Loop Pipelining for High-Throughput Stream Computation Using Self-Timed Rings

Gennette Gill, John Hansen and Montek Singh
Dept. of Computer Science
Univ. of North Carolina, Chapel Hill, NC 27599, USA
{gillg,jbhansen,montek}@cs.unc.edu

ABSTRACT

We present a technique for increasing the throughput of stream processing architectures by removing the bottlenecks caused by loop structures. We implement loops as self-timed pipelined rings that can operate on *multiple data sets* concurrently. Our contribution includes a transformation algorithm which takes as input a high-level program and gives as output the structure of an optimized pipeline ring. Our technique handles nested loops and is further enhanced by loop unrolling. Simulations run on benchmark examples show a 1.3 to 4.9x speedup without unrolling and a 2.6 to 9.7x speedup with twofold loop unrolling.

1. INTRODUCTION

This paper targets the domain of high-performance digital ICs that are implemented using pipelined dataflow architectures. We focus on *stream processing* architectures, *i.e.,* those that take a stream of data items and produce a stream of processed results. High-speed stream processors are a natural match for many high-end applications, including 3D graphics rendering, image and video processing, digital filters and DSPs, cryptography, and networking processors. The development of fast stream processors is likely to be key to sustaining the explosive growth we are witnessing in consumer electronics, multimedia applications, and high-speed networking.

While stream processors are well-suited to implementing algorithms that are dataflow in character, a key challenge is the efficient implementation of control constructs. In particular, the presence of conditionals ("if-then-else" constructs) and loops ("for" and "while" constructs) in algorithms typically creates performance bottlenecks that limit the throughput of the resulting stream processor even if the remainder of the algorithm is efficiently pipelined. While there has been some recent work on addressing this challenge for conditionals [4, 7], there is no satisfactory approach for efficiently handling loop constructs. This paper, therefore, focuses on algorithmic loops, and provides an efficient approach for their high-throughput implementation.

Existing approaches to implementing loop structures are limited in the throughputs they can achieve. They focus primarily on limited concurrency improvements: (i) shaving off delays from the critical path of an iteration, *e.g.* by local transformations that change sequential operations into parallel ones, and (ii) slight overlapping of adjacent iterations, *i.e.* overlapping the end of one iteration with the start of the next iteration of the *same* data item, or overlapping the end of an item's last iteration slightly with the next item's first iteration. These approaches can somewhat shorten the execution time of each data item's computation. However, none of these approaches truly allow multiple data items to be processed concurrently; only a single data item is allowed to be processed by the loop hardware at any time. Thus, in existing approaches, even if the loop is pipelined at the circuit level, it is effectively *unpipelined at the algorithm level.*

This paper presents an approach for efficiently implementing iterative loops in hardware, which truly achieves loop pipelining at the algorithmic level. Unlike existing approaches that only target local concurrency improvements, our approach focuses instead on allowing *multiple successive data items to be computed concurrently,* thereby offering more substantial throughput benefits. Our approach is applicable to iterative algorithms in which successive data items can be computed independent of each other, but where successive iterations on the same item are allowed to be dependent on each other. This class of algorithms is quite rich and can be used in several real-world applications including networking, encryption, multimedia applications, and differential-equation solvers.

Our contribution is twofold: (i) *novel architectural building blocks* for implementing iterative architectures, and (ii) an *automated synthesis approach* that allows high-level iterative algorithms to be mapped onto these building blocks.

The core architectural building block—a self-timed ring structure—directly implements iterative computations in an efficient manner. We introduce a novel ring interface composed of several "helper" blocks that allow the ring to operate at its ideal throughput, *i.e.* neither under-utilized nor congested. Our approach takes advantage of one of the key benefits of self-timed architectures: modular design. The architectural building blocks can be easily composed together because they all have the same flexible external interface, which is robust to differences in internal implementations and variations in internal delays.

Our synthesis approach takes as input a high-level specification of an iterative algorithm and generates the structure of its loop-pipelined implementation. Our synthesis approach also handles conditionals, sequential blocks, and parallel blocks, and is therefore applicable to a rich class of algorithms. The modularity of our building blocks allows our synthesis approach to easily handle *nested loops,* with arbitrary level of nesting. Finally, our approach is further enhanced by incorporating *loop unrolling* at the hardware level, *i.e.* replicating loop hardware to further exploit data parallelism.

Our approach has been validated by applying it to a set of complex examples with real-world applications, including iterative polynomial root-finding, ordinary differential-equation solving, greatest-common-divisor computation, a simple encryption algorithm, etc.

ICCAD '06, November 5-9, 2006, San Jose, CA

Each example was synthesized using a mature commercial asynchronous synthesis flow—Haste/TiDE (formerly Tangram) tool suite from Handshake Solutions [1], a Philips subsidiary—both with and without our loop pipelining approach. The results of applying our approach are quite encouraging: 1.3–4.9x increase in throughput without loop unrolling. When a twofold loop unrolling was applied, even higher speedups were achieved: 2.6–9.7x.

The remainder of this paper is organized as follows. Section 2 presents background on self-timed ring architectures, and discusses previous work on loop optimization. Section 3 then introduces our new approach to loop pipelining; for simplicity of presentation, this initial discussion focuses only on loops with a fixed number of iterations. The basic approach is then extended in Section 4 to provide several advanced features: handling loops with variable iteration counts; handling nested loops; and exploiting loop unrolling.

