Journal of VLSI Signal Processing, 2, 139–148 (1990)

Bubbles Can Make Self-Timed Pipelines Fast†

MARK R. GREENSTREET AND KENNETH STEIGLITZ
Department of Computer Science, Princeton University, Princeton, NJ 08544

Received November 23, 1988, Revised November 16, 1989.

Abstract. We explore the practical limits on throughput imposed by timing in a long, self-timed, circulating pipeline (ring). We consider models with both fixed and random delays and derive exact results for pipelines where these delays are fixed or exponentially distributed random variables. We also give relationships that provide upper and lower bounds on throughput for any pipeline where the delays are independent random variables. In each of these cases, we show that the asymptotic processor utilization is independent of the length of the pipeline; thus, linear speedup is achieved. We present conditions under which this utilization approaches 100%.

1. Introduction

Many problems 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 problems of the same size, and with no increase either in the complexity of the processor or the communication bandwidth. Examples of such designs include systolic arrays for signal processing applications [1], chips for matching subsequences in DNA strings [2], and a machine [3] for the lattice-gas model of Frisch, Hasslacher, and Pomeau [4]. Each of these designs is a simple concatenation of identical stages, and the system has *linear speedup—n* stages have n times the throughput of one.

To realize linear speedup, the period of computation (the time for each processor to complete a computation) must be independent of the length of the pipeline. In traditional, synchronous designs, the period of computation is determined by the period of the clock. One important practical limit of synchronous designs is the need to distribute a clock so that the timing requirements for inter-processor communication are satisfied. Typical methods of clock buffering do not guarantee that a clock pulse can be propagated reliably through an arbitrarily long chain of buffers (see Appendix 1). Unless the clock period is increased as the pipeline is made longer, accumulation of local variations can ultimately produce violations of the clocking requirements. Self-timed signaling [5], [6] as described in this paper, ensures reliable communication with a period of computation that is independent of the pipeline length.

Section 2–4 provide background material on self-timed designs. In Sections 5–8, we consider the throughput of a circular, self-timed pipeline with several models for processing and storage times. We consider fixed times and random times with bounded and exponential distributions. Under each of these distributions, asymptotic linear speedup is realized. However, different constant factors are realized for the different distributions. For several, utilization approaching 100% can be obtained for arbitrarily long pipelines.

2. Self-timed Pipelines

In self-timed pipelines, the flow of computation is controlled by *completion* and *acknowledge* signals. As we will show, this form of control does not suffer from the asymptotic throughput limitations of synchronous designs described in Appendix 1. When Stage i completes a computation, it sends a completion signal to Stage $i + 1$ (modulo n, the number of stages in the ring). When Stage i receives new input data, it sends an acknowledge signal to Stage $i - 1$. Figure 1 shows the self-timed pipeline that we will analyze in this paper.

The boxes labeled f perform the computation; the circles labeled C store the results and control the flow of data. In this figure, data flows to the right on the upper branches, and acknowledge signals flow to the left on the lower branches.

†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.

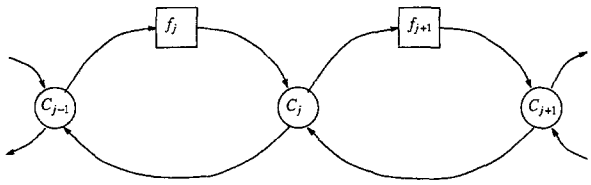


Fig. 1. A self-timed pipeline. The processors are labeled f and the C-elements C.

Register C_j is the output register for f_j and simultaneously the input register for f_{j+1}. What this means is that a new result cannot be stored in C_{j+1} until f_{j+1} has completed its computation using the old result and stored it in C_{j+1}. The contents of C_j cannot change until two things happen: (1) f_j completes its computation of a new result, and (2) C_{j+1} acknowledges receipt of a result (which means that the old result from C_j has been used and can be overwritten).

We can visualize this sequence of events by placing tokens on the signal lines. Figure 2(a) shows a situation at time t_1: The token at the upper left input of C_j is the completion signal from f_j; the token at the lower right input is the acknowledge signal from C_{j+1}. At time t_2 (figure 2(b)), C_j stores the new result, transferring the completion token to its upper right output and the acknowledge token to its lower left output, the lower right input of C_{j-1}. At time t_3, f_{j+1} computes a new result, transferring the completion token to the upper left input of C_{j+1}.

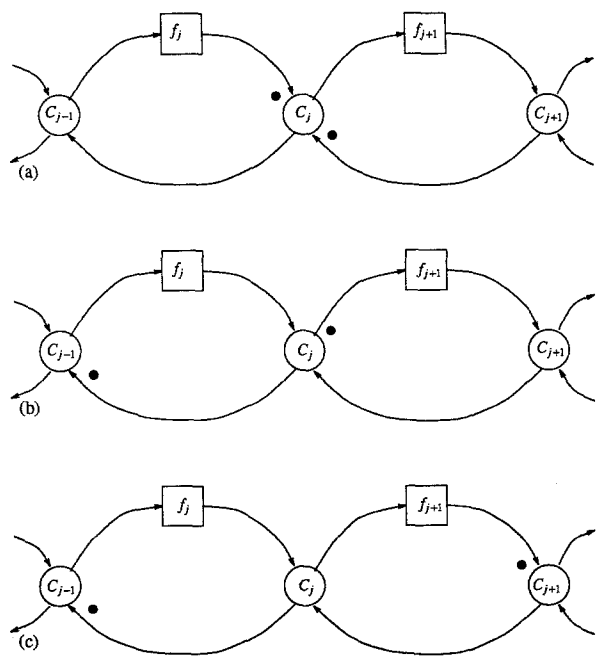


Fig. 2. Tokens in a self-timed pipeline. Part (a) shows the state when C-element C_j is about to fire; (b) shows the state after firing; and (c) shows the state after processor f_{j+1} has completed its computation.

