Static Scheduling for Dynamic Dataflow Machines

Micah Beck*
Keshav K. Pingali*
Alex Nicolau**

TR 90-1076
January 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Research supported by an NSF Presidential Young Investigator award (NSF grant #CCR-89-58543), and grants from the General Electric Corp. and the Math Sciences Institute of Cornell.

**Research supported by ONR grant #N00014-88-K-0310 and NSF grant #CCR-87-04367.

Static Scheduling for Dynamic Dataflow Machines

Micah Beck[1]
Keshav K. Pingali[1]
Cornell University

Alex Nicolau[2]
University of California, Irvine

Abstract

Dataflow machines can "unravel" loops automatically so that many iterations of a loop can execute in parallel. Unbounded loop unraveling can strain the resources available on the machine and, in extreme cases, deadlock can occur due to overcommitment of resources. Previous efforts to address this problem have focused mainly on runtime mechanisms of debatable utility. Loop bounding, a compile-time technique, controls parallelism by introducing dependencies between loop iterations. The loop is given enough resources for the concurrent execution of some number of iterations, say k. The $k+1$st iteration uses the same resources as the first iteration and starts only after the first iteration is complete, and so on. Thus, the granularity of resource allocation is based on the rather arbitrary syntactic notion of a loop iteration.

In this paper, we argue that loop bounding can lead to inefficient use of resources and propose an alternative way of compiling loops for pipelined execution. We introduce the notion of a *stage decomposition* of a loop, which defines a partition of a loop iteration into stages. We show how the problem of choosing a stage decomposition for a particular loop can be tackled by applying compile-time analyses and static scheduling techniques. Such techniques have been developed for scheduling loops on very long instruction word (VLIW) machines which, like dataflow machines, can exploit fine-grained parallelism in programs. These analyses permit the compiler to allocate resources according to expected patterns of usage, thus reducing overall resource requirements. Finally, we show how our schema can be implemented on the Monsoon dataflow machine being built at M.I.T.

[1]Research supported by an NSF Presidential Young Investigator award (NSF grant #CCR-89-58543), and grants from the General Electric Corp. and the Math Sciences Institute of Cornell.

[2]Research supported by ONR grant #N00014-88-K-0310 and NSF grant #CCR-87-04367

1 Introduction

Executing a program in parallel takes more resources than executing it sequentially. Consider the expression $f(x) + g(y)$ where the two function calls can be executed in parallel. If R_f and R_g denote the resources which must be allocated to execute $f(x)$ and $g(y)$ respectively, parallel execution of the two function calls can require resources $R_f + R_g$. Under sequential execution, the resource requirement is $\max(R_f, R_g)$, which is less than the resources required for parallel execution. When resource demands become excessive, the efficiency of execution may be reduced as the machine thrashes between parallel tasks and, in extreme cases, program execution can deadlock. Therefore, parallel machines must tread a fine line between exposing *too little* parallelism, thereby increasing execution time, and exposing *too much* parallelism, thereby straining resources.

This problem is particularly critical in dataflow machines which attempt to exploit parallelism at all levels in functional language programs. In most programs, the resulting resource demands are enormous and any realistic scheme for executing programs on dataflow machines must include some mechanism for controlling parallelism [CA88]. One possibility is to build in runtime mechanisms that control program execution. Typically, these mechanisms maintain an execution profile that shows the status of each task as well as relationships between tasks (such as which task spawned a given task). This profile is used to either grant or defer requests to spawn new tasks. A variety of resource management policies have been studied such as granting a request only if there are sufficient resources, if the task to be spawned is on the critical path, or if the requesting task is the lowest, leftmost task in the task tree [Rug87]. We believe that some such runtime machinery will be necessary, although at this point, no clear consensus has emerged yet as to what the best runtime mechanism or task management policy is. However, every runtime mechanism has some overhead and it is preferable to use compiletime mechanisms wherever possible.

In this paper, we address the problem of controlling the execution of *loops* in dataflow programs. Even on pipelined, von Neumann machines, efficient use of parallel hardware requires the use of *software pipelining* techniques which enable overlapped execution of multiple iterations of the loop. This effect is obtained on dynamic dataflow machines through "loop unraveling" — in effect, a loop can unfold dynamically so that an unbounded number of its iterations are active concurrently. Unfortunately, Arvind and Culler have shown that unbounded loop unraveling can lead to inefficient use of enormous amounts of resources [CA88]. They proposed a compilation technique called *loop bounding* to get around this problem. In effect, loop bounding adds dependencies to the loop so that only a small number of loop iterations are executing at any time [CA88]. As we discuss in Section 2, loop bounding is very ad hoc — no use is made of information about dependencies between operations in the loop. In Section 3, we present a new scheme for compiling loops for execution on dynamic dataflow machines. We define the notion of a *stage decomposition* of a loop and show how such a decomposition can be used to achieve overlapped execution of loop iterations. We also introduce the notion of an optimal decomposition. In Section 4, we show how an optimal decomposition can be derived using loop scheduling techniques developed for very long instruction word (VLIW) architectures. In Section 5, we deal with the complication introduced into our schema by operations with long latencies such as loads. Our use of VLIW scheduling techniques in Section 4 is based on an assumption of fixed latency for all operations. In Section 6 we show how such fixed latencies can be estimated for a dynamic dataflow architecture in which latencies are variable and unbounded. To judge the utility of the resulting compilation schema, we would have liked to perform experiments on a real dataflow machine. In the absence of such a machine, we will be content to argue that the schema proposed in this paper is reasonable and should be subjected to experimentation when dataflow machines become available.

2 Loop Unraveling and Loop Bounding

Consider the following simple program that computes the inner product of two vectors:

$$
\begin{array}{l}
sum := 0 \\
\textbf{for } i = 1 \textbf{ to } N \\
\quad t1 := X[i] \\
\quad t2 := Y[i] \\
\quad t3 := t1 * t2 \\
\quad sum := sum + t3
\end{array}
$$

How would this program execute on a pipelined von Neumann machine like the Cray? Before an instruction can be issued, its operands must be available and its destination register must be free. Moreover, instructions are issued *in order*; if an instruction cannot be issued at some point in time, all instructions behind it are also held up. Therefore, in our example, issue of the add instruction from iteration i must precede issue of instructions from the $i+1$th loop iteration. Since the add instruction from iteration i can be issued only after the two LOAD's and the multiply instruction of that iteration have been completed, there is little overlap of execution between iterations of this loop. In particular, the memory system will be processing at most two LOAD requests at any time, even if it could sustain many more concurrent operations. This can severely limit parallelism, particularly if LOAD operations take a long time to execute.