2. PREVIOUS WORK AND BACKGROUND

2.1 Previous Work

Previous Approaches in Hardware Design. Several approaches in hardware design have attempted to optimize iterative computation. However, their main objective is to reduce the *latency* of the loop, as opposed to improving its throughput. One such approach is the CASH compiler by Budiu and Goldstein [4], which translates an ANSI C program into data-flow hardware. A different approach by Theobald and Nowick [9] targets generation of distributed asynchronous control from control-dataflow graphs, with the objective of optimizing communication between the controllers, and between a controller and its associated datapath object.

The above approaches offer very limited throughput benefit, since they are mainly targeted to improving a loop's latency. In particular, the optimizations introduced by these approaches focus on shaving off some delays from each loop iteration, and offer modest concurrency increases: the tail-end of an iteration for a given data item is allowed to overlap somewhat with the start of that same item's next iteration, or the tail-end of an item's last iteration overlaps somewhat the start of the next item's first iteration. Neither approach allows multiple distinct data items to be truly processed concurrently. Hence, the throughput benefits of these approaches are modest, and nowhere in the 2–10x range targeted by our proposed approach.

Like our approach, work by Kapasi, Dally et al. [7] targets efficient implementation of stream algorithms. However, it does not address the critical challenge of pipelining loops and only addresses the problems related to conditional branches.

Williams and Horowitz [10, 11] introduced the classic work on self-timed rings. An iterative floating-point divider chip was designed using a self-timed ring structure [11], which was later adapted for use in the earlier HAL processors. However, this design did not actually allow multiple data sets to be inside the ring concurrently. In addition, their approach does not target translation of high-level algorithms directly into hardware. A key contribution of [10] was analysis of the performance of self-timed rings which can be loaded with multiple data sets. This analysis is reviewed in Section 2.2.2.

Previous Approaches in Software Design. There are many loop optimization techniques for software compilers; we mention a only few relevant approaches here. In software pipelining [3] the goal is to re-structure a loop so as to reduce the impact of inter-instruction dependencies by mixing instructions from successive iterations. Parallelizing compilers often use some combination of loop unrolling and compaction to achieve parallelization between successive loop iterations [2]. There are other approaches used by parallelizing compilers: loop skewing, index-set splitting, and node splitting [8]. However, many of these approaches apply only to loops that iterate over arrays, with each iteration operating on a subsequent array element. As such, they have limited applicability to algorithms that require multiple iterations for each data item.

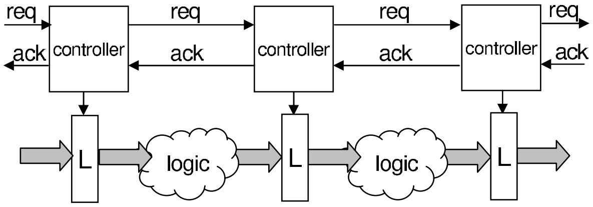

Figure 1: A simple self-timed pipeline

2.2 Background: Self-Timed Pipelines and Rings

This section briefly reviews *asynchronous* or *self-timed* pipelines and rings, including their structure, operation and performance.

2.2.1 *Asynchronous or Self-Timed Pipelines*

Structure. Figure 1 shows the basic structure of a self-timed pipeline. Each pipeline stage consists of a controller, a storage element ("data latch"), and processing logic. A key feature is the absence of a system-wide clock for global synchronization. Instead, synchronization is achieved locally through request/acknowledge handshaking between neighboring stages.

Operation. Data is transferred from one stage to next according to the handshake protocol chosen for the pipeline. Typically, a stage generates a *request* to initiate a handshake with its successor stage, indicating that new data is ready. If the successor stage is empty it accepts the data and performs two further actions: (i) it acknowledges its predecessor for the data received and, (ii) initiates a similar handshake with its own successor stage further down the pipeline.

Performance. There are three metrics typically used to characterize the performance of a self-timed pipeline: *forward latency*, *reverse latency*, and *cycle time.* The throughput of the ring as a whole can be derived from these three metrics. The forward latency is simply the time it takes one data item to flow through an initially empty pipeline. Thus, if the latency of Stage$_i$ is L_{Stage_i}, then the latency of the entire pipeline is:

$$L_{Pipeline} = \sum_i L_{Stage_i} \qquad (1)$$

Similarly, the reverse latency characterizes the speed at which empty stages or "holes" flow backward through an initially full pipeline. The reverse latency of the entire pipeline is simply the sum of the stage reverse latencies:

$$R_{Pipeline} = \sum_i R_{Stage_i} \qquad (2)$$

The cycle time of a stage, denoted by T_{Stage_i}, is the minimum time that must elapse after a data item is produced by that stage before the next data item will be generated. Since a complete cycle of a stage typically consists of transmitted a data item forward, followed by accepting a hole from the next stage, the following relationship holds:

$$T_{Stage_i} = L_{Stage_i} + R_{Stage_i} \qquad (3)$$

The maximum throughput a stage can support is simply the reciprocal of its cycle time. The cycle time of a linear pipeline is typically limited by the cycle time of its slowest stage:

$$T_{Pipeline} = \max_i \left(T_{Stage_i}\right) \qquad (4)$$

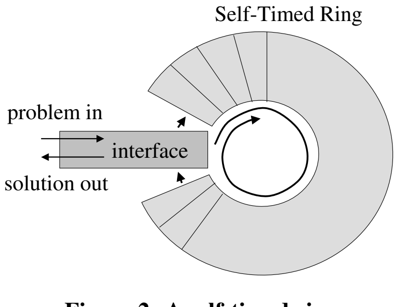

Figure 2: A self-timed ring

2.2.2 *Self-Timed Pipeline Rings*

Frequently, self-timed pipeline stages must be arranged in configurations other than simple linear pipelines to meet the requirements of an application. One configuration of special interest in this paper is a *self-timed ring.* The ring structure allows one data item to repeatedly pass through the same sequence of processing stages, thereby allowing iterative computations to be implemented.

