cross inside. This is a *merge* which has two inputs and one output. A merge is not a real instruction executed at run time. The simplest way to think about a merge is that the token that emerges from its output can come from either one of its inputs, and no matching of inputs occurs.

While the one-barrier approach may look deceivingly correct, tokens from two iterations *can* conflict for some loops, such as the `fib` loop. On a dataflow graph, conflicts between tokens from different iterations is indicated by either two or more tokens ending up on the same arc at the same time, or two tokens from different iterations matching at an instruction. Having two tokens on the same arc is a problem on Monsoon as that requires storing two values on the same frame slot when we only have space for one. Even if we could store more than one value on a frame slot, we would not know which value to use for matching when a token subsequently arrives at the other input of the same instruction.

In the dataflow graph of Figure 2, two tokens carrying `(next p)` from neighboring iterations can end up on the shaded value-input arc of `(next p)`'s gate at the same time. This happens if at the end of iteration i, after tokens carrying `(next i)`, `(next c)` and `(next p)` have arrived at the barrier, the tokens carrying `(next i)`, `(next c)` *and* values produced by them including those in iteration $(i+1)$ are processed before `(next p)` of iteration i is processed. Processing of the token carrying `(next c)` of iteration i results in several tokens in iteration $(i+1)$ including one which carries `(next p)` of iteration $(i+1)$. This is sent to `(next p)`'s gate while `(next p)` from iteration i is still waiting on the same input arc!

The problem with the naive one-barrier implementation of sequential loop is that the resources that we are reusing include the frame slots used by the array of gates in the barrier. The signal tree of the barrier, on the other hand, does not include completion of usage of these frame slots. Thus the signal from this signal-tree is not a good enough indication that the next iteration can start.

3.2 A Brute-force, Two-barrier Sequential Loop Schema

A brute force fix for the incorrect one-barrier approach uses two barriers as shown in Figure 4. The signal tree of barrier 2 indicates not only that computation has finished within the loop body itself, but that the frame slots belonging to the gates in barrier 1 can be reused. As every output from barrier 2 is recirculated back, the signal tree of barrier 1 cannot signal completion until every output of barrier 2 has left the barrier. This makes it safe to reuse the frame slots used by barrier 2's gates when the signal tree of barrier 1 signals completion.

Although the two-barrier approach is correct, it is very expensive. On Monsoon, each barrier costs approximately $4n$ cycles where n is the number of nextified variables. The total cost of using the two-barrier approach is then $8n$ cycles. This is a huge overhead! Consider the loop of `fib` for example. The overhead will be about 24 cycles when the "useful work" done each iteration is only 3 to 4 cycles. RISC code implementing the same loop will only take 4 cycles each iteration!

3.3 Culler's Sequential Loop Schema

Culler[5] describes another correct implementation of sequential loops which is less costly than the two-barrier approach. The schema is shown in Figure 5. One can still discern parts of two barriers, but the two parts of each barrier, signal tree and array of gates, have been pulled apart. In addition, the schema cleverly avoids one array of gates by instead using the array of switches and gating the predicate

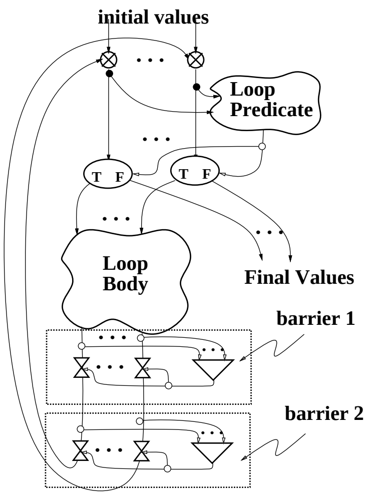

Figure 4: Brute force implementation of sequential loop with two barriers at the end of each iteration. This implementation is correct but very expensive.

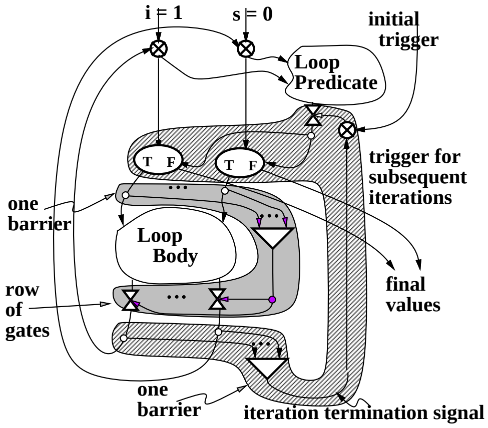

Figure 5: Culler's sequential loop schema.

