THROUGHPUT OF LONG SELF-TIMED PIPELINES

Mark R. Greenstreet
Kenneth Steiglitz

CS-TR-190-88

November 1988

Throughput of Long Self-Timed Pipelines†

Mark R. Greenstreet and *Kenneth Steiglitz*
Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

We explore the practical limits on throughput imposed by timing in a long, self-timed, circulating pipeline (ring).

We first consider the case when computation, communication, and storage are combined in a single operation, and the time for this operation is random with an exponential distribution. This pipeline is amenable to queuing theory analysis, and we show that the asymptotic processor utilization is independent of the length of the pipeline, but is at most 25%.

This suggests a design where computation and communication are separated from storage. We analyze this pipeline with various distributions of processing time, and show that linear speedup can again be achieved, but in this case with utilization approaching 100%.

1. Introduction

Many problems of signal processing and general computation are amenable to solution by a one-dimensional pipeline. Such an architecture has the property that, in principle, the pipeline can be made arbitrarily long, with a proportionate increase in throughput on the same sized problem, and with no increase either in the complexity of the processor or the communication bandwidth. A prototype machine based on this idea, for the lattice-gas model of Frisch, Hasslacher, and Pomeau [1], was recently reported in [2]. The resultant machine is simply a concatenation of identical, full-custom VLSI chips, and the system has *linear speedup* — n such chips have n times the throughput of one.

To achieve massive parallelism using this architecture, one would like to build very long pipelines. One important practical limit to this extensibility is determined by the need to time such a distributed system, and this paper explores that limit for a self-timed, circulating pipeline (ring).

In Section 2, we consider clocked designs based on long chains of clock buffers. We examine possible implementations of these buffers, and show that many typical designs are not suitable for arbitrarily long pipelines. This motivates considering self-timed designs.

Sections 3 and 4 provide background material on self-timed designs [3]. In Section 3, we show how common methods of self-timed signaling can all be considered to be based on the same Boolean abstraction, and in Section 4, we use this abstraction to design a ring-oscillator which captures the timing behavior of these self-timed designs

†This work was supported in part by NSF Grant MIP-8705454, U. S. Army Research Office - Durham Contract DAAG29-85-K-0191, and DARPA Contract N00014-82-K-0549.

independent of their function.

In Section 5 we consider the throughput of a circular, self-timed pipeline under certain simplifying assumptions. We assume that computation, communication, and storage are all coalesced into a single operation, and that the time for this operation is random with an exponential distribution. We use queuing theory models to show that linear speedup can be realized in such self-timed pipelines. However, the asymptotic utilization in this simple model is only 25%.

In Section 6, we separate the computation and communication from the storage, and we consider more detailed models of the computation time. In particular, we examine the three cases where processing times are fixed, bounded, and exponentially distributed with an offset. In each of these cases, under suitable conditions, we show that linear speedup is still obtained, and with utilization approaching 100%.

2. Clock Buffering for Pipelines

In [4], Fisher and Kung present several clock distribution designs which (implicitly) use chains of buffers, as illustrated in Fig. 1. They require that the skew introduced by each buffer is bounded. Under this assumption, they claim that arbitrarily long pipelines can be constructed with a clock period that is (asymptotically) independent of the size of the array. The difficulty with their argument is that they neglect variations in clock skew during operation of the circuit — for example, between leading and falling edge, or from pulse to pulse.

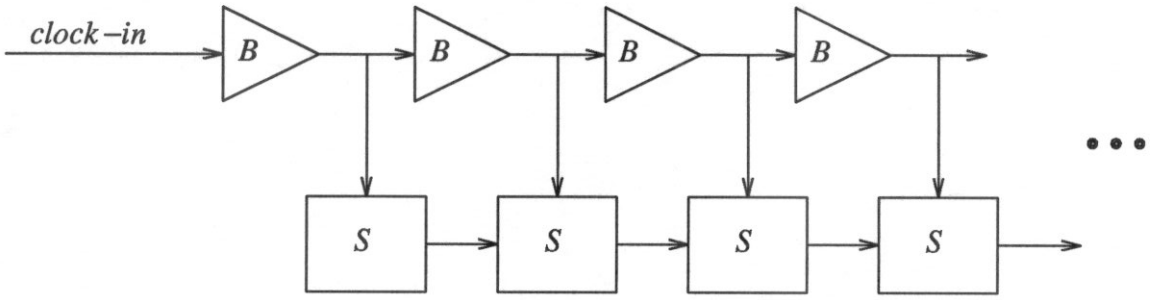

Fig. 1. A linear array of processing stages, clocked by a linear array of buffered clock signals, after [4]. The buffers are labeled "*B*", and the processing stages "*S*".

Consider differences in delay between leading and falling edges in the buffers of Fig. 1. Let t_{lh} be the delay for a rising edge, t_{hl} for a falling edge, and $t_d = \max(t_{lh}) - \min(t_{hl})$. Without loss of generality, assume $t_d > 0$ (otherwise, the following argument applies with h and l exchanged). It is possible, in the worst case, that each stage delays rising edges by $\max(t_{lh})$ and falling edges by $\min(t_{hl})$. The high portion of each clock pulse output is then t_d shorter than the input pulse (or non-existent). Given enough stages, the clock pulse completely disappears.