Structure. Figure 2 shows the basic structure of a self-timed ring. The ring contains a number of stages which perform computation and an interface stage whose function is to load data items into the ring and to drain results from the ring.

Operation. Once an item is introduced into the ring through the interface stage, it revolves inside the ring until some terminating condition indicates that the computation is complete. The result is then drained from the ring through the interface stage.

Performance. The classic work by Williams [10] introduced a useful metric for measuring the performance of a ring: the number of times any data item crosses the interface stage per second. We will refer to this measure as the *ring frequency.*

The performance of the ring is highly dependent on its *occupancy, i.e.,* the number of data items revolving inside it. When the number of data items is small, the ring frequency is low, and the pipeline is said to be "data limited." On the other hand, when nearly every stage of the pipeline is filled with data items, the performance is once again limited because holes are needed to allow data items to flow through the pipeline; in this scenario, the pipeline is said to be congested, or "hole limited."

Data Limited Operation. If there are k data items in the ring, then in the time a particular data item completes one revolution around the ring (*i.e.,* $\sum_i L_{\mathit{Stage}_i}$), all k items would have crossed the interface stage. Therefore, the maximum ring frequency attainable is proportional to the ring occupancy:

$$\textit{Ring Frequency} \quad \leq \quad k \Big/ \sum_i L_{\mathit{Stage}_i} \tag{5}$$

Hole Limited Operation. If the ring is filled with data items in nearly all stages, then the ring frequency is limited by the number of holes in the ring. For one data item to cross the interface stage, exactly one hole must cross in the reverse direction. If there are h holes in the ring, then in the time a particular hole completes one revolution around the ring (*i.e.,* $\sum_i R_{\mathit{Stage}_i}$), all h holes would have crossed the interface stage, thereby allowing h data items to advance. The maximum ring frequency is therefore proportional to the number of holes. Thus, if N is the number of stages in the ring, then $h = N - k$, and we have the following bound on the performance:

$$\textit{Ring Frequency} \quad \leq \quad (N-k) \Big/ \sum_i R_{\mathit{Stage}_i} \tag{6}$$

Figure 3 shows a plot of the ring frequency versus its occupancy. The rising portion of the graph represents the data limited region, where performance rises linearly with the number of data items. The falling portion, similarly, represents the hole limited region, where performance drops linearly with a decrease in the number of holes.

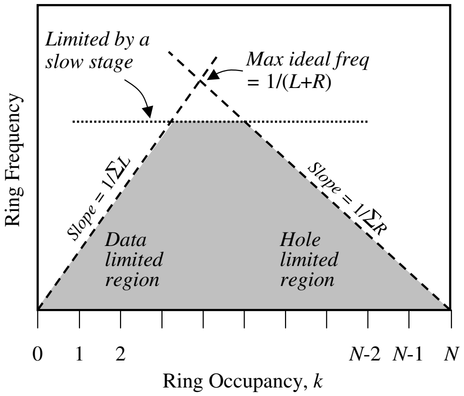

Figure 3: Upper bounds on the ring frequency

Limitations Due to a Slow Stage. If all the stages in the ring have similar forward and reverse latencies, then the maximum attainable performance will be the frequency at which the rising the falling lines of Figure 3 intersect. This point represents a frequency that is the inverse of the cycle time of each ring stage: $1/T = 1/(L+R)$. However, if some stages are slower than others, then the ring frequency will be limited by the cycle time of the slowest stage. In the figure, the horizontal line represents the maximum operating rate that can be sustained by the slowest stage in the ring [10]:

$$\textit{Ring Frequency} \quad \leq \quad 1 \Big/ \max_i (T_{\mathit{Stage}_i}) \tag{7}$$

The overall ring performance will always be constrained to lie under the canopy formed by the three lines in Figure 3.

3. BASIC APPROACH: PIPELINING LOOPS

We now introduce our basic approach for converting iterative loops into self-timed pipeline rings with the benefit of significantly higher throughput. In this section, we focus on a simple case: a **for** loop with a *constant iteration count, i.e.,* it iterates the same number of times on each data set. Our advanced approach is introduced in Section 4, which is able to handle a wider variety of specifications, including loops with variable iteration counts, nested loops, and unrolled loops.

We first motivate our work in Section 3.1 by showing how a **for** loop can be a bottleneck in the compute pipeline, thereby severely limiting throughput obtained. Then we introduce our approach for eliminating this bottleneck, including hardware templates for pipelining the loop (Section 3.2), an algorithm for mapping a high-level code fragment onto this template (Section 3.3), and an analysis of the performance benefit our of approach (Section 3.4).