To execute this loop faster, the processor must be able to interleave the issue of instructions from different iterations. On dynamic dataflow machines like Monsoon[Pap88], this is accomplished by "loop unraveling". In brief, each iteration is given its own activation frame. This frame, like a stack frame in von Neumann computers, consists of all the operand memory required for the execution of that iteration. Since each iteration gets its own frame, an iteration can begin executing as soon as the loop index for that iteration is computed. Unfortunately, because of data dependencies between iterations, the iteration may make little progress in performing computation, even though a frame has been allocated for its use. Thus, unbounded loop unraveling can result in the unproductive use of frame memory. For an example of this, consider Figure 1 which shows the dataflow graph corresponding to the inner product example. This example also introduces some of the dataflow operators we need for our schema.

Execution of the loop begins when tokens are placed on the input arcs of the graph. These tokens are the initial inputs to the CHANGE TAG operators and the predicate. If the predicate evaluates to true, the CHANGE TAG operators send their input tokens into the body of the loop. When the loop index i passes through the ALLOC operator, a new memory frame is allocated for the next iteration and a token whose value is the tag associated with this new frame is generated[3]. We refer to a value which is calculated in each iteration and then passed into the next iteration as a "circulating value", or "nextified variable" in Id terminology. The CHANGE TAG operator is used to pass such values between iterations. When a circulating value passes through the CHANGE TAG operator, one of two things happens.

1. If the loop predicate is true, then the token which carries the circulating value is assigned the tag of the new frame, and it thus "passes into" that frame.

2. Otherwise, the circulating token is assigned the tag of the parent frame, and it thus passes out of the loop as a result value.

Note that this use of the CHANGE TAG operator implements two functions: it serves to switch tokens between the next iteration and the parent context, and it also delays a value from passing into the next iteration until a frame has been allocated for it. This dual function is central to the implementation of the "switching throttle" which is used in the bounded loop schema presented later in this section.

Let us see how this dataflow graph can "unravel" on a dynamic dataflow machine. Suppose that the latencies of the two LOAD operations is large compared to the latencies of arithmetic operations and CHANGE TAG. In that case, a token carrying the value two will be produced on the line labeled "next i" long before a token appears on the line labeled "next *sum*", and a frame will be allocated for the second iteration long before there is any computation to be performed there. In fact, an unbounded number of

[3]In reality, the ALLOC operator is a complicated schema which communicates with the operating system. However, for our discourse, it is sufficient to think of it as a single operator.

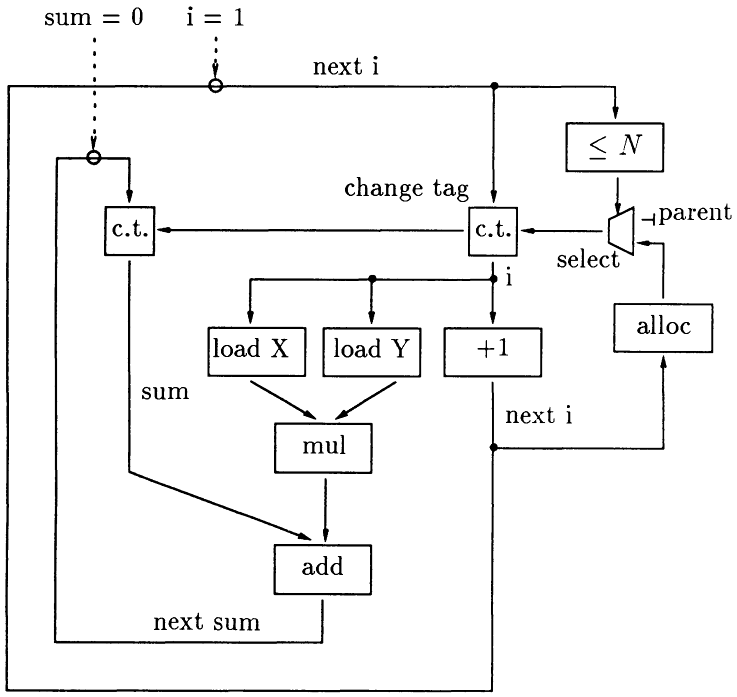


Figure 1: Inner Product Dataflow Graph

frames can be allocated concurrently, even if computation is proceeding only in a small number of them.

This might suggest that loop unraveling is harmful and should be prohibited. On the other hand, notice that the LOAD operations in an iteration can begin execution once the frame for that iteration has been allocated. This is desirable because of the large latencies of the LOAD operations. Thus, loop unraveling gives us overlapped execution of loop iterations on dynamic dataflow machines, but at the price of unbounded resource requirements.

2.1 Loop Bounding

One compromise is to allow a limited form of loop unraveling. This is the idea behind *loop bounding* [CA88]. Loop bounding allocates a fixed set of k memory frames to successive loop iterations. The $k+1$st iteration uses the same frame storage as the first iteration and is allowed to start only after the first iteration is complete, and so on. In this way, a sliding window of k iterations are allowed to execute concurrently. The number of concurrent iterations k can be chosen at runtime. Figure 2 shows a schematic view of the synchronization in a bounded loop (for $k = 3$).

The bounded loop schema for an explicit token store machine like Monsoon is shown in Figure 3. In this picture, the loop is bounded to a window of three iterations: the first iteration executes in frame 1, the second in frame 2, the third in frame 3, the fourth in frame 1 again, and so on. The main concern is to ensure that a frame is empty before it is reallocated. In particular, circulating values must be sent into a frame only when the previous iteration executing in that frame has completed. This synchronization is accomplished by placing a *switching throttle (s-throttle)* at the exit point of each frame. The switching throttle corresponds to that portion of the dataflow graph depicted in Figure 1 which synchronizes and directs the circulating values. The implementation of the switching throttle is illustrated in Figure 4. Its behavior is exactly as described earlier: it either delays the circulating values until the frame for the next iteration is available (as indicated by a "release signal"), or else it switches them to the parent context, depending on the value of the loop predicate.

The s-throttle serves another purpose not previously discussed: detecting termination of activity in each frame. As the circulating values pass

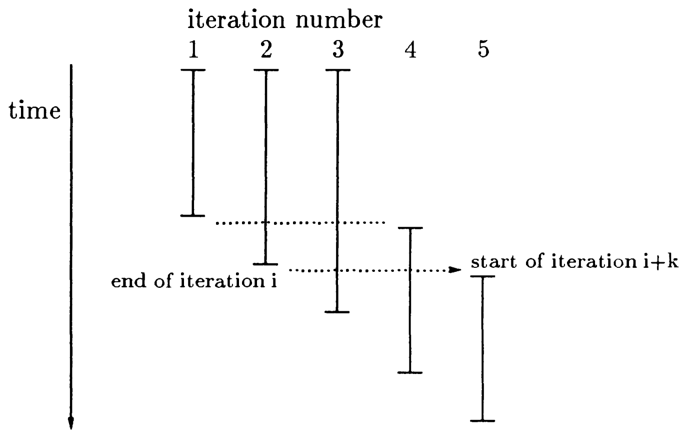


Figure 2: Synchronization in a Bounded Loop ($k = 3$)

through the s-throttle, they are combined in a signal tree which fires when all have passed through. Any activity in the frame which does not generate a circulating value upon termination (such as a write to memory) must also generate a completion signal which is added to this tree. In order to ensure proper synchronization, the release signal of each s-throttle is wired to the termination signal of the s-throttle in the following frame. The release of each s-throttle is initialized with a token, since every frame is initially empty.