To prevent pulses from disappearing, a one-shot could be added to the output of each buffer. When the input of the one-shot is high and the output is low, the one-shot generates a high output pulse that has a duration of at least w_h and does not fall before the input. Likewise, the one-shot guarantees that the length of the low output pulse is at

least w_l. To show that this does not produce a reliable buffer chain, we construct a counterexample in two steps. The first step exploits variations of delays through the one-shots to produce a string of minimum-width clock pulses. The second step exploits variations of width requirements to force a clock pulse to be missed by a buffer.

For simplicity, we assume $w_h = w_l = w$. Each buffer delays pulses (from input to output) by some amount t. Due to variations as the circuit operates, t and w are bounded random variables with ranges $\Delta t = \max(t) - \min(t)$ and $\Delta w = \max(w) - \min(w)$. We assume the input clock is symmetric with high and low intervals of τ (where $\tau \geq w$).

If the first stage delays the first (e.g. rising) edge by $\max(t)$ and all subsequent edges by $\min(t)$, then the first pulse has a width of $\tau - \Delta t$ at the input to the second stage. If the first k stages delay the first edge by $\max(t)$ and all subsequent edges by $\min(t)$, then the first two edges are separated by $max(\tau - k\Delta t, w)$ at the output of Stage k. Narrowing of the separation to less than w is prevented by the one-shot. Let

$$k = \lceil (\tau - w)/\Delta t \rceil$$

If subsequent stages continue to delay the first edge relative to the others, the interval between the second and third edges will be reduced. For any positive integer m, at the output of Stage mk, the first m edges will be separated by the minimum amount, w.

The second step of the construction exploits variations in w. Let

$$m = \lceil \max(w)/\Delta w \rceil$$

Assume the first mk stages impose a minimum width of $\min(w)$ and Stage $mk+1$ imposes a minimum width of $\max(w)$. According to the first step of the construction, each of the first km stages delays the first clock edge by $\max(t)$ and all following edges by $\min(t)$. At the output of Stage km, the first m edges occur at intervals of $\min(w)$. The output of Stage $m \cdot k+1$ can only toggle every $\max(w)$ time units. By comparing the time to input m edges to Stage $mk+1$ (i.e. $m \cdot \min(w)$), and the time to output them from Stage $mk+1$ (i.e. $m \cdot \max(w)$), we see that we must have

$$m \cdot \min(w) + \max(w) < m \cdot \max(w)$$

which contradicts the way we chose m, and shows that Stage k+1 must miss a pulse.

These arguments show that two typical implementations of buffers using digital components cannot be used for arbitrarily long chains. Analogous arguments apply to their analog equivalents. If the digital buffers are replaced by a chain of phase-locked loops, the local phase drift and varying top frequencies will result in a failure scenario like that of the one-shot based implementation. If the digital buffers are replaced by simple analog buffers, noise is inevitably added to the signal at each stage of the buffer chain. Given enough stages, the clock signal is completely masked by noise.

We conclude that typical methods of clock buffering do not guarantee that a clock pulse can be propagated reliably through an arbitrarily long chain of buffers.

Accumulation of local variations can ultimately produce violations of the clocking requirements. We conjecture that a reliable clocking mechanism for arbitrarily long pipelines must use feedback. This can be accomplished using self-timed signaling protocols, which we study in the remainder of this paper.

3. Abstraction of Self-timed Signaling

In the simplest version of a pipeline, each stage repeatedly performs a cycle of accepting new data from the previous stage, computing a new result, and transmitting this new result to the next stage. Before beginning a new cycle, a processor must wait until its predecessor has produced a new result (if it isn't available already). In self-timed designs, this is accomplished by providing *completion information* along with the data. This allows a processor to determine when its predecessor has completed a cycle and produced a new result. Likewise, before completing a cycle, a processor must wait until its successor has input its previous result. This is accomplished with an *acknowledge signal.*

From now on, the term *control signal* will be used to refer to the completion or acknowledge information, to distinguish it from the *data signals*. Control signals can be provided either by adding a wire (separate from the data) which the sender uses to indicate its state, or by encoding completion on the same wires as the data. We call these *separate control* and *encoded control*, respectively.

As an example, consider an eight-bit bus. To transmit completion using separate control, a total of nine-wires are used. Eight wires are used to transmit the data, and the ninth is set low when data is changing and high when new data is available. To transmit a single bit using encoded completion, two wires can be used. The pair (L, H) denotes a false value, (H, L) denotes a true value, and (L, L) denotes a new computation in progress. Using this encoding, eight-bits can be transmitted with sixteen wires. More compact encodings can be used; however, separate completion requires fewer total wires in general than encoded completion. On the other hand, separate completion cannot be used if the skew between wires is large; encoded completion is required for complete insensitivity to delay.

Acknowledging can be performed with either *four-cycle* or *two-cycle* signaling [3]. With four-cycle signaling:

1. The sender transmits a new value and signals completion.
2. The receiver accepts the new value and raises the acknowledge line.
3. The sender drops the completion signal.
4. The receiver drops the acknowledge signal.

With two-cycle signaling:

1. The sender transmits a new value and toggles the completion signal.
2. The receiver accepts the new value and toggles the acknowledge line.

Four-cycle signaling is typically easier to implement, but two-cycle signaling can achieve greater data-bandwidth over the same wires.

Either form of signaling can be implemented using either form of completion. For example, using two-cycle signaling with encoded completion, true and false can be encoded (H, L) and (L, H) on even numbered cycles, and (H, H) and (L, L) on odd

numbered cycles.