3.1 Motivation: The Loop Bottleneck

We illustrate our method using the simple code example shown in Figure 4. This example represents a generic code fragment for a *streaming* hardware system that iterates on every data set using a **for** loop. In the example, the symbols $s1$ through $s8$ represent statements.

The code consists of a *main* procedure that performs communication with the environment and a *compute* function that does the computation. Once a data set is received from the environment by *main,* it is sent to *compute* for processing, and the result is then sent to the environment. These operations are performed repeatedly by *main,*.

```
func compute(in_context)
  s1; s2;
    for i = 1 to N
      s3; s4; s5; s6;
    end
  s7; s8;
  return(out_context)

proc main
  while (true) do
    read(input);
    output = compute(input);
    write(output);
  end
```

Figure 4: Sample code for a stream processor that iterates on every data set using a *for* loop

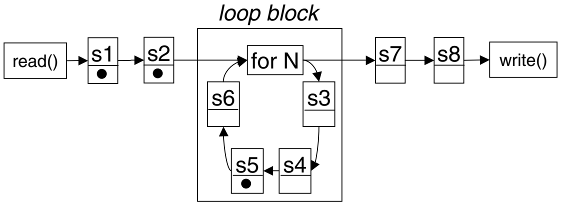

Figure 5: A simple implementation of the *compute* function

Rather than specifying variables in this code, we consider functions to have a *context* which is operated upon and modified by each statement. At the beginning of the *compute* function, the context consists of the entire input set sent to the function call, labelled *in_context*. This initial context is augmented with any locally defined variables before being sent into function. At the end of the function, the context holds the set of outputs to be returned to the calling environment.

The operation of the code fragment is as follows. The *main* procedure reads a data set from the environment and invokes *compute* function. The value passed from *main* forms the initial context (*i.e.,* set of inputs) for *compute.* Inside the body of *compute,* a certain number of statements—*e.g.,* $s1$ and $s2$—operate on the context. Next, the **for** loop operates on the context for N iterations. Then a certain number of statements, represented by $s7$ and $s8$, operate on the context after the *for* loop. Finally, the modified context is returned to *main*, which communicates the relevant portion of it to the receiving environment.

A direct translation of this code into data-driven hardware typically yields the schematic structure shown in Figure 5. Each statement in the code—$s1$ through $s8$—becomes a pipeline stage. Statements $s3$ through $s6$ along with the **for** statement compose the *loop block*. During operation, the data streams in from the environment through the *read()* stage and is streamed out to the environment through the *write()* stage.

A key observation is that the *loop block* has the same external interface as an individual pipeline stage, even though internally it contains an entire ring for iterative computation. Specifically, the *loop block* accepts one data set from the predecessor stage, performs a calculation, and passes the computed result on to a successor stage. It does *not* accept new data until the results of the calculation have been accepted by the successor stage.

Interestingly, the presence of a loop in the *compute* body has the same effect on the performance of the stream processor as *a single pipeline stage with long latency.* As discussed in Section 2.2.1, the throughput of a pipeline is determined by the cycle time of the slowest stage (Equation 4), which in turn directly depends on that stage's latency (Equation 3).

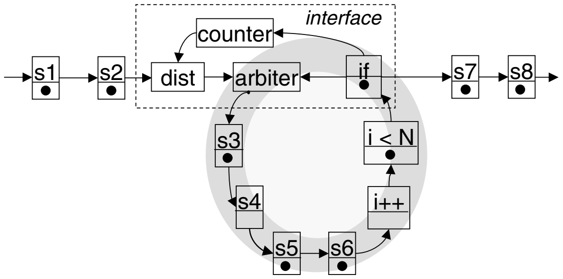

Figure 6: Proposed implementation of *compute*

As a result of this long effective latency and cycle time of the *loop block*, the throughput of the entire stream processor is severely limited. Even if each of the individual stages—*s1 through s8*—are implemented efficiently, the presence of a single long-latency *for* loop drastically diminishes the throughput obtained through pipelining. Figure 5 illustrates this bottleneck scenario by indicating the presence of data with a dot. Stages downstream from the *loop block* are idle while the stages upstream from the *loop block* are stalled.

Although some existing approaches can generate slightly more optimal implementations than that of Figure 5, they still suffer from the same bottleneck illustrated by this example, as discussed in Section 2.1. While these approaches [4, 9] somewhat increase concurrency, they only allow a very slight overlap between successive data sets.

3.2 Proposed Ring Structure

In order to overcome the bottleneck of loop computation, our approach introduces a novel structure based on a self-timed ring that significantly improves performance by operating on several data sets at once. However, there are three restrictions on the class of iterative streaming algorithms that can be translated using our method: (i) calculations on each distinct *data set* should not depend on each other, (ii) no communication with the caller environment should occur within the body of the loop, and (iii) different statements in the loop body should not share any resources. Note that our translation scheme still applies to algorithms in which consecutive iterations on the *same* data set do depend on each other.

Our proposed ring structure for the code in Figure 4 is illustrated in Figure 6. In our implementation, the ring as a whole does not have the same interface behavior as an individual pipeline stage: the ring can actually accept a new data set while the previous one is (or several previous ones are) still being operated on, space permitting.

The interface is composed of special-purpose "helper" stages—a *distributor,* a *counter,* an *arbiter,* and an **if** stage—which allow multiple data sets to enter the ring without interfering. Figure 6 illustrates the ability of the ring to operate on multiple data sets by indicating the presence of data with a dot. Note that each data set within the ring is in a different stage of computation.