Limiting the unraveling of a loop to a fixed number of concurrently executing iterations can increase the length of the critical computation path through the loop. However, as we will show in Section 4, if one assumes fixed durations for all loop operations *including loads*, then a minimal value of k can be determined which does not increase the critical path through the loop. If one can choose worst case durations for all operations, then it is possible to calculate a value of k which will never increase the length of the critical path. If one chooses expected durations, then an value of k can be determined which can be expected not to increase the length of the critical path. The applicability of such approximate models is discussed in

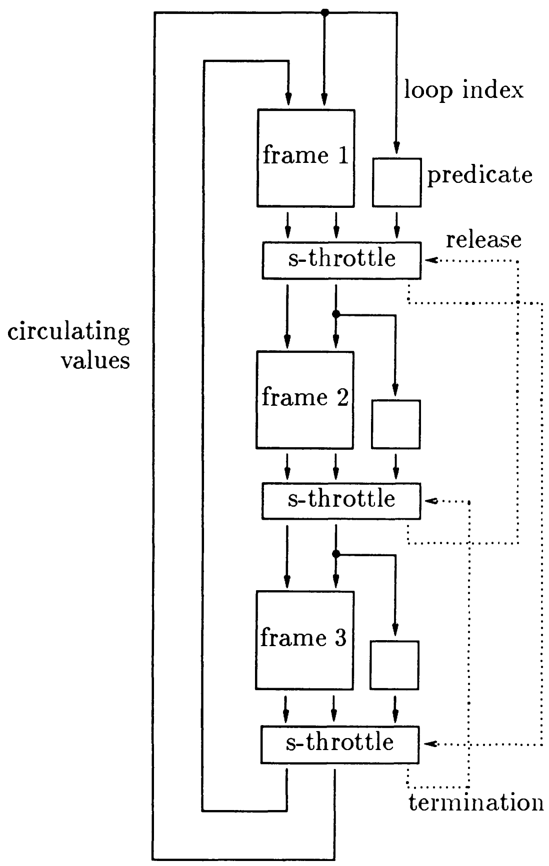


Figure 3: Schematic Bounded Loop ($k = 3$)

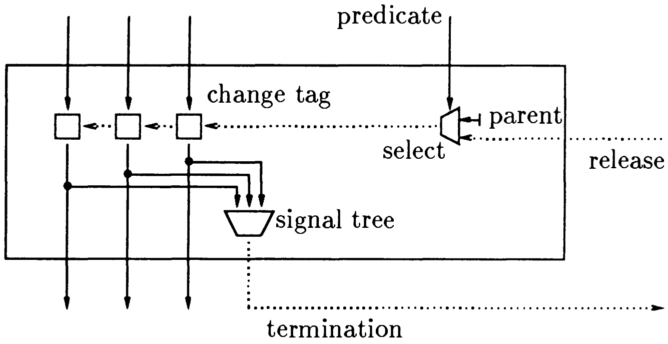


Figure 4: Block Diagram of Switching Throttle

Section 6.

Unfortunately, even under the assumption of fixed operation latencies, loop bounding may not do a good job of allocating frame memory. The reason for this is that loop bounding allocates memory on the basis of frames which represent entire iterations, whereas execution proceeds instruction by instruction. In our inner product example, the memory locations used by the LOAD operations will remain allocated until the add has completed, although it will not be used after the multiply has executed. In the next section, we discuss a completely different approach to solving the problem of unbounded loop unraveling, which does not suffer from this drawback.

3 Loop Pipelining

In this section, we discuss a loop compilation schema that uses an analysis of the dependencies between loop operations, together with assumptions about operation latencies, to distribute into *stages* the operations which comprise an iteration. Intuitively, execution of a loop with n stages is similar to that of a hardware pipeline with n stages. The resulting schema schedules the use of frame memory more effectively than loop bounding can.

3.1 Loop Dependencies

The central tool used in reasoning about the fine-grained scheduling of loops is that of *dependencies* between operations. A dependency between operations A and B implies that A must complete before B is initiated. In dataflow graphs, the only dependencies are *data dependencies*, which exist when the output of one operator is used as the input for another. We consider two types of data dependencies: loop-independent and loop-carried.

A *loop-independent* dependency is between operations in the same iteration of a loop. Thus, in our inner product example, there is a loop-independent dependency between the multiply and add operators C and D. A *loop-carried* dependency exists between operations in successive loop iterations. In our example, the only loop-carried dependency is between the addition operator and itself, since the output of the addition in each iteration is sent to the addition in the next iteration.

We represent a set of loop operations and the associated dependencies as a graph with labeled nodes:

Definition 1 *A* dependency graph *is a four-tuple* $\langle \mathcal{O}, E_I, E_C, l \rangle$ *where*

1. $\mathcal{O}$ *is a set of identifiers for loop operations,*

2. $E_I, E_C \subseteq \mathcal{O} \times \mathcal{O}$ *are sets of edges representing loop-independent and loop-carried dependencies, and*

3. $l : \mathcal{O} \rightarrow \mathcal{Z}^+$ *is a function which assigns a latency to each operation.*

$sum := 0$
for $i = 1$ **to** N
A: $t1 := X[i]$
B: $t2 := Y[i]$
C: $t3 := t1 * t2$
D: $sum := sum + t3$

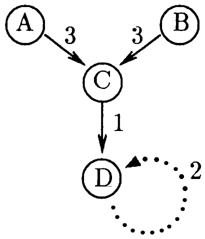


Figure 5: Inner Product Dependency Graph

Figure 5 illustrates the dependency graph for our inner-product example, with latencies chosen arbitrarily for purposes of illustration. In Section 6, we discuss how these latencies can be estimated on a real machine. The loop-carried dependency is shown as a dotted arc, and each arc is annotated with the latency of the associated operation.

3.2 Stage Decomposition

We now consider a loop execution scheme which controls loop execution at a finer level than loop bounding. The schema is parameterized by a decomposition of the loop operations into *stages.* Each operation must occur in exactly one stage.

More formally, we have the following definition.