The two methods of signaling completion and the two methods for acknowledging provide four general possibilities for self-timed communication. All four versions share certain essential properties. First, the control signal alternates between two possible states. These two states can be either "flag up" and "flag down" (four-cycle), or the toggling of the control signal (two-cycle).

For purposes of timing analysis, the control signal can be considered as a single Boolean value, the *control wire*. For the separate completion methods, the control wire is actually a separate physical wire, but for encoded completion it is an abstraction of the combined data and control wires. This abstraction removes the details of the data transmission and leaves only simple Boolean control values. With this view, the circular pipeline becomes a ring-oscillator, and it is such an oscillator that we examine in the next section. The results derived for the ring-oscillator apply directly to the operation of the circular pipeline.

4. The C-element Protocol

In a pipeline, data must be stored at each stage. For simplicity, we consider the case where each stage can store one value, and for the sake of discussion, we consider this storage to be associated with the *output* of each stage. When Stage $i-1$ completes a computation, its output must remain stable until Stage i has completed its computation using that value. When Stage i sends a completion signal to Stage $i+1$ (indicating that *output*(i) can be used), the same signal can serve as an acknowledgment to Stage $i-1$ (indicating that *output*($i-1$) has been used). Therefore the (control) state of each stage can be specified by a single Boolean value, which we will call the *state of that stage.*

For four-cycle signaling the state bit is T (true) to indicate that a new output value is available. Clearly, Stage i cannot begin a new computation before Stage $i-1$ outputs a new value (is in state T). This suggests the (partial) transition rule:

old outputs of stages			new output
i-1	i	i+1	i
T	F	-	T

As described above, Stage i must hold its output value until Stage $i+1$ acknowledges it by changing to state T. This leads to the (partial) transition rule:

old outputs of stages			new output
i-1	i	i+1	i
-	T	T	F

These transition rules are necessary but not sufficient to specify what happens under all conditions. We derive sufficient conditions by observing that the same Boolean abstraction applies to both 2- and 4-cycle signaling. Since the rules for 2-cycle signaling must be symmetric with respect to T and F, the partial rule for a transition to T must be

conjuncted with the complement of the partial rule for the transition to F, and vice-versa. This gives the complete set of transition rules:

old outputs of stages			new output
i-1	i	i+1	i
T	F	F	T
F	T	T	F

This corresponds to the ring-oscillator constructed with Muller C-elements shown in Fig. 2.

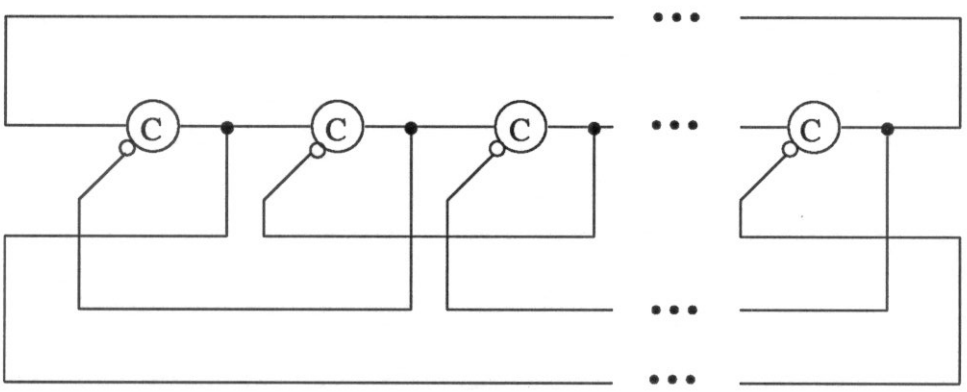

Fig 2. Ring oscillator constructed with C-elements.

This ring-oscillator can support waves of different modes. The simplest case is a single-mode wave. For such waves, the stages in the ring can be divided into two groups of contiguous processors: those whose outputs are T, and those whose outputs are F. A possible sequence generated by a ring of eight stages is shown below:

T	**T**	**T**	**T**	F	F	F	F
T	**T**	**T**	**T**	**T**	F	F	F
F	**T**	**T**	**T**	**T**	F	F	F
F	F	**T**	**T**	**T**	F	F	F
F	F	F	**T**	**T**	F	F	F
F	F	F	**T**	**T**	**T**	F	F
F	F	F	F	**T**	**T**	F	F
F	F	F	F	F	**T**	F	F
F	F	F	F	F	**T**	**T**	F
F	F	F	F	F	**T**	**T**	**T**

The transition rules prevent the crest (T outputs) of the wave from ever overtaking the

trough (F outputs), or the trough from overtaking the crest. The single-mode wave is therefore stable.

A processor is *active* if it can change its state. In 2-cycle signaling each change of state corresponds to the completion of a computation in the usual sense. In 4-cycle signaling, new computations are completed only every other state change. By the transition rules, a processor is active if and only if it is in the same state as its successor and the opposite state as its predecessor.

For the single-mode wave, there are one or two stages active at any time. Thus, the utilization of processors in a ring with a single-mode wave decreases as the inverse of the number of processors in the ring. The solution to this problem is to use higher mode waves. In a k-mode wave, there are k groups of contiguous processors whose outputs are T and k groups with F outputs. Thus, there can be up to $2k$ active stages, and the utilization of processors can be much higher. As in the single-mode case, the transition rules prevent a crest from overtaking a trough or a trough from overtaking a crest. Again, each of the higher mode waves is stable.