Our pipelined ring should ideally be filled with as many data sets as possible without causing throughput degradation due to congestion. As discussed in Section 2.2.2, every self-timed ring has some ideal occupancy, C, at which it achieves maximum throughput. Therefore, our strategy is to allow at most C elements to be present inside the ring at any time. The ideal occupancy can be analytically computed if the forward and reverse latencies of stage inside the ring are known, as discussed in Section 2.2.2. If, however, these latencies are unknown, or highly data dependent, the ideal ring occupancy is determined by simulation.

The ring operates as follows. When a data set arrives from stage

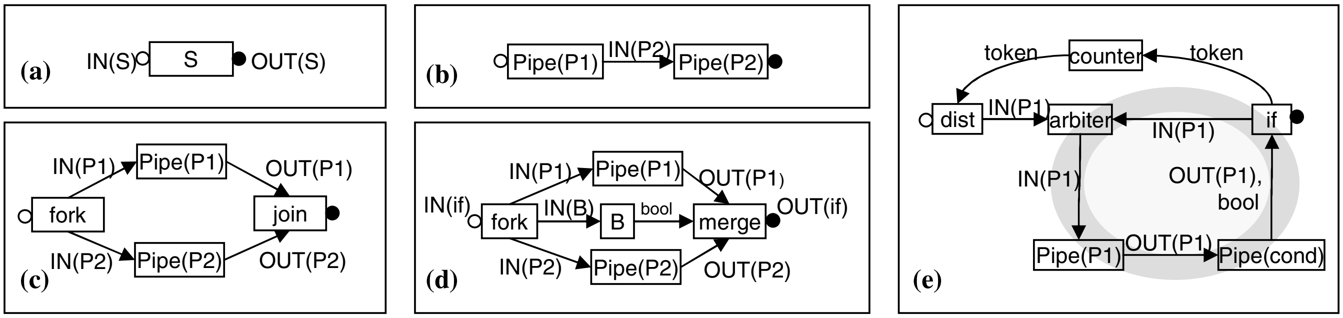

Figure 7: A graphical representation of each composition function a. Output of single_stage(S) b. Output of compose_sequential(P1, P2) c. Output of compose_parallel(P1, P2) d. Output of compose_cond(B, P1, P2) e. Output of compose_loop(P1, cond)

```
Pipe (P : program).
 begin
   if P is a single assignment statement S then
        output single_stage(S)
   else if P is the sequential block ‘‘P1; P2’’ then
       compose_sequential( Pipe(P1),  Pipe(P2) )
   else if P is the parallel block ‘‘P1 || P2’’ then
       compose_parallel( Pipe(P1), Pipe(P2) )
   else if P is the conditional ‘‘if(B) then P1 else P2’’ then
       compose_conditional( Pipe(B), Pipe(P1), Pipe(P2) )
   else if P is a loop ‘‘for (n) P1’’ or ‘‘while (cond) do P1’’ then
       compose_loop( Pipe(P1), Pipe(cond) )
 end
```

Figure 8: Our transformation algorithm

$s2$ as a new input to the loop, it is first sent to the *distributor* (labelled “dist” in the figure). If the ring is at less than ideal occupancy (*i.e* if *counter* $< C$), the distributor sends the data set into the ring and increments the counter to keep track of the ring occupancy. The data set then progresses through the stages of the loop body. At the end of an iteration, the data set passes through an **if** stage. If the terminating condition is not met, the **if** stage passes the data set back to the beginning of the ring. If the terminating condition is met, the **if** stage sends the data set out of the ring and also sends a signal to the decrement counter. If the ring had stopped accepting new data sets because it had reached ideal occupancy, this decrementing of the counter will allow a new data set to enter the ring.

One arbiter is necessary in this hardware scheme because data can enter the beginning of the loop from two places. New data arrives via the distributor, and current data loops back via the **if** stage. These two events can occur at arbitrary times, making arbitration necessary.

3.3 Transformation Algorithm

The algorithm for our hardware translation scheme is shown in Figure 8. The input to our algorithm is a high-level program; the output is a pipelined structural implementation of that program. Currently, our approach handles the following types of language constructs: sequential blocks, parallel blocks, conditionals, and loops.

Our algorithm assumes the ability to perform *live variable dataflow analysis,* in order to find IN and OUT sets of sections of code [8]. In particular, the IN set is the set of data values that are required to go into a code fragment either because they may be used inside, or because they may need to be relayed to a successor of that fragment. The OUT set of a code fragment is the union of the IN sets of its successors. These sets are used to determine which values need to be communicated between stages.

Figures 7(a-e) show a graphical representation of our pipeline composition functions. In these figures, each box represents a set of one or more pipeline stages. Each arrow represents a communication between pipeline stages. Labels on the arrows indicate the *context*, or set of variables, that must be passed between stages. (Note that the context is computed using the IN and OUT sets of sections of code.)

A stage or multi-stage block that is not yet connected to other stages or blocks indicates an input port with an open circle, ○, and an output port with closed a circle, ●. These ports will be connected to ports of other stages or blocks at the next upper level of hierarchy during the recursive traversal of our algorithm.