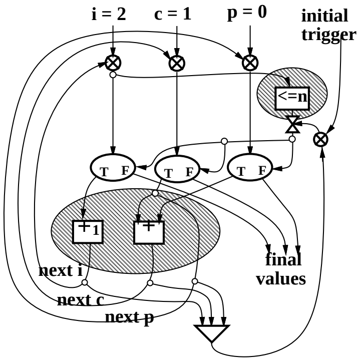

Figure 6: `fib` wired up under the improved the one-barrier sequential loop schema. Note how conflict can still occur at the value-input to p's switch.

input to the switches. The reader is referred to [5] for a correctness proof of this schema.

Culler's schema is less costly then the two-barrier approach. On Monsoon, the overhead for a sequential loop with n nextified variable is approximately $6n$ cycles per iteration, a significant saving compared to $8n$ cycles for the two-barrier approach. This was the best implementation we had before the current work, and was used in the Id compiler. However, $6n$ cycles per iteration is still a hefty overhead. For `fib`, this still means 18 cycles of overhead. We will next present new sequential loop schemata that have much lower overhead costs.

4 Self-gating Sequential Loop Schema

The self-gating sequential loop schema is motivated by an optimized version of the one-barrier approach of Section 3.1. In the optimized version, we use the same clever trick that is used in Culler's sequential loop schema to avoid an array of gates by making use of the switches at the iteration boundary. This still-incorrect schema is shown in Figure 6 for the `fib` code. The schema has very little overhead. In terms of our cost measure, the overhead is approximately $n + 2$ cycles where n is again the number of nextified variables. However, it suffers from the same problem as the one-barrier approach.

The schema has no problem with conflicts within the loop-body because of the signal tree and gating of the predicate into the switches. The setup ensures that tokens for the next iteration are prevented from passing the row of switches until all tokens of the previous iteration have cleared out of the loop-body. There is also no problem with conflict within the loop-predicate region if we ensure that the output of the loop-predicate region is only produced when all computations inside the loop-predicate region have completed. This can be ensured locally within the loop-predicate block.

The problem with the optimized one-barrier schema lies in the part of the loop outside the loop-body and loop-predicate blocks. In particular, the trouble spots are the "loop-back" arcs that recirculate the nextified variables produced in the loop-body back to the switches. Looking at Figure 6, we find that once again, if the tokens of `(next i)` and `(next c)` are processed before that of `(next p)`, a fast recirculating `(next c)` from iteration i can get into the next iteration, become `(next p)` of iteration $(i + 1)$ which recirculates back to the switch for `p` and conflicts with the `(next p)` of iteration i which is still there.

Culler's schema removes such conflicts by holding the nextified values behind the row of gates outside the loop-body until all the nextified values from the previous iteration have passed through the switches. (See Figure 5.) We can, however, do better. Careful examination of the optimized one-barrier schema shows that we only need to hold *each* nextified value behind a gate until *its* previous nextified value has gone through its switch. This alone will ensure that no conflict can occur at the input to each switch. While we still need an array of gates to hold back the `next` values, each gate is triggered by a token carrying the current value

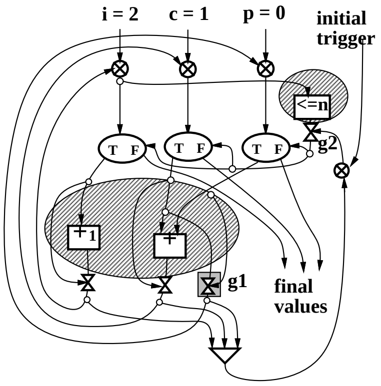

Figure 7: The loop in `fib` wired up under the self-gating sequential loop schema.

of the corresponding nextified variable. Figure 7 illustrates this schema with the `fib` code. The reader should check that no conflict can arise at the input to p's switch as `(next p)` of the $(i+1)^{th}$ iteration is held behind the gate labeled `g1` in the figure until `(next p)` of the i^{th} iteration has moved through its switch. We call this gating of the `next` value of each nextified value with its current value *self-gating.* A formal correctness proof of this schema is given in [1]. This schema avoids one of the signal trees in Culler's schema. We will also see in Section 4.1 that it is easy to optimize the self-gating schema.

The self-gating sequential loop schema incurs an overhead of approximately $4n + 2$ cycles per iteration where n is the number of nextified variables. This is, approximately, a 33% savings over the $6n + 2$ overhead of Culler's schema. Further improvement is possible, and we will see them next.

4.1 Optimization of the Self-gating Sequential Loop Schema

The self-gating sequential loop schema has another big advantage over Culler's schema in that in most cases, it can easily be optimized. Self-gating is not always necessary and it is not difficult to determine this at compile time. Strictness information plays a crucial role in the optimization.