The C-element oscillator functions properly independent of delays in the logic elements or wires. Between any two changes of the output of a stage, both inputs to the stage must change. This ensures that each stage "waits" for its predecessor and successor stages to perform their computations before changing its output. Furthermore, this waiting also compensates for delays in the wires or inverters. If any component is unusually slow, all other components will wait for it, and the oscillator functions properly. Likewise, an unusually fast component will be forced to wait for the completions of its neighbors. Formal proofs of the delay independence of the oscillator are given in [5,6].

5. Linear Speedup with Exponentially Distributed Processing Times

In the Boolean abstraction, C-elements in the ring oscillator correspond to computational elements of the actual pipeline. Here, we model the time for computation with random delays in the C-elements. This allows us to determine the performance of self-timed pipelines. We use techniques of queuing theory to analyze the case of exponentially distributed processing times. This forms the basis for understanding other situations.

We assume that the time for each stage to complete a transition is exponentially distributed with mean τ. Furthermore, we assume that the times for different stages or different transitions made by the same stage are independent random variables. In this framework, the ring-oscillator can be described by simple Markov chains.

First, consider an oscillator with n stages and a single-mode wave. The crest of the wave can start at any of the n stages and the crest can have lengths from 1 to $n-1$, so there are a total of $n(n-1)$ possible states for the ring. By rotational symmetry, there are $n-1$ equivalence classes for these states corresponding to the $n-1$ possible lengths of the crest. We call these $S_1, \ldots, S_{n-1}$. For states in S_1 and S_{n-1}, one stage is active; in the other states, two stages are active. We can therefore compute the utilization of the processors by determining the distribution of the S's.

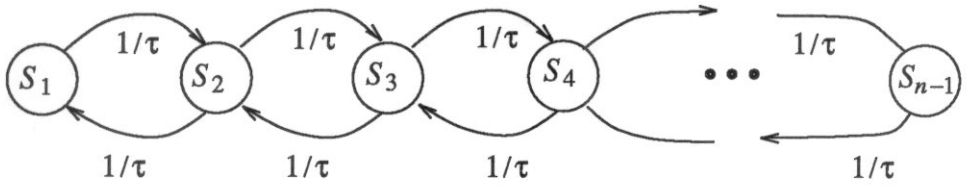

Fig. 3. State-transition rate diagram for the single-mode case. The arc labels $1/\tau$ indicate the rates of transition.

Figure 3 shows the state-transition rate diagram [7] for the oscillator with a single-mode wave. It is easy to show by induction that each of the classes is equally probable; the ring is in state S_i with probability $1/(n-1)$ for all $1 \le i \le n-1$. The probability that only one stage is active is

$$P\{\text{ one stage active }\} = P(S_1) + P(S_{n-1}) = \frac{2}{n-1}$$

Since either one or two stages are active,

$$E[\#\text{ active stages }] = P\{\text{ one stage active }\} + 2P\{\text{ two stages active }\} = \frac{2(n-2)}{n-1}$$

and the utilization for a ring with a single-mode wave is therefore

$$U_1 = \frac{E[\#\text{ active stages }]}{\text{total \# of stages}} = \frac{2(n-2)}{n(n-1)}$$

(See Fig. 4.) These predictions were compared with the results of a Monte-Carlo simulation with 10,000 computations per processor, and the two agreed within 1%. Better utilization can be achieved with higher modes, and we next consider that case.

For higher mode waves, the number of possible states grows exponentially with n in the worst case, and the simple analysis presented above becomes intractable. However, asymptotic results can be derived. Let n be the number of stages in a ring, and let k be the mode of the wave in the ring. The *average wavelength* is $w=n/k$. By fixing w, meaningful asymptotic results can be obtained in the limit as n and k go to infinity.

A *segment* refers to a sequence of adjacent processors in the same state — a crest or trough of the wave. Only the last stage in a segment can be active, and it is active if and only if the length of the segment is greater than one. Thus, the utilization of processors can be determined once the distribution of segment lengths is known.

By the symmetry of the ring, the segment lengths are identically distributed. We assume that segment lengths are also independently distributed in the limit that n goes to infinity. This conjecture is supported by simulation results.

Each segment can be viewed as a queue. A transition which lengthens a segment corresponds to an arrival to the corresponding queue; a transition which shortens a segment corresponds to a departure; a segment of length one corresponds to an empty queue