Definition 2 *A* stage decomposition $D = \{\mathcal{O}_1, \mathcal{O}_2, \ldots\}$ *of a dependency graph* $G = \langle \mathcal{O}, E_I, E_C, l \rangle$ *is a finite sequence of disjoint subsets of* $\mathcal{O}$ *such that* $\cup_i \mathcal{O}_i = \mathcal{O}$.

To simplify our discussion, we will impose two restrictions on dependencies.

Definition 3 *A stage decomposition* D *is* well-formed *iff every operation* $o \in \mathcal{O}_i$ *satisfies the following two conditions:*

1. *A loop-independent dependency must be between operations in either the same stage or in adjacent stages.*

$$\forall \langle o', o \rangle \in E_I \; [o' \in (\mathcal{O}_i \cup \mathcal{O}_{i-1})]$$

2. *A loop-carried dependency must be between operations in adjacent stages.*

$$\forall \langle o', o \rangle \in E_C \; [o' \in \mathcal{O}_{i+1}]$$

The first condition will be relaxed when we discuss long latency operations in Section 5. The second condition can be enforced by adding "dummy" operations with zero latency to carry the dependency between iteration stages [4].

One possible stage decomposition of our example inner product loop would be:

$$\mathcal{O}_1 = \{A, B\}, \quad \mathcal{O}_2 = \{C\}, \quad \mathcal{O}_3 = \{D\}$$

This decomposition is not well-formed, since there is a loop-carried dependency from an operation in $\mathcal{O}_3$ back to itself, which violates condition 2 above. However, it can be made well formed by adding a dummy operation to $\mathcal{O}_2$ to carry this dependency. For now, we will consider the choice of stage decomposition to be arbitrary, but in Section 4 we will show how stages can be chosen to minimize the length of the critical path of computation.

Given such a stage decomposition, loop pipelining is implemented by synchronizing the start of stage j of iteration $i + 1$ with the completion of stage j of iteration i. Figure 6 shows a schematic view of this synchronization and Figure 7 illustrates how this synchronization is implemented. Because synchronization in this schema is performed on the basis of stages rather than iterations, the two functions of the s-throttle are separated into a switch and a throttle.

In our schema, a switch is like an s-throttle with the release wired to a constant; it does not synchronize but simply directs tokens either to a destination in the current frame or to the parent frame. A throttle is like an s-throttle with the predicate wired to the constant *true*; it delays tokens

[4] To fit loop-carried dependencies in $\mathcal{O}_1$ into the framework it is necessary to introduce another stage $\mathcal{O}_0$, which contains only dummy operations. Such dummy operations are removed in the actual implementation.

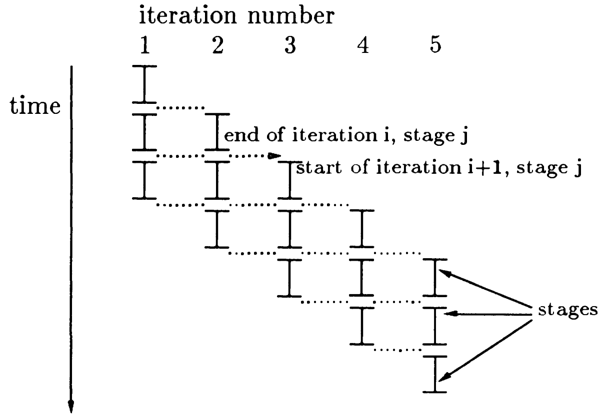


Figure 6: Synchronization in a Pipelined Loop (3 stages)

until the arrival of a release signal. Figure 8 shows a detailed view of the implementation of these components. Termination in this schema is somewhat complex: a frame is free when all values have passed the throttle and the switch, and when any other activity in the loop has completed. We indicate the output of a signal tree which detects these conditions simply as a termination signal emanating from the entire stage.

3.3 Asymptotic Completion Rate

How should we compare the performance of different stage decompositions of the same loop? Our primary concern is that the loop should execute as quickly as possible. In order to evaluate stage decompositions from this point of view, we must define an abstract notion of speed of execution. The measure we will use is the asymptotic rate at which a loop executing in pipelined manner can complete iterations. We take into account only the time taken for loop operations, and delays due to dependencies and pipeline synchronization; we ignore the cost of loop overhead. In order to measure this asymptotic rate, we define the *operation schedule* of a loop decomposition. This is the schedule of the loop operations as they exe-

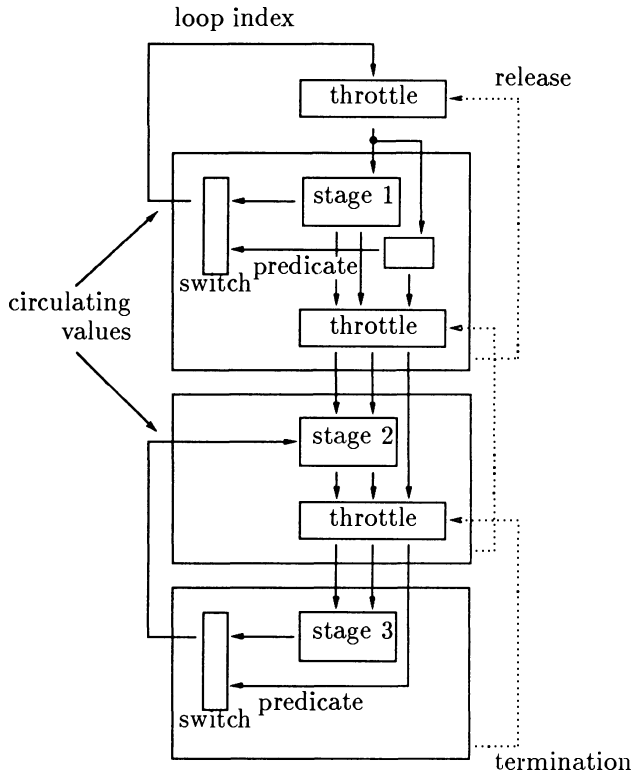


Figure 7: Schematic Pipelined Loop (3 stages)