4.1.1 Removing Redundant Self-gates

The self-gating of each nextified variable serves to ensure that during each iteration, the `next` value of a nextified variable does not recirculate back to its switch until its current value has passed through the switch. For most nextified variables, there is data-dependence between the current value and the `next` value such that the `next` value cannot be produced until the current value is available in the loop-body. In such cases, the role performed by self-gating is inherent in the code of loop-body itself, so there is no need to add the self-gating. Another way to look at the data-dependence between the current and `next` values of the nextified variable is that the computation of the `next` value is strict in the current value. This information can be gathered easily by strictness analysis.

In the loop of `fib` for example, the value of `(next i)` is strict in the value of `i`. Similarly for `(next c)` and `c`.

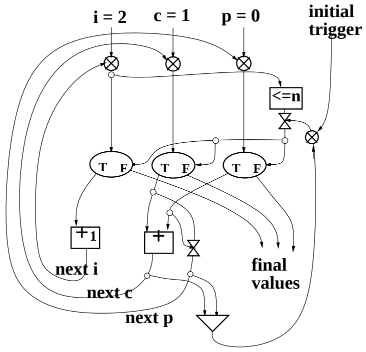

Figure 8: The loop in `fib` wired up under the fully optimized self-gating sequential loop schema.

There is therefore no need to self-gate either of these nextified variables. The resulting dataflow graph, together with the optimization described in the next section, is shown in Figure 8.

4.1.2 Reducing Iteration Termination Signal Tree Size

A second optimization is to reduce the number of inputs to the iteration termination signal tree. Again, we make use of data-dependence that is already present in the code to achieve this. Nextified variables are included in the iteration termination signal tree so that we do not let the loop predicate value get past gate `g2` in Figure 7 until all the `next` are available. But if the predicate computed by the loop-predicate block is strict in the `next` value of a particular nextified variable, we do not need to include that nextified variable in the iteration termination signal tree. In the `fib` example, `(next i)` is required before the predicate $\boxed{\texttt{<=n}}$ can execute. There is therefore no need to include it in the iteration termination signal tree. The dataflow diagram of the `fib` loop including this optimization is shown in Figure 8

For a loop with n nextified variables, m of which need self-gating, and p of which need to be included in the iteration termination signal tree, the overhead introduced by the sequential loop schema is approximate $(p + 3m)$. Usually, m is much smaller than n and frequently 0, while p is smaller than, but closed to n. Overall, this results in very low overhead for the sequential loops. In the `fib` example, the overhead is only 5 cycles per iteration, barely 30% of the 18 cycles overhead incurred under Culler's schema.

5 Frame-based-variable (fbv) Sequential Loop Schema

While the self-gating sequential loop schema is a big improvement over Culler's sequential loop schema, sequential loops are still fairly expensive in Id. Consider again the example that computes $\sum_{i=1}^{n} i$ (Figure 1). Figure 9 shows its implementation in the optimized self-gating schema.

The loop in `sum_1_to` actually incurs very little overhead that is directly attributed to the sequential loop schema. No self-gating is needed and `(next i)` does not need to be included in the iteration termination signal tree because the loop-predicate blocks's output is strict in it. Despite all

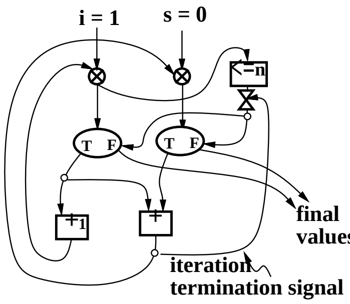

Figure 9: The loop of $\sum_{i=1}^{n}$ wired up under the optimized self-gating sequential loop schema.

these favorable conditions, Monsoon takes 12 cycles to execute one iteration of this loop. A typical piece of RISC code performing the same task takes only 4 cycles. Further improvement is needed and we begin this effort by identifying the sources of inefficiency.

5.1 Remaining Sources of Overhead in Sequential Loops

An examination of Figure 9 shows that processor cycles are incurred by the following that are not present in RISC code: (1) switching, (2) fanout, and (3) matching bubbles.

Switching refers to the fine-grain switching at loop iteration boundary. This mechanism determines if the `next` values of nextified values go to the next iteration or exit the loop. Each nextified variable is switched individually.

Fanout are needed due to architectural constraints. On Monsoon, each instruction can have at most two destinations while some are restricted to only one. If a value is needed at more destinations than the instruction producing it can cope with, we need to use *fanout* instructions to supply the value to all the places where it is needed. A fanout instruction takes an input token and produces two tokens carrying the same value. In general, a tree of fanout instructions is used to duplicate a value sufficiently many times to supply all its destinations.