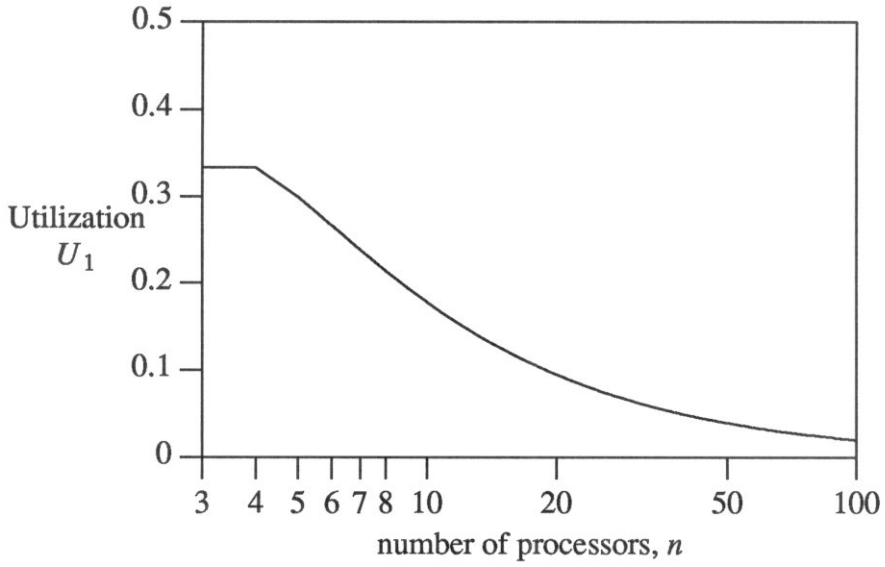

Fig. 4. Utilization U_1 vs. number of processors n.

(no departure possible). A segment can grow if the successor segment has a length greater than one, and the time for the transition which lengthens the segment is exponentially distributed with mean τ. Thus, the arrival rate is $(1-p_1)/\tau$, where p_1 is the probability that the length of a segment is one. The departure rate is $1/\tau$: whenever a segment includes more than two stages, the last stage is active. Because this is an M/M/1 queuing system [7], the expected length of a queue is $(1-p_1)/p_1$, and the expected length of a segment is $1/p_1$. By definition, the expected length of a segment is $w/2$. Thus, $p_1 = 2/w$. This yields an expected utilization of

$$U_\infty(w) = \frac{2(w-2)}{w^2}$$

From this result it follows that choosing $w = 4$ maximizes the value of U_∞, and that this maximum value is $U_\infty(4) = 1/4$. Thus, assuming that segment lengths are independent for large n, we have shown:

Linear Speedup Property Given independent exponentially distributed processing times, the asymptotic utilization for the self-timed pipeline is independent of its length.

6. Efficient Pipelines

In the previous section, we showed that self-timed pipelines can be realized with linear speedup. However, the asymptotic utilization of processors with these pipelines is only 25%. This low utilization has two causes: first, our arrangement of the pipeline prevents two adjacent processors from being simultaneously active; and second, the large variance of the exponentially distributed processing times causes waiting. In this section, we begin by separating the computation from the storage elements, thereby allowing

overlapping computation by adjacent processors. We then show that, under various assumptions about the distribution of processing times, utilizations arbitrarily close to 100% can be achieved.

6.1 Separating Computation and Storage

To achieve high utilizations, overlapping computations by adjacent processors are essential. To take an example from a practical design, in [8], a self-timed, circular pipeline with three stages for division is presented where overlapping execution is realized by exploiting known relative timings of different operations. In particular, four-cycle signaling with encoded completion is used in the divider, and computations can be either "evaluation" (i.e. computing a new quotient bit and partial remainder) or "precharging" (i.e., returning to the "no-data" state). Precharging in Stage 2 can be performed faster than evaluation in Stage 1, and this allows these two operations to overlap. However, the authors of [8] note that the subsequent evaluation in Stage 2 cannot begin as soon as the new data from Stage 1 is available. The start of the evaluation in Stage 2 must be delayed to guarantee that it does not start before the precharge in Stage 3 starts. This delay is caused by the fact that the critical timing path combines computation, storage, and control.

In this section, we will present a new pipeline arrangement which realizes the following design objectives:

1. overlapping computations can be performed in adjacent processors,
2. computations begin as soon as new data is available,
3. correct operation is guaranteed without any timing assumptions.

This design exploits the delay insensitivity of the C-protocol. In particular, arbitrary delays may be added to the path from one stage of the pipeline to the next. Each computation alters data values while preserving completion status. Since only the completion information is relevant in the Boolean abstraction, each computation is, for the purposes of timing analysis, the identity function: i.e., a delay. This delay can be "factored-out" of the C-protocol. As shown in Fig. 5, the computation is performed by the processor f which transports data from the output of one C-element to the input of the next. Each C-element stores values output by the preceding processor. A processor can start computing as soon as new input data is available from the C-element in the preceding stage. The result is stored in the C-element of its own stage after the successor stage has used the previous result. This allows processors in adjacent stages to be active simultaneously. Since the C-protocol is delay insensitive, this pipeline is correct and delay insensitive.

To achieve high utilization, most processors must be simultaneously computing new results. For a processor to be active, its output must be in the opposite state of its input. This in turn requires that the outputs of the C-elements before and after the processor be in opposite states. If most processors are active computing new results, there will be long sequences of adjacent C-elements in alternating states. Thus, a pipeline with high processor utilization will have an average wavelength very close to two.

For the C-element of Stage i to transfer its input to its output, it must be in the same state as the C-element of Stage $i+1$. Therefore, the processor of Stage $i+1$ cannot be active when the C-element of i changes its output. We call these inactive stages *bubbles*. When C-element i transfers its input to its output, it enters the same state as C-element

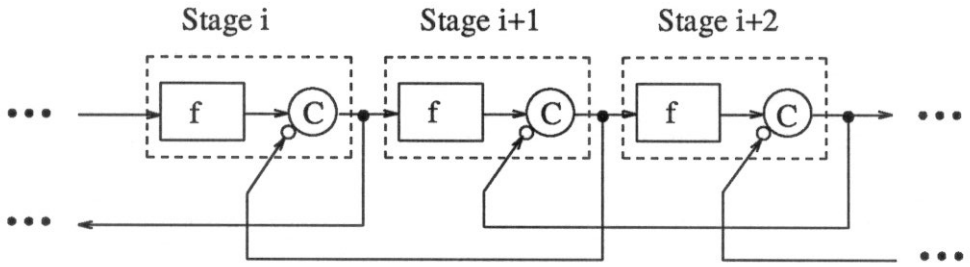

Fig. 5. Modified pipeline: the computations f and the C-elements are separated.