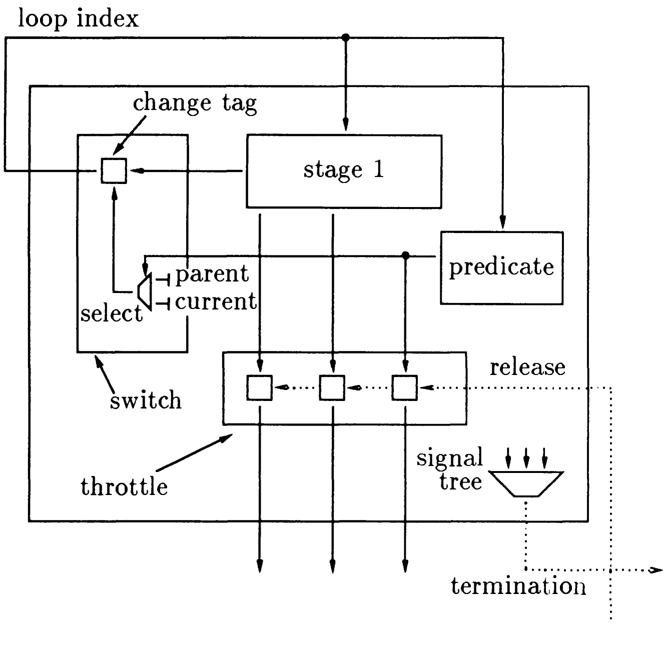


Figure 8: Detail of Pipelined Loop (first stage)

cute in one "beat" of the pipeline after a large number of iterations have started (*i.e.,*the pipeline is full). It is a greedy schedule of operations in which execution of an operation is constrained only by loop-independent dependencies which do not cross stages, and by loop-carried dependencies.

Definition 4 *Given a well-formed stage decomposition D of a dependency graph $G = \langle \mathcal{O}, E_I, E_C, l \rangle$, let $I(o), C(o) \subset \mathcal{O}$ be the sets of operations defined as follows:*

$$\begin{aligned} I(o) &= \{o' \in \mathcal{O} \mid \langle o', o \rangle \in E_I\} \\ C(o) &= \{o' \in \mathcal{O} \mid \langle o', o \rangle \in E_C\} \end{aligned}$$

Then the operation schedule *of D is a mapping $p : \mathcal{O} \rightarrow \mathcal{Z}^+$ defined as follows:*

$$p(o) = max(\{0\} \cup \{p(o') + l(o') \mid o' \in (I(o) \cap \mathcal{O}_i) \cup C(o)\})$$

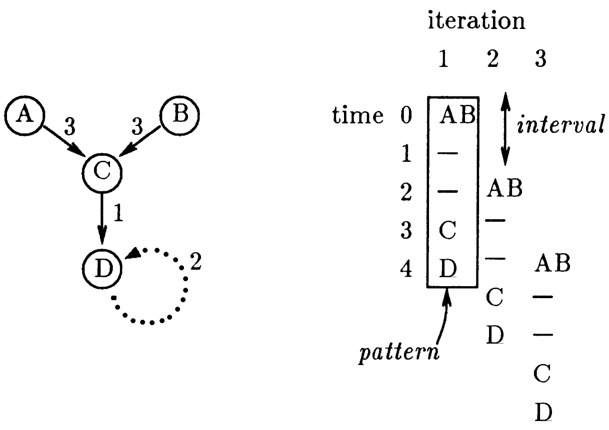
Since duration of a pipeline beat is the maximum duration of any stage, the asymptotic rate of completion of iterations is just the length of the operation schedule.

Definition 5 *Given a well-formed stage decomposition D of a set of loop operations $\mathcal{O}$, the* completion interval *of D is*

$$max\{p(o) + 1 \mid o \in O\}$$

where $p : \mathcal{O} \rightarrow \mathcal{Z}^+$ is the pipeline pattern of D.

If we ignore the cost of the synchronization required to implement our pipelined loop schema, then a stage decomposition with minimal completion interval will yield a loop with asymptotically optimal execution speed. By ignoring the cost of synchronization, we define a model in which an arbitrary ordering can be imposed on instructions without cost. This is exactly the abstract model used in scheduling very long instruction word (VLIW) architectures, and as we will now show, algorithms from that area are applicable.



instr	operations		
1	D	A	B
2	C	–	–

Figure 9: Inner Product VLIW Schedule and Code

4 Loop Scheduling Techniques

On conventional pipelined machines (like simple RISC machines and VLIW machines), some of the benefits of loop unraveling can be obtained through a compile-time transformation called *software pipelining* [Aik88,Lam87] [RYYT89]. Intuitively, the compiler works out how the loop would unravel if execution of operations was constrained only by data dependencies. Using this information, the compiler can then transform the body of the loop to overlap operations from successive iterations. The new loop can execute as fast on the pipelined machine as the original loop would on a machine with dynamic loop unraveling. We will use this technique to calculate a stage decomposition of the loop.

4.1 VLIW Scheduling

A VLIW schedule for a loop can be specified by a schedule of operations in each iteration and an initiation interval between the start of iterations [Aik88].

Definition 6 *A* VLIW schedule *of a dependency graph* $G = \langle \mathcal{O}, E_I, E_C, l \rangle$ *is a pair* $\langle t, \delta \rangle$ *where:*

1. $t : \mathcal{O} \rightarrow \mathcal{Z}^+$ *is a pattern which assigns each loop operation a time relative to the start of the iteration, and*

2. δ *is a time interval between the start of iterations.*

Definition 7 *A VLIW schedule* $\langle t, \delta \rangle$ *of a dependency graph* $G = \langle \mathcal{O}, E_I, E_C, l \rangle$ *is* consistent *iff*

1. $\forall \langle o, o' \rangle \in E_I \; [t(o') \geq t(o) + l(o)]$

2. $\forall \langle o, o' \rangle \in E_C \; [t(o') \geq t(o) + l(o) - \delta]$

The interpretation of such a schedule is that an operation $o \in \mathcal{O}$ in iteration i is scheduled at time $t(o) + (i-1)\delta$. Essentially, no operation is scheduled before its time. A VLIW schedule can be coverted into code by considering the inverse mapping $t^{-1} : \mathcal{Z}^+ \rightarrow \mathcal{P}(\mathcal{O})$ which assigns to each time step a (possibly empty) set of instructions. This mapping corresponds to a VLIW program for the body of the loop, as Figure 9 illustrates for our inner product example.

If resources (such as registers or the number of operations that can be scheduled at any time step) are bounded, the problem of computing a VLIW schedule with minimum iteration interval is known to be NP-complete [GD79]. The assumption of finite resources is relaxed in the case of abstract VLIW architectures — it is assumed that the processor has an unbounded number of registers and can dispatch any number of operations in each cycle. Let $G = \langle \mathcal{O}, E_I, E_C, l \rangle$ be a dependency graph of an inner loop without conditionals, $n = |\mathcal{O}|$ and $m = |E_I \cup E_C|$. Then, assuming unbounded resources, the consistent schedule of G with minimum iteration interval can be determined in time $O(nm \log n)$ by a longest path algorithm in the dependency graph [Lam87]. The schedule illustrated in Figure 9 is such a longest-path schedule.