Bubbles are caused by matching. During matching of two tokens, the first token to arrive does not result in actual execution of an instruction. Nevertheless, Monsoon takes one cycle to process the token. As the ALU part of the pipeline is not used during this cycle, the cycle is known as a *matching bubble* or bubble in short. Thus, each binary operation takes at least two cycles to execute with the first one a bubble.

All the features listed above are needed to support fine-grain parallel execution in the pure-dataflow model. Fanout overhead can be reduced slightly with better architectural support such as the "repeat" feature found on the Epsilon-2 machine[7]. But as long as each operand value is distributed on a token in the conventional dataflow model, switching and matching bubble cycles are inevitable. They are deeply tied to pure-dataflow's model of data transfer and instruction execution.

The analysis yields some good news and some bad ones. The bad news is: if we insist on staying within the pure-dataflow model, there is little we can do to improve the efficiency of loops except perhaps add hardware to perform fanout and matching without using the main processor pipeline. It is not clear how much is gained through such hardware solutions when the cost of hardware is taken into account. On the other hand, if we are willing to go beyond the pure-dataflow model by adopting some elements of von Neumann style execution, we can improve the efficiency of most sequential loops through a different compilation strategy. This is the basis of our next approach.

5.2 The Frame-based-variable (Fbv) Approach

This new implementation of sequential loops is motivated by the observation that sequential loops *must* synchronize at the loop boundary. The presence of this synchronization makes some of the matching in the loop-body itself redundant.

Specifically, all the `next` values of nextified variables are synchronized at the end of each iteration. When these values cross over to the next iteration, they become the current values of the nextified variables. Thus, by the time the next iteration starts executing, we already know that the current values of nextified variables are available, and there should be no need to synchronize(match) before their uses. If we can somehow harness this property, we should be able to reduce the number of matching bubbles.

There is also no point in switching each nextified value individually. Fine-grain switching is done so that computations needing only the switched value can proceed as soon as possible. However, due to the barrier at iteration boundary, we are not making use of this benefit of fine-grain switching at all. As fine-grain switching is fairly expensive, it will be a great saving if we can get rid of them or reduce their number.

5.2.1 A Mixed Dataflow-von Neumann Execution Model

In order to reap the benefits outlined above, we need an execution style with "von Neumann" flavors: the values of nextified variables are passed *via frame memory locations* instead of on tokens. `Next` values are stored directly into frame memory locations, and uses of a nextified variable simply read from these locations. The approach shares some similarities with multithreading described in [18]. With such an approach, we only need to retain one nextified variable that is passed via tokens and switched at the iteration boundary to initiate each iteration. As most nextified variables are stored explicitly in frame slots, this approach is called the *frame-based-variable (fbv)* sequential loop schema.

Ideally, we would like the operations that use nextified variables to read the frame memory locations directly, and those that produce the `next` values to store them into frame memory directly. Unfortunately, this requires a frame-memory to frame-memory 3-address instruction set that is not supported by Monsoon. It is also doubtful whether such instructions can be supported efficiently on real machines. Memory hierarchy to support efficient fine-grain execution is an interesting and important topic but is unfortunately beyond the scope of this work.

Coming back to Monsoon, storing into frame memory has to be done via an explicit store instruction. For reads, however, we can at times avoid identity instructions, which are Monsoon's equivalent of explicit load instructions. Monsoon supports a one-address accumulator style code. In the case where a nextified variable is used in a binary operation which has a token input as the other operand, Monsoon is

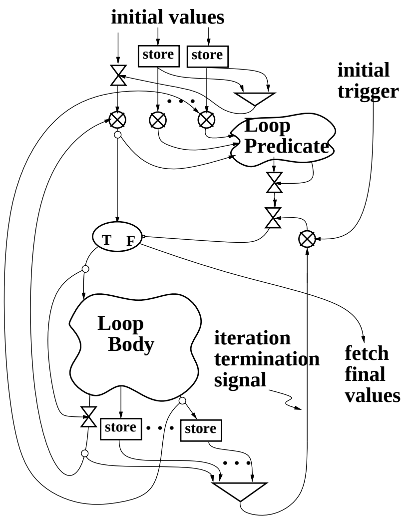

Figure 10: The skeleton of the frame-based-variable (fbv) sequential loop schema. This is not complete. Additional mechanisms is needed to prevent WAR hazards.

able to read the value of the nextified variable from memory directly. In cases where both inputs of an instruction are from frame memory, we need to insert an identity instruction that fetches one of the operands explicitly. The identity instruction will have to be *triggered*, i.e. have a trigger input, as the arrival of a token is the only way to initiate execution on Monsoon. A trigger in this case is similar to a trigger to a gate instruction in that the token is merely used to enable an operation.