$i-1$. Thus, Stage i becomes a bubble. At the same time, Stage $i+1$, which had been a bubble, starts a new computation. Thus, the number of bubbles in the pipeline is conserved, and these bubbles move in the direction opposite that of the data. The relationships between bubbles and waves is described in Table 1.

n	number of processors
w	average wavelength
$k = n/w$	mode
$b = n - 2k = (w-2)n/w$	number of bubbles
$s = n/b = w/(w-2)$	average bubble spacing

Table 1. Relationships among fundamental pipeline parameters.

There is a trade-off in choosing how many bubbles to include in a pipeline. If there are not enough bubbles, utilization will be limited by C-elements that must wait for bubbles to arrive. If there are too many bubbles, utilization will be limited by processors that must wait for new data. In general, both forms of waiting will occur, and degrade the utilization of processors. However, if the processing time, t_f, and the storage time, t_c, are fixed, either waiting for bubbles or waiting for data can occur, but not both. We examine this case first.

6.2 Fixed Processing Times

If t_f and t_c are fixed, and the spacing between bubbles is small, there is no waiting for bubbles, as we will see later. At any time each stage is in one of the following three states:

1. Computing a new result (requires time t_f).
2. Storing a new result (requires time t_c).
3. Waiting for new data from the predecessor stage (which we define as t_b).

By definition, $U = t_f/(t_f+t_c+t_b)$. When a stage is in State 3, it is a bubble. It follows that $(s-1)t_b = t_f + t_c$. Since Stage i must remain a bubble for at least as long as it takes Stage $i-1$ to store a value in its C-element, $t_b \geq t_c$. This yields

$$U = \frac{s-1}{s}\frac{t_f}{t_f + t_c}, \qquad s \leq \frac{t_f}{t_c} + 2$$

In the second case processors wait for bubbles to arrive. This means there are a small number of bubbles in the pipeline (large spacing), and therefore each processor will complete its computation before the C-element of the successor has changed state. Once the bubble arrives, time t_c elapses before the C-element changes its output. With a spacing of bubbles of s, st_c time elapses between any two successive computations on the same processor. This yields a utilization of

$$U = \frac{t_f}{s\, t_c}, \qquad s \geq \frac{t_f}{t_c} + 2$$

Thus, in the small-spacing case, utilization is maximized by choosing s as large as possible, and vice-versa in the large-spacing case. It follows then that the highest utilization is obtained when $s = t_f/t_c + 2$, and

$$U_{opt} = \frac{t_f}{t_f + 2t_c}, \qquad s = \frac{t_f}{t_c} + 2$$

Thus, utilization arbitrarily close to 100% can be realized for designs with t_c sufficiently small.

The above arguments assume that waiting for bubbles does not occur if $s \leq t_f/t_c + 2$, and that waiting for data does not occur if s is greater than this value. It can be easily verified that in the steady-state this is always true. If $s < t_f/t_c + 2$ and a stage waits for a bubble, then the "late" bubble will be delayed by t_c (which is less than t_b) at each stage until it arrives at a stage which is waiting for data. Thus bubbles catch up with the computation (for $s < t_f/t_c + 2$). A similar argument shows that waiting for data does not occur if $s > t_f/t_c + 2$.

6.3 Bounded Processing Times

If t_f and t_c are bounded random variables, then the results for fixed times can be used to establish bounds on performance. Let t_w be the time spent waiting for a bubble to arrive (i.e. the stage has completed its computation, but the C-element cannot transfer the result to the output). Then,

$$U = \frac{E(t_f)}{E(t_f) + E(t_c) + E(t_b) + E(t_w)}$$

Since $(s-1)E(t_b) = E(t_f) + E(t_c) + E(t_w)$, and $E(t_w) \geq 0$, it follows that

$$U \leq \frac{s-1}{s}\frac{E(t_f)}{E(t_f) + E(t_c)}$$

This provides an upper bound on utilization.

To derive lower bounds on utilization, it is useful to consider a graph which models the computations of the pipeline. The vertices of the graph are $v(p,g)$ where p denotes a processor, $0 \leq p < n$, and g specifies the "generation" (i.e. how many computations processor p performed before the one corresponding to this vertex). Each computation requires two inputs (corresponding to the two inputs to each stage). There is an arc from $v(p,g)$ to $v(q,h)$ if the output of stage p at generation g is an input to stage q at generation h. The arc has a weight of t_c if it corresponds to the input which is connected directly to the C-element (a "storage" arc), and a weight of $t_f + t_c$ if the arc corresponds to the input of the processor for the stage (a "computation" arc). If t_f and t_c are random variables, then so are the arc weights. For completeness, there is a special vertex, v_r, which is the root of this directed acyclic graph. There is an arc of weight 0 from v_r to $v(p,0)$ for all p.