A variation that has been studied by Aiken and Nicolau is to perform the scheduling not on the original loop body, but after unrolling that loop some number of times [Aik88]. This results in a better schedule in some cases, but the unfolding needed to achieve the optimum schedule can, in pathological cases, be exponential in the size of the dependency graph. The payoff from this additional work is quite limited — it is easy to show that the time required for n iterations of the loop is decreased by n/δ, or at most one time unit per iteration.

Since optimum loop scheduling with resource constraints is known to be NP-complete, adaptations of non-bounded scheduling algorithms are used. One simple approach is to calculate a non-bounded schedule and then, taking the operations in some heuristically-chosen order, move them back in the schedule to the first available spot which does not violate resource constraints. This approach will approximate the unbounded schedule when resource constraints are not the primary scheduling consideration, but may produce a very bad schedule when resources are tight.

The other extreme is to schedule for resources first, and then worry about dependencies. The Doacross method [Cyt86] is a moderated form of this approach. First, uniprocessor code is developed for a single loop iteration. This allows effective scheduling of the resources of each processor, as in register allocation. While loop independent dependencies must be respected in such uniprocessor code, loop carried dependencies are ignored. The smallest initiation interval at which the uniprocessor code can be scheduled is then calculated. This interval may be considerably larger than the minimum possible.

A more robust approach is to combine the resource constraints with other dependencies in constructing the schedule. This avoids the construction of an overall schedule which is far from the resource constraints, and which must then be corrected with an overall heuristic. If heuristic decisions are performed early, then later scheduling operations can take account of them. Both Lam [Lam87] and Ebcioglu and Nicolau [EN89] have modified the basic code motion step of their respective scheduling algorithms to respect resource constraints. These researchers report that their heuristics often achieve optimal results, however no analytical characterizations of their behavior have been reported.

4.2 VLIW Scheduling and Loop Bounding

A VLIW schedule provides information which can be applied to many different kinds of scheduling. For instance, the value of k necessary for a k-bounded loop to execute at the speed of the VLIW schedule can be calculated. Given a VLIW schedule $\langle t, \delta \rangle$, let the *length* of the schedule l be defined to be the maximum difference $|t(o) - t(o')| + 1$ between the times of two operations in the same iteration. Then a k-bounded loop for $k = \lceil l/\delta \rceil$ will execute at the speed of the VLIW schedule.

The significance of this result depends on the quality of the VLIW schedule and the nature of the timing assumptions which went into the abstract model. A schedule which is optimal in the sense of Aiken and Nicolau will result in a bounded loop which completes iterations at a rate which asymptotically approaches the speed of an unbounded loop. If worst-case timing assumptions can be made, then this result will always hold. If *expected* timing assumptions are made (as described in Section 6) then the quality of the resulting schedule is less clear, and such approximations should be validated through experimentation.

4.3 VLIW Scheduling and Loop Pipelining

A stage decomposition can be derived from a VLIW schedule by choosing a *stage depth*, and "slicing" the schedule into stages as follows. Given VLIW schedule $\langle t, \delta \rangle$ and a stage depth d, we let

$$\mathcal{O}_i^{t,d} = \{ o \in \mathcal{O} \mid (i-1)d \leq t(o) < id \}$$

While other methods can be found to derive a stage decomposition from a VLIW schedule, this simple slicing approach will meet our optimality criterion.

The central observation which links VLIW scheduling to stage decompositions is this: given a schedule $\langle t, \delta \rangle$, then for $d \leq \delta$ the completion interval (as defined in Section 3), of the stage decomposition $D^{t,d} = \{\mathcal{O}_i^{t,d}\}$ is exactly δ. This implies that if $\langle t, \delta \rangle$ is a VLIW schedule which is *optimal* in the sense that it has minimum iteration interval δ, then $D^{t,d}$ will have minimum completion interval. The heuristic scheduling algorithms which generate an approximately optimal schedule will yield a decomposition with pipeline interval close to the minimum.

Once we have calculated a schedule for our loop operations, the question arises of the best choice of stage depth d. The answer is that choosing $d = \delta$ is best; we now examine why. If $d < \delta$ then the stage decomposition $D^{t,d}$ will have more stages than $D^{t,\delta}$, and so the critical path through the loop may pass through more throttles. Since the completion interval of $D^{t,d}$ is δ, the resulting dataflow graph will complete iterations at a slower rate. The use of frame memory is minimized by a very small choice of d, but this is not our main concern here. If $d > \delta$, then the resulting stage decomposition may not have minimum completion interval. Thus the natural choice is $d = \delta$.

5 Long Latency Operations

In Section 3 we introduced a pipelined loop schema for the Monsoon Explicit Token Store dataflow architecture, which is derived from the k-bounded loop schema for Monsoon [Pap88]. That schema allocates k identical memory frames, and assigns them to loop iterations in round-robin fashion. Values which circulate between iterations, or "nextified variables" in Id terminology, are passed between frames.

Our pipelined schema instead allocates a single memory frame, and constrains execution of k different loop iterations to disjoint *stages* of this frame. Thus values will flow in two directions: between stages of a single iteration and between iterations. In this schema, all communication between iterations must be between adjacent stages: values are passed from stage i of iteration j to either stage $i+1$ of iteration j or to stage i of iteration $j+1$. Any communication pattern can be built out of these communication paths.

The schema as presented so far assumes that every operation will complete in the same iteration as it is initiated in. This means that a value generated by an operation in stage i will be consumed by an operation in stage i or else in the first time step of stage $i+1$. However it is not uncommon for a load from memory to take a long time, and the VLIW model of execution assumes that if multiple operations of long duration are initiated, their results will be returned in the order in which they were initiated. In dataflow models, the order of completion of operations is arbitrary. When an operation returns its value to the stage which initiated it, correct ordering is guaranteed by that fact that in our schema there is only a single executing instance of each stage.

An operation which returns a result to a later stage has no such guarantee. Without some special mechanism, such a loop runs the risk of initiating a second operation before the result of the first has been consumed. A dataflow graph which can place two tokens on a single arc in this way is not meaningful. What is required is a "FIFO n-buffer" which can buffer a fixed number n of results and deliver them to a destination in the order in which the generating instructions were executed.