3.4 Performance Benefiand Overheads

Previous work on performance analysis of rings allows us to predict the speedup obtained by the use of our method. As discussed in Section 2.2.2 and shown in Figure 3, the *ring frequency* is proportional to the total number of data sets that are revolving inside the ring, as long as the ring is not congested (*i.e.,* “hole limited”). Therefore, the maximum speedup of our approach is proportional to the ideal ring occupancy.

The speed improvement of our method is largely due to improved hardware utilization. A ring that holds only one data set has a high amount of unused hardware at any given time. By allowing multiple data sets, we are able to obtain high hardware utilization from the components within the ring.

Our approach adds some overhead that decreases the actual speedup and increases total area. Certain “helper” stages—such as the distributor, counter and arbiter—increase the latency of the ring and add a small area overhead. Also, the counter has some latency and therefore will not allow new data to enter immediately after old data leaves. The most notable increase to area is the extra storage elements that are required in order to hold the entire context at each ring stage. This overhead is necessary to allow each data set to hold its own copy of the loop’s context, thereby enabling multiple data sets to coexist independently within the ring.

4. ADVANCED APPROACH

In Section 3 we described our basic scheme for translating a loop with a fixed number of iterations into a self-timed ring with maximal throughput. In this section, we describe three advanced techniques: (i) handling loops that have a variable (*e.g.,* data-dependent) number of iterations, (ii) handling nested loops, and (iii) using loop unrolling to further improve performance.

4.1 Data-Dependent Loops

Many algorithms contain loops that iterate a different number of times depending on the value of the input data set. Our basic approach of translating the loop into a self-timed ring can still be applied, but one problem arises. Specifically, if each data set is allowed to leave the loop as soon as its computation has finished, the data

```
func GCD( a, b )
  while( b != 0)
    s = a - b;
    if (s < 0)
       then swap(a,b)
       else a := s
  return C
```

Figure 9: Euclid's GCD solver

Table 1: Reordering in GCD computation.

inputs		original	re-ordered output	
a	b	output	value	tag
100	208	4	19	1
209	190	19	3	2
45	219	3	4	0
252	114	6	6	3
136	146	2	2	4
43	169	1	5	6
15	155	5	1	5
133	77	7	7	7

```
  func bisection ( coefs, tol, pos, neg )
1   while( abs (pos - neg) < tol)
2     mid := (pos + neg) / 2;
3     a := poly_eval( coefs , mid );
4     if a < 0
5       then neg = midpt;
6       else pos = midpt;
    return midpt;

 func poly_eval( coefs, x, degree )
   a = coefs[degree];
   for(i = degree-1; i ≥ 0; i--)
     temp = a * x;
     a = temp + coefs[i];
   return a
```

Figure 10: Code for bisection and polynomial evaluation

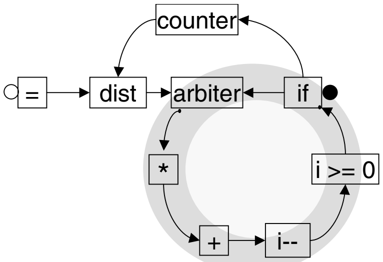

Figure 11: Structure for polynomial evaluation

sets will exit the loop in some order that may differ from the order in which they entered. Thus, the data sets exit the loop *out of order*.

One example of an algorithm that has a data-dependent iteration count is Euclid's algorithm for computing the greatest common divisor (GCD) of two integers. A pseudo-code implementation of this algorithm is shown in Figure 9. If the approach described in Section 3 were used to implement this code, the values in the output stream would be out of order. Table 1 shows the output in the original order and the output generated by the GCD ring assuming an occupancy of three.

Our solution to the reordering problem is to append a unique tag to each data set; the tag represents a sequence number. For example, the tags 0 through 7 are used in the GCD example in Table 1. This tag becomes a part of the context of the loop, ensuring that even if the results emerge from the loop out of order, they are still tagged correctly.

This tag can be used in two different ways: (i) the out-of-order results are simply passed on to the environment along with their tags, or (ii) a *re-order buffer* is used to correctly order the items before sending them to the environment.

The first method is preferable if the environment can handle tagged outputs, which can be useful in applications such as graphic renderers where outputs need not come out in order. The second method, which introduces a re-order buffer, is preferable if the calling environment must remain naïve to the re-ordering problem. There has been much research on the design of re-order buffers in the field of computer architecture [6]; our approach simply leverages that work.

4.2 Nested Loops

Nested loops are implicitly handled by the recursive pipelining algorithm shown in Figure 8. We provide an example here of a nested loop transformation to illustrate this capability.

An example of an algorithm that has nested loops is the *bisection* algorithm for finding a zero of a polynomial. Figure 10 shows the pseudo-code for this algorithm. The body of function *bisection* contains a **while** loop which successively halves the search interval for finding a zero of the polynomial. Each iteration of this loop requires the value of the polynomial to be evaluated at the mid-point of this interval. This evaluation is carried out by calling the function *poly_eval* which, in turn, is an iterative algorithm based on Cramer's rule, and contains a **for** loop with data-dependent iteration count.

When our algorithm is applied to the bisection code in Figure 10, it begins by calling the function *compose_loop* on the entire program. (For convenience the individual statements in the code will be referred to by their numbered labels.) The function *compose_loop* in turn triggers a call to *Pipe* to pipeline the body of the loop. After handling line 2, the algorithm calls *compose_sequential(Pipe(3), Pipe([4-6]))*. The output of *Pipe(3)* is a ring structure that implements the polynomial loop, as shown in Figure 11. The *poly_eval* ring is finally composed sequentially with the rest of the statements within the *bisection* body to form the structure shown in Figure 12.