The general fbv sequential loop schema is shown in Figure 10. Self-gating of the switched nextified variable can be optimized out as usual if the data-dependence check allows it. As before, we need to collect an iteration termination signal. For nextified variables that are not switched, the signal outputs of the store instructions are fed into the iteration termination signal tree. We cannot optimize any of these out on the ground that the loop-predicate is strict in that nextified variable. This optimization only apply to switched nextified variables.

`Next` values that are used in the loop-predicate block are still supplied via tokens. That accounts for the re-circulation arcs, but there are usually relatively few of them as nextified variables not used in the loop-predicate block are not recirculated. The initial values of the nextified variables have to be stored before the loop can start execution. This is done by the row of store instructions at the top of Figure 10. Lastly, the final values returned by the loop need to be fetched explicitly.

5.3 Data Hazards in the Frame-based-variables Approach

The fbv sequential loop schema described in the last section suffers from a major complication arising from the parallel execution of code *within* the loop-body. The body of each iteration is executed in parallel in a sequential loop, even though the iterations of the loop are executed one at a time. Under the fbv approach, each frame slot used to store a nextified variable is both read from and written into every iteration. For correctness, we have to make sure that every use of a nextified variable reads the current value and not the `next` value. But because of parallel execution in the loop-body, this is not guaranteed automatically and it is possible to have *Write After Read (WAR) hazards*[3] where the update occurs too soon.

A nextified variable suffers from WAR hazard if there is a use of the nextified variable that is *not* guaranteed to occur before the `next` value of the nextified variable is computed. A use is guaranteed to occur first if it is strict for the computation of the `next` value. There are several ways to deal with this problem; we explored two general solutions: (1) delay the store of the `next` value until it is safe to perform the store; (2) use two frame slots for each nextified variables, one for reads, one for the write, moving the value from the write slot to the read slot at the iteration boundary.

5.3.1 Fbv hazard solution 1: Delaying the Store

We can avoid WAR hazards by delaying the store of each `next` value with a suitably triggered gate, i.e. insert a gate instruction on the arc supplying the value to be stored. We call each gate instruction used this way a *store-gate*, and this solution, the *gated-store approach*. The trigger input to each store-gate is carefully chosen so that its availability signals the completion of all the reads from the corresponding frame slot, making it safe for the store to proceed.

For a nextified variable `x` that suffers WAR hazard, the most straight forward way to obtain a trigger for its store-gate is to perform an explicit fetch of `x` from its frame slot. The output of this fetch is supplied to all the uses of `x` that are not strict for `(next x)`, and also used to trigger the store-gate of `x`. This ensures that the store will never occur until all reads from `x`'s frame location are done. Figure 11 shows the fbv sequential loop schema with this approach to handle WAR hazards. Storage of the initial values of nextified variables is left out in the figure to avoid clutter.

The gated-store solution described above can be optimized. The gated-store solution inserts an explicit fetch for each nextified variable that has a hazard problem. Each of these fetches is triggered with an input that initiates the fetch. By using the output of a fetch to trigger another fetch and carefully ordering the fetches, we can add data-dependence between uses of nextified variables and their respective `next` values without changing the overall semantics of the program. Such added data-dependences can remove some of the WAR hazards.

Figure 12 show how this is applied to the loop in `fib`. The left side of the figure shows the loop without this optimization. As both `p` and `c` have uses that are not strict for `(next p)` and `(next c)` respectively, they need store-gates as shown. With clever choice of triggers for the fetches, however, we can trigger the fetch of `c` with the output from the fetch of `p`. As `(next p)` is dependent on the output of this fetch of `c`, this choice of fetch triggers effective makes `(next p)` dependent on the only use of `p`, making it unnecessary to gate the store of `(next p)`. The resulting dataflow graph is shown on the right side of Figure 12.

[3]Write After Read hazard is a term used in RISC literature to refer to incorrect ordering of memory operations where a write that should occur after a read from the same location is allowed to occur before the read. The read ends up reading a newer value stored to that location.

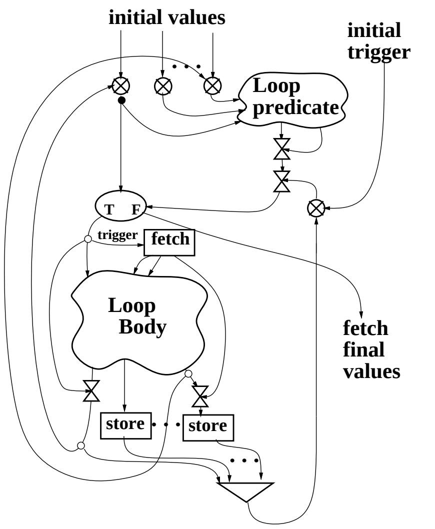

Figure 11: The frame-based-variable sequential loop schema with the gated-store, explicit fetch, approach to dealing with WAR hazards.