The time at which the computation of $v(p,g)$ starts is equal to the length of the longest path from v_r to $v(p,g)$. We denote the length of this path as $L(v(p,g))$. This can be seen by considering the operation of the C-element. The time at which the C-element becomes active is the later of the times at which the inputs become available, which corresponds to the longest path.

Let G be the computation graph for a pipeline, and $e = (v(p_1,g_1),v(p_2,g_2))$ be an arc in G. Let x be the weight of e. Consider G' which is identical to G except that arc e has weight x' with $x' \geq x$. Let $L'(v)$ be the length of the longest path from v_r to v in G'. Since $L'(v) \geq L(v)$ for all v, G' provides an upper bound on when computations take place in the pipeline represented by G.

Consider a pipeline where t_f and t_c are bounded random variables (e.g. rectangular). It follows immediately from the arguments above that

$$U \geq \frac{E(t_f)}{\max(t_f)} U_{\max}$$

where $U_{\max}$ is computed as for the deterministic case with $\max(t_f)$ and $\max(t_c)$. For sufficiently narrow distributions of t_f and sufficiently small t_c, utilizations arbitrarily close to 100% can be realized.

6.4 Exponentially Distributed Processing Times with Fixed Offset

Unlike synchronous pipelines, which must be designed according to worst-case timing assumptions, the operation of self-timed pipelines is determined primarily by the average-case performance. High utilization can be maintained even if the distribution of processing times is unbounded. As an example, we consider processing times which are exponentially distributed with an offset, in particular

$$\mu(t) = \begin{cases} 0 & t < t_0 \\ e^{-\lambda(t-t_0)} & t_0 \leq t \end{cases}$$

To simplify the analysis, we assume $t_c = 0$. (The analysis of Section 5 corresponds to $t_f = 0,\ t_c > 0$.)

The behavior of the pipeline can be divided into three cases depending on the spacing of bubbles:

1. Close spacing: $s \ll \lambda t_0$. Waiting for new data is the primary loss of utilization and the analysis for fixed t_f provides an upper bound which is close to the actual utilization:

$$U \leq (s-1)/s \tag{1}$$

2. Intermediate spacing: $s \approx \lambda t_0$. Both waiting for new data and waiting for bubbles occur frequently. Based on Monte-Carlo simulation, we observed

$$U < \lambda t_0/(1+\lambda t_0) \tag{2}$$

   is an upper bound and reasonable approximation. This is the region of maximum utilization.

3. Large spacing: $s \gg \lambda t_0$. Waiting for bubbles is the primary cause of lost utilization. We present a more detailed analysis of this case below.

We compute an estimate based on the rate of flow of bubbles around the pipeline for the case that $s \gg \lambda t_0$. Let t_a be the average time between starting a computation and the subsequent arrival of a bubble; for large s, $t_a > t_0$. Since t_c is zero, a stage is only a bubble for a positive amount of time if it becomes a bubble while its predecessor is still computing a new result. This happens with probability $e^{-\lambda(t_a - t_0)}$. When a bubble stops at a stage, the average length of the stay is $1/\lambda$. This yields:

$$\frac{(s-1)}{\lambda} e^{-\lambda(t_a - t_0)} = t_a$$

from which it follows that

$$t_a = t_0 + \frac{1}{\lambda} \log\left[\frac{s-1}{\lambda t_a}\right]$$

and

$$t_a + \frac{1}{\lambda} \log\left[\frac{t_a}{t_0}\right] = t_0 + \frac{1}{\lambda} \log\left[\frac{s-1}{\lambda t_0}\right]$$

As s goes to infinity, the right-hand side goes to infinity, and thus

$$\lim_{s \to \infty} \frac{t_a + (1/\lambda)\log(t_a/t_0)}{t_a} = 1$$

Since this assumes the bubble always arrives after t_0, this analysis provides a lower bound on waiting time, and thus an upper bound on utilization. In particular,

$$U < \frac{E(t_f)}{E(t_a)}$$

In the limit as $s \to \infty$, $t_a \to \infty$, and the bubble arrives after t_0 with probability approaching one. Combining this with the limiting form for t_a given above, we get

$$U \approx \frac{\lambda t_0 + 1}{\lambda t_0 + \log\left[\frac{s-1}{\lambda t_0}\right]} \tag{3}$$

These three upper bounds provide a good approximation of the utilization of the self-timed pipeline. Figure 6 shows data obtained from Monte-Carlo simulations along with the three upper bounds for a pipeline with $t_0 = 0.9$, $\lambda = 10$, (i.e. mean 1.0, variance 0.1, processing times) and $t_c = 0$. For distributions with λt_0 sufficiently large, utilizations arbitrarily close to 100% can be realized, even though the worst-case processing time is unbounded.