4.3 Loop Unrolling

Unrolling the loop body to form a ring with a greater number of stages can greatly improve performance when combined with our hardware translation approach. Intuitively, this improvement results from the duplication of hardware inside the loop and a corresponding increase in the ring occupancy. Thus, the unrolled loop is able to perform more "work" per unit time.

More formally, the ring frequency (at ideal occupancy, *cf.* Equation 7) remains fairly unchanged when the loop is unrolled, but every "tick" at the loop's interface now represents a greater amount of work completed. In particular, if the loop is unrolled u times, every time a data set crosses the interface stage, it indicates that u iterations have just been completed on that data set, rather than just one. Therefore, ignoring overheads, the loop's effective computation throughput increases by factor equal to the number of times it is unrolled, u.

As a second-order effect, loop unrolling actually also has the benefit of somewhat reducing the overhead of the special-purpose "helper" stages: the distributor, the counter and the arbiter. That overhead is now amortized over a larger ring. As a result, the latency of each data set will tend to somewhat decrease and hardware utilization will slightly increase. One possible negative effect of loop unrolling is that it can cause some data sets to be iterated over more times than necessary, thereby requiring extra checks within the unrolled loop to preserve the semantics of the computation.

Although loop unrolling is a common technique in both software

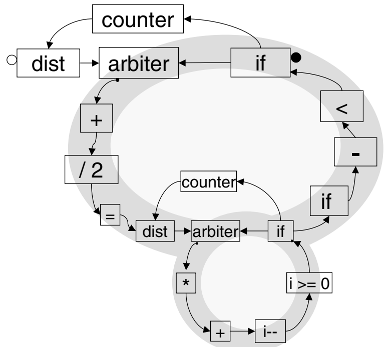

Figure 12: Bisection loop with nested polynomial evaluation

and hardware optimization, our current use of it has much different goals and performance effects. In software compilers, the primary benefit of loop unrolling is to introduce more room to allow instructions to be reordered, with the purpose of reducing stalls due to branch and data hazards. In hardware translation approaches, such as [4, 9], loop unrolling is used in conjunction with compaction to increase concurrency within the loop body. However, these approaches do not allow an increase in the occupancy of the loop, thereby obtaining limited throughput improvement. In contrast, our approach increases the loop occupancy by the same factor as the number of times it is unrolled, thereby obtaining dramatically higher speedup.

5. RESULTS

Benchmarks. Our approach targets iterative algorithms that operate on a stream of independent data sets. Section 4 discussed several such examples: GCD, BISECT, and POLY. In addition to those examples, we tested implementations of the following algorithms as benchmarks:

- BTREE: The BTREE algorithm searches a binary tree residing in ROM for a given input key. A key match returns the data value associated with the node that matches the key. Behavior is highly data-dependent: the number of loop iterations depends on how deep the input key's node is in the tree.

- CRC: The cyclic redundancy check (CRC) algorithm calculates an 8-bit checksum for a given block of input data. This algorithm has a fixed iteration count (16 iterations).

- ODE: The ODE example is the ordinary differential equation solver from [12, 13, 9], based on the Euler method. It receives as input the coefficients of a third degree ordinary differential equation, along with an interval over which to integrate, initial conditions, and a step size. Its output is the final value of the dependent variable. The number of loop iterations depends on the size of the interval and the step size.

- TEA: Tiny Encryption Algorithm (TEA) encrypts a 64-bit block with a 128-bit key. The number of loop iterations depends on the number of encryption rounds chosen. For this example, the number of rounds was fixed at 32 (16 loop iterations).

Experimental Setup. The benchmark examples were synthesized using the Haste/TiDE synthesis flow from Handshake Solutions [1], a Philips subsidiary. Haste (formerly "Tangram") is the only mature automated asynchronous synthesis flow available at present. While the control-dominated architectures that Haste currently produces are not an ideal match for pipelined dataflow applications, this experimental setup allows the relative performance benefit of our approach to be accurately estimated.

For each of the benchmark examples, three different implementations were synthesized: (i) a baseline version ("Original") that did not use our approach; (ii) a second version ("Pipelined") that used our loop pipelining approach, but did not use loop unrolling; and (iii) a final version ("Unrolled") that employed a twofold unrolling for its loop along with our loop pipelining approach. Simulation and area estimation tools from the Haste suite were used to quantify the latency, throughput and area of the resulting implementations. Since the performance of some of the examples is data-dependent, input streams containing as many as 100 data sets were used, and latency and throughput results were averaged over them.

Results. Tables 2–3 summarize the results of our experiments. Table 2 presents the area, latency and throughput obtained for each of the benchmarks. The final column presents the normalized throughput (relative to the "Original" version), to illustrate the performance benefit of our approach. Finally, the area and latency overheads of our approach are summarized in Table 3, once again normalized with respect to the "Original" version.

Discussion. The results demonstrate that a substantial impact on throughput is achieved by loop pipelining: up to 9.7x speedup. Without the use of loop unrolling, our approach obtains a throughput improvement by a factor of 1.3 to 4.9. When a twofold loop unrolling was applied along with loop pipelining, the speedup obtained was even higher: a factor of 2.6 to 9.7x.