5.3.2 Fbv hazard solution 2: Using 2 frame slots.

The second solution to the hazard problem uses 2 frame slots for each nextified variable. In the loop-body, we always read from one location and store into the other. At the end of each iteration, we move the values from the store-locations to the read-locations before allowing the next iteration to start. Figure 13 shows the resulting fbv sequential loop schema. We will refer to this solution as the *2-slot approach*.

This solution is very simple in that the compiler needs to do almost no analysis to implement it. Surprisingly, even though a move from one frame memory location to another takes two instructions on Monsoon, the solution is still fairly efficient as there is no gating whatsoever.

The one drawback of this schema is that the string of fetch-store instructions performing the moves in-between iterations can result in rather long latency. This is particularly bad for Monsoon which is very sensitive to a single long thread. Unlike most pipelined RISC processors, each Monsoon processor interleaves eight independent thread of computation, such that a single thread will only occupy one out of every eight cycles. Latency of an n instruction thread is therefore $(8 \times n)$. In addition, eight-fold parallelism is needed on each processor to keep it busy. By linearizing work into a thread, the 2-slot approach reduces parallelism. We will see some of these effects in the next section when we present some run time statistics.

6 Run Time Statistics on Monsoon

In this section, we present the run time statistics of eight programs compiled using four different sequential loop schemata: Culler's schema, self-gating schema, gated-store-fbv and 2-slot-fbv. The programs are run on a Monsoon configured with one processing element (PE) and one I-Structure (IS) board. The statistics are gathered with hardware support and is non-intrusive to the first order[16, 12]. We will first briefly describe the eight programs before examining the statistics.

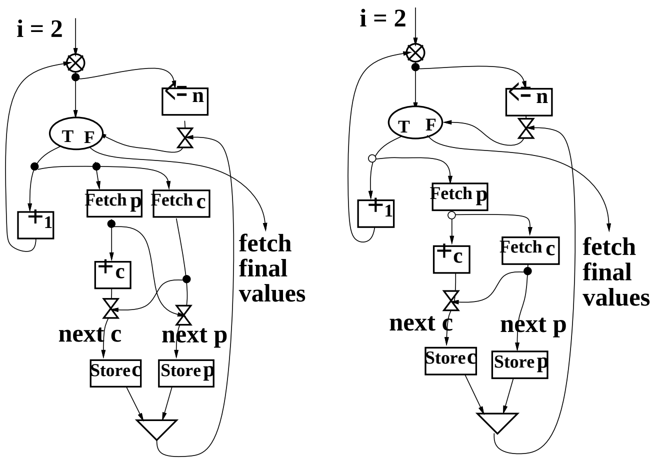

Figure 12: `fib` wired under the fbv sequential loop schema with the general gated-store, explicit fetch, approach to dealing with WAR hazards. The dataflow graph on the left is without optimization while the one on the right optimizes the triggering order.

6.1 The Benchmarks

The following benchmarks are used to compare the different sequential loop schemata. Each has a sequential loop as its innermost loop.

1. Matrix Multiply creates two $n \times n$ double-precision floating point matrices, computes their product matrix and returns the sum of the elements of the product matrix as its result. The multiplication of matrices is coded up as straight-forward, triply nested loops with the innermost loop computing a dot-product.

2. Blocked MM is similar to Matrix Multiply except that it is blocked so that each iteration of the innermost loop computes the inner-product of a 4×4 block of the result matrix.

3. Simple[4] is a hydrodynamics and heat conduction simulation program. It uses a Lagrangian formulation of equations to simulate the behavior of fluid in a sphere. Unlike the other benchmarks described here, this is a sizable program and is representative of a class of numerical programs.

4. Paraffins enumerates all the distinct isomers of every paraffin[4] with n or fewer Carbon atoms. The program generates lists of paraffins and finally returns an array filled with the number of distinct paraffins of each size up to and including the maximum size specified by the user.

5. Fib computes the n^{th} Fibonacci number. The code is listed in Section 3.1 and does the computation with a sequential loop. It runs in $O(n)$ time.

[4]Paraffins are molecules with chemical formula C_nH_{2n+2}, where C and H stand for carbon and hydrogen atoms, respectively, and $n > 0$.