In our inner product example, this issue arises because the result of the LOAD operations is not available at the beginning of the second stage. Thus, the value cannot pass though the throttle at the start of the sec-

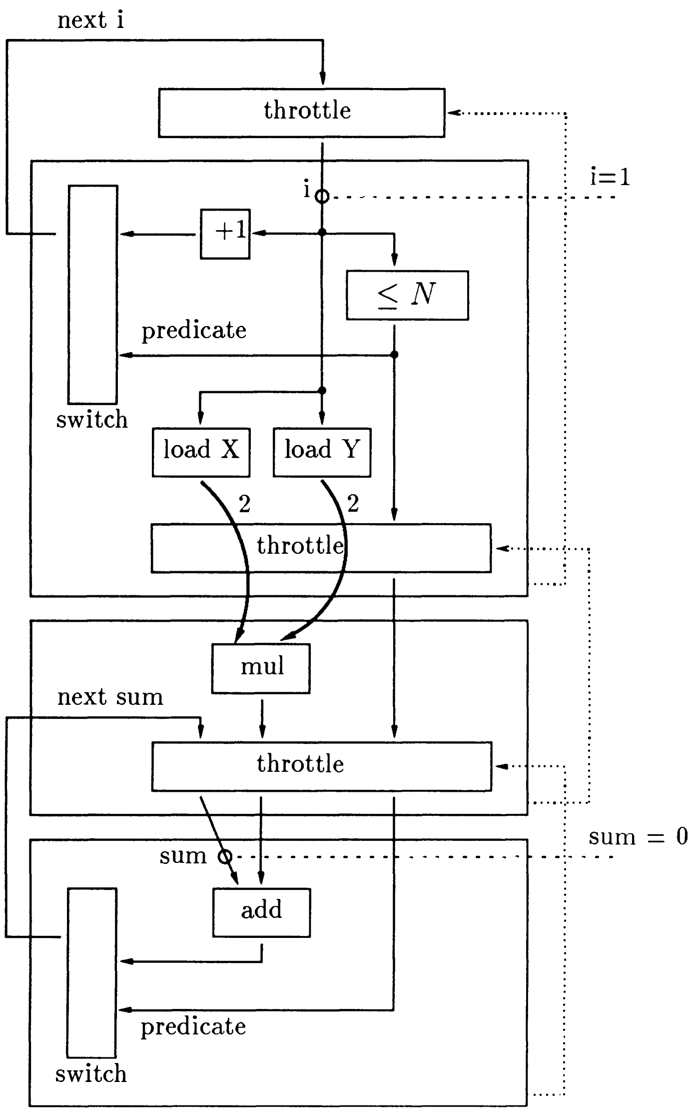


Figure 10: Pipelined Inner Product

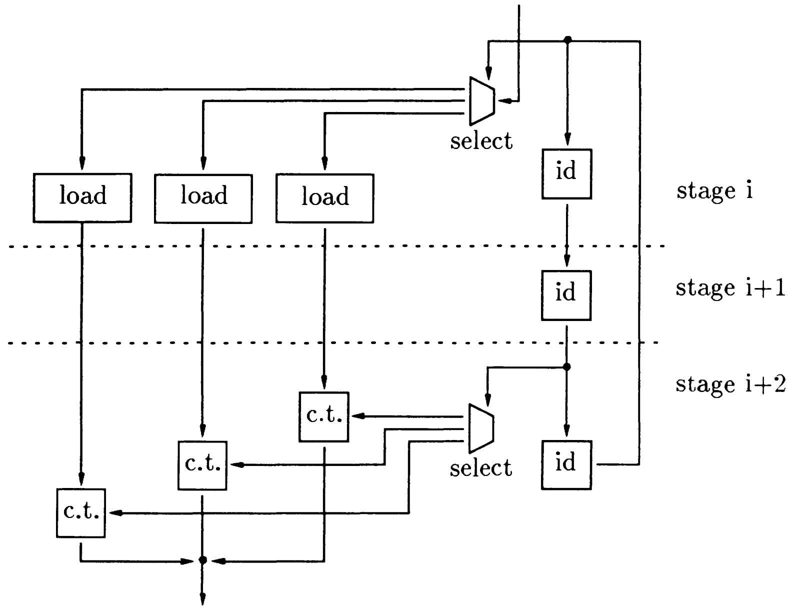


Figure 11: Single Frame FIFO 3-Buffer Implementation

ond stage, but must be delivered directly to the multiply operation. In Figure 10, which illustrates the pipelined schema for this example, we represent the need for a FIFO 2-buffer between the LOAD operations and the multiply by a heavy arc annotated with the number 2. We now discuss the implementation of these buffers.

If an operation returns a value $n-1$ stages after it is initiated, then it is possible that there can be n outstanding operations at any time. A buffer cannot be simply implemented in frame memory as one would in imperative memory for technical reasons relating to the mechanism which regulates instruction dispatching. The destination of a token is actually an instruction, not a location in memory. A token which must be stored in frame memory to await its match is sent to a location specified in the instruction. This mapping of instructions to locations in frame memory is fixed.

Thus, rather than address into our buffer with n offsets into frame memory, we use the addresses of copies of the LOAD instruction which will write into the buffer, as shown in Figure 11. Each copy sends its output to a different CHANGE TAG operation; thus the location in frame memory corresponding to that CHANGE TAG operation implements a location in the buffer. A particular iteration will write into a buffer location with a given LOAD will also extract the result by enabling the corresponding CHANGE TAG operation.

In order to coordinate between the LOAD and the CHANGE TAG, special tokens are circulated with indices into the buffer, and a SELECT operator choses which pair of LOAD/CHANGE TAG operations to enable. n special tokens are circulated among the n stages between the initiation of the operation and the use of its value, each bearing a value specifying one these n buffer locations. For ease of illustration these tokens are depicted as circulating through a ring of identity operators as they pass through the iteration stages; these identity operators need not be implemented. There may be better a implementation of FIFO buffers on Monsoon; we have simply demonstrated one possibility.

The multiplication of frame storage in this scheme is analogous to the duplication of contexts in the k-bounded schema. However the extra synchronization which we introduce allows us to allocate extra space only where it is needed to increase parallelism.

6 Discussion

Our pipelined loop schema is based on the premise that a useful schedule can be constructed to predict the execution of a loop on a dynamic dataflow architecture. Such a schedule is based on assumptions about the maximum duration of operations; when such assumptions are violated, speed of execution may suffer. We now present one approach to deriving such timing assumptions, and argue that this approach is reasonable; the ultimate test lies in experimentation.

6.1 Timing Assumptions for Dataflow

How can the software pipelining algorithms discussed in Section 4 be applied in the dataflow context? The scheduling algorithms we discussed relied on the assumption that the latency of *all* operations is known to the compiler. In a dataflow architecture, even ALU operations of fixed duration will suffer unpredictable latencies because of the dynamic nature of instruction scheduling. We will start with a simplified timing model which does not take such variability into account, and will then argue how our methods can be applied to a realistic model.

In an explicit token store dataflow processor like Monsoon, the basic ALU operation execution consists of the following steps:

1. A fetch from instruction memory
2. Matching operands and issuing an instruction
3. An ALU execution cycle
4. Generating result tokens

Each of these steps has known duration, and so the time for execution can be determinted. We denote this "basic cycle time" as T, and will calculate timings in units of T. Memory writes will take one cycle, and while the extra latency needed for a read from memory is unpredictable, but we will assume an expected value of M cycles.