In greater detail, algorithms that were pipelined into a relatively small number of stages (CRC and GCD) had a limited potential for loop pipelining because the maximum capacity of the self-timed ring structures in these cases was low. As a result, the throughput benefit was around 1.3x (without unrolling). On the other hand, algorithms that were highly pipelined (BISECT, ODE, and TEA), had larger ring structures which could accommodate a greater number of data sets concurrently. For these benchmarks, the throughput increase was substantially higher: a factor of 2 to 4.9x (without unrolling).

As expected, the twofold unrolling led to a throughput increase in each case by a factor of 1.7 to 2.0x, yielding an overall combined throughput benefit of 2.6 to 9.7x relative to the original implementation.

Although our approach results in a significant boost in throughput, there are costs associated with the performance improvement. Shown in Table 3 are the increases in total area consumed, and in the average latency per data set.

In terms of area, the pipelined version adds the overheads of loop control discussed in Section 3.4. Each pipeline stage must latch data from the previous stage, so algorithms with many stages or large contexts (*e.g.* BISECT) will see a large increase in area. By unrolling the loop twofold, the total area of the implementation increased by a factor of 1.3–1.9x.

The average latency for a data set also increased when loop pipelining was used. This was expected because, like most traditional pipelining approaches, our approach increases throughput at the expense of latency. The latency overhead in most of the benchmarks was in the 1.4–3.6x range, except for the example that contained a nested loop, BISECT. For BISECT, the latency overhead reported is substantially higher (8.2x) due to the compounded overheads of its nested loops.

While the area and latency overheads may seem daunting, for most applications the main performance measure is overall execution time. By increasing latency and chip area, a dramatic improvement in throughput results, reducing execution time by a large factor.

6. CONCLUSIONS AND ONGOING WORK

In this paper we presented a technique that allows loop hardware to operate on many data sets at the same time. We proposed a novel loop interface and a set of transformations from high-level code to

Table 2: Synthesis Results: Performance Benefit

Algorithm/	Area	Latency	Throughput	Normalized
Approach	(μm^2)	(ns)	(Mega items/s)	Throughput
BISECT				
Original	28928	1946	0.51	1
Pipelined	98420	15960	1.03	2.0
Unrolled	184400	16000	1.76	3.4
BTREE				
Original	2900	40	24.65	1
Pipelined	7335	110	41.91	1.7
Unrolled	10840	75	79.62	3.2
CRC				
Original	4405	66	14.99	1
Pipelined	10730	193	19.79	1.3
Unrolled	15080	137	40.21	2.7
GCD				
Original	1770	108	9.11	1
Pipelined	4998	390	12.24	1.3
Unrolled	6574	277	23.51	2.6
ODE				
Original	8931	571	1.75	1
Pipelined	15610	1338	3.61	2.1
Unrolled	25630	1156	7.07	4.1
POLY				
Original	23661	367	2.71	1
Pipelined	53880	1300	5.19	1.9
Unrolled	99280	1226	8.80	3.2
TEA				
Original	30390	1205	0.83	1
Pipelined	96720	2529	4.04	4.9
Unrolled	166500	1704	8.07	9.7

Table 3: Area and Latency (Relative Overheads)

Algorithm	Pipelined		Unrolled	
	Area (Norm.)	Latency (Norm.)	Area (Norm.)	Latency (Norm.)
BISECT	3.4	8.2	6.4	8.2
BTREE	2.5	2.7	3.7	1.9
CRC	2.4	2.9	3.4	2.1
GCD	2.8	3.6	3.7	2.6
ODE	1.8	2.3	2.9	2.0
POLY	2.3	3.3	4.2	3.3
TEA	3.2	2.1	5.5	1.4

a pipelined loop structure. Our results showed significant throughput improvement over the non-pipelined ring implementation, with further benefits through unrolling.

More work needs to be done to decrease the area overhead of our technique. Our current approach is conservative: each stage in the ring structure stores a distinct copy of the maximal set of live variables that it may need. We are exploring more optimal techniques for reducing this overhead.

In ongoing work, we are re-implementing the above benchmarks in a true dataflow style, in order to eliminate the unnecessary control overheads introduced by the Haste synthesis flow. We anticipate significantly better performance and area results after this modification. In addition, a loop-pipelined GCD design has been implemented in silicon. Initial results indicate fully functional parts, though a complete evaluation is pending.

A full-featured transformation algorithm is the next logical step in this work. Additional features include support for resource sharing, unrestricted communication with the environment, and handling dependencies across data sets. Other loop optimizing techniques, such as compaction, are also being pursued.

7. REFERENCES

[1] Handshake Solutions, a Philips subsidiary. http://www.handshakesolutions.com/.

[2] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *European Symposium on Programming*, pages 221–235, 1988.

[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[4] M. Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[5] A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Dept. of Computer Science, University of Utah, Sept. 1997.

[6] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[7] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In *Proc. of IEEE/ACM Intl. Symp. on Microarchitecture*, 2000.

[8] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[9] M. Theobald and S. M. Nowick. Transformations for the synthesis and optimization of asynchronous distributed control. In *Proc. ACM/IEEE Design Automation Conf.*, June 2001.

[10] T. E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

[11] T. E. Williams and M. A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, Nov. 1991.

[12] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 140–153. IEEE Computer Society Press, Apr. 1997.

[13] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Trans. on VLSI Systems*, 6(4):643–655, Dec. 1998.