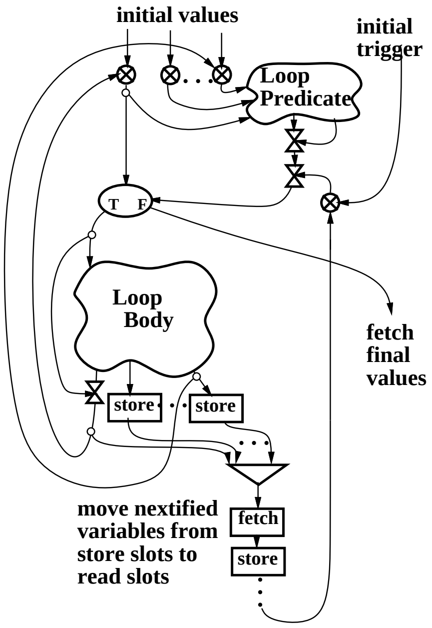

Figure 13: The frame-based-variable sequential loop schema with the 2-frame-slot-per-nextified variable approach to take care of WAR hazards.

6. Merge Sort generates an array of n random numbers and then proceeds to sort it using the merge sort algorithm. The program is recursive but uses a loop to perform merging.

7. Stats generates an array of size n, fills it with random numbers and then finds the minimum, maximum, and average of the values in the array.

8. FFT computes the Fast Fourier Transform of a degree n polynomial. It takes $O(n \log n)$ time.

6.2 The Results

Table 1 shows the results. Due to the lack of space,the data is highly compressed. For each program, three pairs of normalized numbers, each corresponding to using a new sequential loop schema during compilation, are reported. The two numbers in each pair are separated by a slash "/". The first, *token-count*, corresponds to the total number of tokens processed by the Monsoon processor and can be thought of as the useful work done. The second corresponds to the elapsed time for running the program, and is call the *critical path length*. The difference between the two numbers is due to Monsoon's 8-way interleaved pipeline which requires eight fold parallelism at any time to keep a processor busy. When there is not enough parallelism, idle cycles are incurred, which are included in the critical path length but not the token-count. All the numbers are normalized, using the token-count and critical path length of the same program compiled with Culler's schema as basis.

While critical path length is important if we are interested in how fast each program runs on its own, the token-count matters more here as most of these benchmarks are

Program	Self-gating	Gated-store-Fbv	2-slot-Fbv
Matrix Multiply	0.68/0.71	0.53/0.55	0.53/0.55
Blocked MM	0.69/0.70	0.51/0.53	0.51/0.53
Simple	0.96/0.96	0.94/0.95	0.96/0.96
Paraffins	0.88/0.89	0.89/0.90	0.88/0.89
Fib	0.71/0.75	0.35/0.80	0.41/1.06
Merge-sort	0.92/0.93	0.88/0.91	0.91/0.94
Stats	0.79/0.82	0.66/0.74	0.66/0.75
FFT	0.90/0.92	0.87/0.89	0.88/0.91

Table 1: Comparison of various Sequential Loop Schema. Each box contains 2 numbers: normalized token-count/critical path length.

small toy routines that we expect to find as inner loops of larger programs. Real programs are expected to be larger and have several instances of the inner-loops executing at the same time which together provide the required parallelism. We should, nevertheless, try to keep the critical path lengths closed to the token-counts so that the amount of parallelism that has to be provided by other code running at the same time is kept low.

The token-count of every program is lower when compiled with any of the three new sequential loop schemata than with Culler's schema. In addition, the fbv schema with gated-stores gives the best run time in almost every cases. Improvements ranges from 6% to 65%. The low improvement for Simple can be explained by noting that Simple contains loops with very few nextified variables, and also fairly large loop bodies. This means that the loop overhead was relatively small to start with, and hence there is little scope for improvements.

The critical path length of the benchmark programs shows similar improvement. The only exception is Fib compiled under the 2-slot fbv schema where the critical path is actually *longer* than the same program compiled under Culler's schema. This example illustrates the short-coming of the 2-slot fbv schema that we mentioned earlier, namely that of a long single thread where data is move between the write and read slots. Gated-store fbv is still the best in most cases. In other cases it is also not too far off from the best timing.

Based on the run time for our benchmarks, the gated-store fbv sequential loop schema emerges as the best sequential loop schema among those that we have studied, having both low token-counts and critical path lengths.