We will calculate schedules based on the assumption that all operations except memory reads take one cycle. and that all memory reads take a fixed time of $1 + M$ cycles. Our scheduling assumptions model an "empty" dataflow processor in which all resources are devoted to the loop being scheduled. As is well-known, dataflow processors schedule many independent tasks on the same processor to cover memory latency. It is difficult to factor this into our schedule in any precise way, but the effective latency of memory operations is decreased by this technique. From our loop schedule, we can calculate two statistics: the length l and the critical interval δ of the loop. The length is the minimum time taken between the execution of the first and last instructions of any iteration. The critical interval is the minimum iteration interval at which the loop can be scheduled. If our loop is executed in an environment where memory loads appear to take less time than was assumed, then the length of the loop may decrease, the critical interval may decrease, or both may decrease.

If the length of the loop decreases, then the overall time for the computation will remain the same. The early completion of some portions of the schedule will simply make machine resources available to other computations; operations on the critical path are not delayed. If the critical interval decreases, then synchronization can cause future iterations to wait, and the overall computation may be slowed. If both decrease, then the question is which decreases *more*, and the outcome depends on the structure of the loop.

In the case of the loop length being decreased, the computation will still proceed at the fastest rate possible given the available resources. In the case of a decrease in the length of the critical interval, an increase in resources allocated to the loop could speed up the computation. For instance, allocating multiple frames to each stage of the pipeline would allow a later iteration to proceed in a given stage before an earlier one had finished. However, a congested environment is exactly when resources must be conserved, so such remedies are not likely to be helpful.

6.2 Comparing Loop Bounding to Pipelining

Since we motivated our pipelining schema by addressing the shortcomings of loop bounding, it is instructive to compare the two approaches. Both schemas restrict the resources used by a dataflow loop by introducing dependencies between iterations. Loop bounding introduces such dependencies on the basis of iterations, while pipelining uses a decomposition of iterations into stages. Both schemas are parameterized: loop bounding by the number of active iterations k, and pipelining by the choice of stage decomposition. A good choice of parameter can be derived for either schema by the use of static scheduling.

Ignoring the overhead of synchronization, the resource requirements of a k-bounded loop are k times the requirements of a single iteration. Ignoring also the complicating factor of long latency operations such as loads, a pipelined loop requires the same resources as a single iteration ($k = 1$). Our scheme for dealing with long latency operations, which we presented in Section 5, causes an increase in frame memory similar to loop bounding, *but only for those long operations.* Finally, ignoring the effect of variable instruction latencies, the pipelined loop will complete instructions at the fastest rate possible ($k = \infty$).

If instruction latencies were completely predictable in dataflow processors, and optimal schedules could always be calculated, then pipelining would be a strict improvement on loop bounding. However, we allowed ourselves the assumption of known latencies only as an aid to static analysis. If latencies are completely unpredictable, loop bounding can show better performance that pipelining. The looser synchronization inherent in loop bounding quite literally provides a buffer against variable latencies.

One refinement of the basic scheme that gets around this problem is to execute k iterations of the pipelined loop concurrently, using the scheme shown in Figure 12. A pipelined loop can occupy a single memory frame, and uses approximately as much memory as the unpipelined version. If one were willing to allocate more memory frames to the loop, one could allocate k pipelined frames, and spread the execution of iterations among them as in a bounded loop, as shown schematically in Figure 12 The round robin scheduling of loop bounding would make the pipelines quite sparse: $k - 1$ our of every k stages would be empty. This would provide the same sort of buffer against variable latencies as is provided by the bounded loop schema.

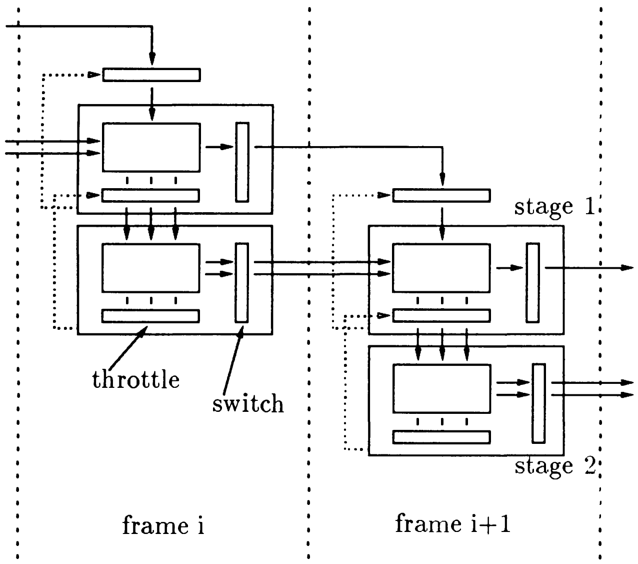


Figure 12: A Bounded and Pipelined Loop

7 Conclusions

Effective compile time management of resources requires the ability to ensure that independent uses of any resource are disjoint. Models of execution such as dataflow and VLIW take advantage of instruction level parallelism, and can also manage resources at that level. We have used compiling techniques for VLIW machines to generate code for dynamic dataflow machines. As with any such discussion of the effect of a fine-grained dynamic mechanism on program execution, the ultimate test is in experimentation. We unfortunately have not yet been able implement our schema on an actual dataflow processor. However, we look forward to such an opportunity to test the validity of our assumptions.

Ackowledgements

Thanks to Jamie Hicks for taking the time to explain the Monsoon loop bounding schema. Thanks also to Anne Rogers and Richard Johnson for their comments on earlier drafts.

E-Mail Addresses

The authors can be contacted at the following electronic mail addresses:

Micah Beck	`beck@cs.cornell.edu`
Alex Nicolau	`nicolau@ics.uci.edu`
Keshav Pingali	`pingali@cs.cornell.edu`

References

[Aik88] Alex Aiken. *Compaction-Based Parallelization.* PhD thesis, Cornell University, 1988.

[CA88] D. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988.

[Cyt86] R. Cytron. Doacross: Beyond vectorization for multiprocessors. Technical report, IBM T.J. Watson Research Center, 1986.

[EN89] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *Proceedings of the ACM SIGARCH International Conference on Supercomputing*, 1989.

[GD79] M.R. Garey and D.S.Johnson. *Computers and Intractability: A guide to the theory of NP-completeness.* Freeman & Company, 1979.

[Lam87] M. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, 1987.

[Pap88] G. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Massachusetts Institute of Technology, 1988.

[Rug87] C.A. Ruggiero. *Throttle Mechanisms for the Manchester Dataflow Machine.* PhD thesis, Manchester University, 1987.

[RYYT89] B. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 departmental supercomputer. *Computer*, 22(1):12–35, January 1989.