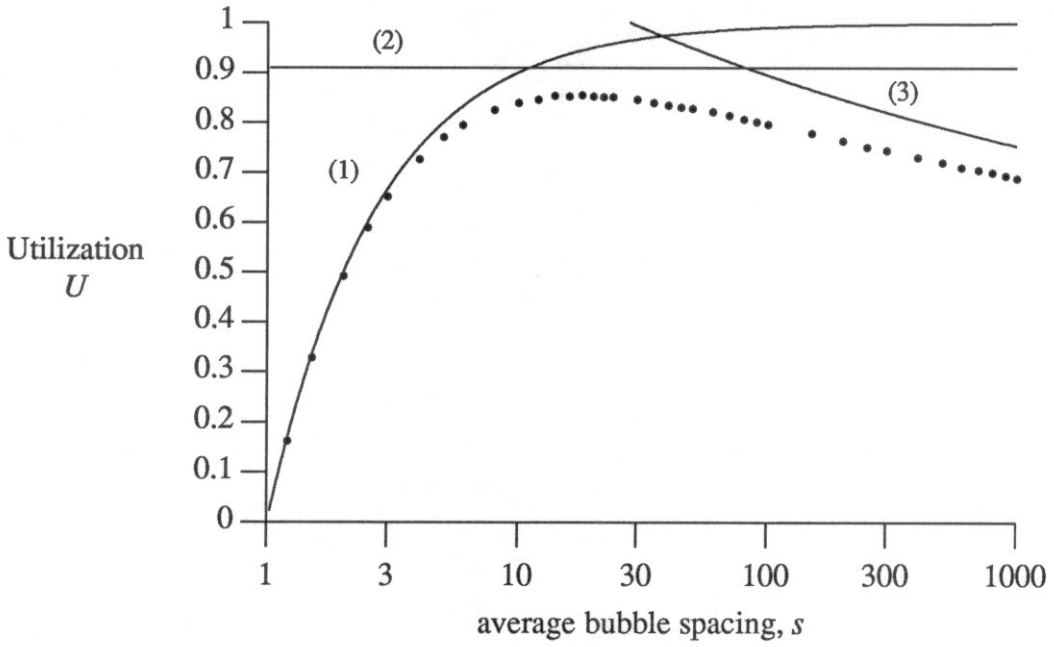

Fig. 6. Utilization vs. bubble spacing with $\lambda t_0 = 10$. Upper bounds shown by solid curves are labeled by equation number. The dots are the results of Monte Carlo simulation.

Each of these three bounds can be understood as considering different paths in the graph of Section 6.3. The case of performance limited by waiting for data (too many

bubbles) corresponds to the paths in which all arcs are computation arcs. Conversely, the case of waiting primarily for bubbles (too large spacing) corresponds to paths in which most arcs are storage arcs. Intuitively, optimal performance will be obtained when neither type of arc dominates in the longest path (a balance in the trade-off). For example, if these types of arcs occur with equal probability, estimates can be derived which correspond closely to (2).

The graph analysis can be used to derive bounds for processing and storage time distributions other than the ones presented here. Let F_1 and F_2 be two distribution functions. If for all t, $F_1(t) \leq F_2(t)$, then the pipeline with processing (storage) distributions of F_1 will achieve higher utilizations than one with distributions of F_2. For example, many unbounded distributions (e.g. Gaussian and gamma) can be bounded by exponential distributions with offsets. Thus, the results of this section can be extended to these distributions.

7. Conclusions

We have shown that self-timed pipelines can achieve linear speedup with utilization close to 100%, under a wide variety of processing time distributions, even unbounded distributions. The variation in processing time is absorbed by bubbles, processors which are temporally idle. By introducing bubbles, the pipeline can operate at a rate which is closer to the average processing time than the worst-case.

The techniques presented here can also be applied to synchronous designs. The C-element ring can be used instead of a buffer chain to distribute clock pulses to each stage in a synchronous pipeline. Feedback between stages in the C-element protocol guarantees correct functioning independent of all delays; thus, this design does not suffer from the limitations of buffer chains described in Section 2. This approach combines the simplicity of synchronous design (for the individual stages) and the robustness of self-timed designs (for interstage timing). As this is a form of separate completion, the timing skew between stages must be kept small relative to the clock period.

8. Acknowledgements

We thank Ehran Cinlar, Claire Kenyon-Mathieu, and F. Miller Maley, whose comments have been helpful in deriving the results presented in this paper.

References

1. U. Frisch, B. Hasslacher, Y. Pomeau, "A Lattice Gas Automaton for the Navier-Stokes Equation," *Phys. Rev. Lett.*, vol. 56 , no. 14, pp. 1505-1508, April 7, 1986.
2. S. D. Kugelmass, K. Steiglitz, "Design and Construction of LGM-1: a Lattice-Gas Machine with Linear Speedup," *Proc. 1988 Princeton Conference on Information Sciences and Systems,* Princeton, pp. 107-114, March 16-18, 1988.
3. C. L. Seitz, "System Timing" in *Introduction to VLSI Systems,* C. A. Mead and L. A. Conway, Addison-Wesley, 1980; pp. 245-258.
4. A. L. Fisher and H. T. Kung, "Synchronizing Large VLSI Processor Arrays," *IEEE Trans. on Computers*, vol. C-34, no. 8, pp. 734-740, Aug. 1985.

5. M. R. Greenstreet, T. E. Williams and J. Staunstrup, ''Self-Timed Iteration,'' *VLSI '87: Proc. of the Int. Conf. on VLSI*, Vancouver, Aug. 10-12, 1987.

6. R. E. Miller, *Switching Theory,* Wiley, New York, 1965.

7. A. O. Allen, *Probability, Statistics, and Queuing Theory, with Computer Science Applications,* Academic Press, 1978.

8. T.E. Williams, M. Horowitz, R.L. Alverson, and T.S. Yang, ''A Self-Timed Chip for Division,'' *Proc. 1987 Stanford VLSI Conference,* Stanford, CA, pp. 75-95, MIT Press, 1987.