7 Conclusion and Future Directions

We explored various ways of implementing sequential loops efficiently on dynamic dataflow machines that perform matching with frame memory. Several other good schemata were not examined due to lack of space. We will quickly point out a few of them, although we believe that what we have presented in this paper are at least as good. One way of reducing the sequential overhead of 2-barrier approach is to use 2 copies of code, with a barrier between each copy. In addition, the gates in the barriers can be removed by simply gating the switch predicate. We believe that the self-gating sequential loop schema is as efficient as this for most cases, given that most nextified variables are strict in their current values. By using the self-gating schema, we avoid the increases in code and frame sizes that accompany duplication of code. The 2-slot fbv can also be improved by duplicating code. This avoids the explicit moves between the two slots by having each copy use opposite slots for reading and writing.[5]

This work also pointed out very clearly the shortcomings of pure dataflow execution. The comparison of pure dataflow execution with RISC style code in Section 5.1 is applicable to dataflow execution in general. The overheads of switching, fanout and matching bubbles is *inherent* in the pure dataflow execution model. These problems have already been noted by other researchers, and means of overcoming them have been suggested by Iannucci[10], Culler[19], Traub and Papadoupolos[18, 20]. In general, a fundamental change in the way data is communicated, from *only* via tokens to using memory as well is needed. This together with *threading* by the compiler allows redundant synchronization to be removed.

The *T project[13], jointly undertaken by our research group at MIT and Motorola, will be investigating and testing these approaches on a real machine. Some interesting open questions remain, including what sort of memory hierarchy, instruction set and scheduling will efficiently support this style of execution. It appears doubtful that the traditional RISC style three-address register-to-register instruction sets with generic general-purpose registers is good match for this style of execution due to the expected high frequency of thread switch.

Acknowledgements

Funding is provided in part by the Advanced Research Projects Agency of the Department of Defence under Office of Naval Research contract N00014-89-J-1988.

References

[1] B. S. Ang. Optimization of Loops for Dynamic Dataflow Machines. Master's thesis, MIT, EECS, Laboratory for Computer Science, December 1992.

[2] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[3] D. T. Chiou. Activation Frame Memory Management for the Monsoon Processor. Master's thesis, MIT, EECS, Laboratory for Computer Science, September 1992.

[4] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE Code. UCID 17715, Lawrence Livermore Laboratory, February 1978.

[5] D. E. Culler. *Managing Parallelism and Resources in Scientific Datalfow Programs.* PhD thesis, MIT, EECS, Laboratory for Computer Science, June 1989.

[6] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[7] V. G. Grafe and J. E. Hoch. The Epsilon-2 Multiprocessor System. *Journal of Parallel and Distributed Computing*, 10(4):309–318, 1990.

[8] J. E. Hicks. Id Compiler Back End for ETS and Monsoon. CSG memo 310, MIT Lab for Computer Science, Cambridge, MA, June 1990.

[9] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada, and T. Yuba. The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems. *Journal of Information Processing*, 10(4):219–226, 1987.

[10] Robert A. Iannucci. *Parallel Machine, Parallel Machine Languages: The Emergence of Hybrid Dataflow Computer Architectures.* Kluwer Academic Publishers, 1990.

[11] Y. Kodama, S. Sakai, and Y. Yamaguchi. A Prototype of a Highly Parallel Dataflow Machine EM-4 and its Preliminary Evaluation. *Proceedings of InfoJapan*, 10(4):291–298, 1990.

[12] Venkat Natarajan, Derek Chiou, and B. S. Ang. Performance Visualization on Monsoon. *Journal of Parallel and Distributed Computing*, May 1993. (To appear).

[13] R. H. Nikhil, G. M. Papdopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th ISCA, Gold Coast, Australia*, May 1992.

[14] R. S. Nikhil. Id Reference Manual, Version 90.1. CSG memo 284-2, MIT Lab for Computer Science, Cambridge, MA, September 1990.

[15] R. S. Nikhil and Arvind. Id: a Language with Implicit Parallelism. CSG memo 305, MIT Lab for Computer Science, Cambridge, MA, February 1990.

[16] G. M. Papadopoulos. Program Development and Performance Monitoring on the Monsoon Dataflow Multiprocessor. In *Proceedings of the Workshop on Instrumentation for Future Parallel Computing Systems.* ACM Press, 1989.

[17] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* Research Monograph in Parallel and Distributed Computing. MIT Press, 1992.

[18] G. M. Papdopoulos and K. R. Traub. Multithreading: A Revisionist View of Dataflow Architectures. In *Proceedings of the 18th ISCA, Toronto, Canada*, May 1991.

[19] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the 5th FPCA, Cambridge, MA*, pages 50–72, 1991. (Springer-Verlag LNCS 523).

[20] K. R. Traub. Multi-thread Code Generation for Dataflow Architectures from Non-Strict Programs. In *Proceedings of the 5th FPCA, Cambridge, MA*, pages 73–101, 1991. (Springer-Verlag LNCS 523).

[5] I would like to thank one of the referees for pointing out these possibilities.