What we have described in this section summarizes well known ideas from the literature of self-timed circuits and sets the stage for the analysis of such pipelines. The token-passing view of the pipeline is a simple instance of a Petri Net [7]. The storage element C_j functions as a Muller C-element [5] insofar as control is concerned. In [8] it is shown that three-stage pipelines of this form function correctly regardless of the delays of the f and C elements. In [9] this result is generalized to pipelines of any number of stages (≥ 3).

3. Basic Properties of the Self-Timed Pipeline

From the rules governing the pipeline, it is easy to see that the number of tokens is invariant and equal to the number of stages. Let $loop_j$ be the cycle (input of f_j) $\rightarrow$ (upper left input of C_j) $\rightarrow$ (lower right input of C_{j-1}) as indicated in figure 3. Each loop contains exactly one token, and it is either a completion or an acknowledge token. We say that each such loop is in one of the three states *left*, *right*, or *down*, according to the location of the token. By the rules governing the flow of data and signals, the number of loops in the *down* state is invariant.

We call the processor f_j *active* if $loop_j$ is in state *left*. This corresponds to the intuitive notion of active: f_j begins a new computation when the token (i.e., new data) arrives at its input, and transfers the token from *left* to *right* when the computation is completed. In the same spirit, we say that C_j is *active* when $loop_j$ is in the *right* state and $loop_{j+1}$ is in the *down* state. This means that C_j is allowed to fire, sending an acknowledge back to C_{j-1} and data forward to f_{j+1}.

If all loops were in the *left* state, all processors would be active. However, once these computations were completed, all loops would be in the *right* state, and the pipeline would be deadlocked. For progress to be made,

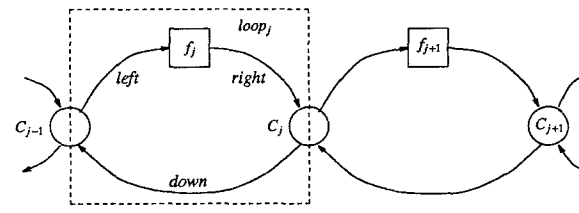


Fig. 3. Definition of $loop_j$.

C-elements must be enabled to fire; this requires that there be loops in the *down* state.

We refer to a loop in the *down* state as a *bubble*. Let n be the number of loops in the pipeline, and b be the number of bubbles. The number of active processors is bounded from above by $n - b$. The number of active C-elements is bounded from above by $min\ (b, n - b)$ because every active C-element must have a loop in the *down* state to its right (and there are b such loops) and a loop in the *right* state to its left (and there are at most $n - b$ such loops).

4. Performance of Self-Timed Pipelines

We describe the performance of a pipeline with two related quantities: *throughput* and *utilization*. The *throughput* is the total rate of computations made by all processors. If the throughput is proportional to the number of processors in the limit as the number of processors goes to infinity, we say that the pipeline has the *linear speedup* property. The *utilization* U of a processor is the fraction of the time that the processor spends computing. In the cases considered here, all processors have the same utilization. A utilization of 100% indicates that every processor is computing all of the time. The pipeline has linear speedup if the utilization is bounded from below by a positive constant as the number of processors goes to infinity. In the analysis of the following sections, we will derive exact values and bounds for utilization under various conditions.

In general, the times for computation and storage operations are random variables. A *realization* of a pipeline is given by an initial state and an infinite set of values, the times for each operation. As shown in Appendix 2, Corollary 2.1, this completely specifies the operation of a pipeline. We assume that the random variables for computation times are independent and identically distributed; t_f denotes a random variable with this distribution. Likewise, we assume that storage times are independent and identically distributed and write t_c to denote one such variable.

To compute the utilization of a pipeline, we must determine the fraction of the time each processor is active. By the strong law of large numbers, the average time per computation for any processor in a given realization is $E[t_f]$ almost surely [10]. The *average waiting time* for a processor is the average time between starting successive computations. For each realization, the average waiting time is the same for every processor. For the pipelines considered in this paper, the average waiting time is the same for almost every realization (see Appendix 2), and we denote it by T. In this case, we define

$$U = \frac{E[t_f]}{T}$$

5. Fixed Processing and Storage Times

We first consider pipelines with fixed t_f and t_c and a single bubble ($b = 1$). Each time f_j completes a computation, $loop_j$ must cycle through states *left*, *right*, and *down*. The average waiting time T is the sum of the average times spent in each of the three states *left*, *right*, and *down*. The time spent in the *left* state is always t_f, and the time spent in the *right* state is at least t_c. Let T_{down} be the average time spent in the *down* state. We have $T \geq t_f + t_c + T_{down}$. By a simple balancing argument, $nT_{down} = T$. Combining these two relationships yields $T \geq [n/(n-1)](t_f + t_c)$. To obtain a second bound on T, note that the bubble must travel around the ring each cycle; so $T \geq nt_c$. We claim that for any pipeline the greater of these two bounds gives the exact value for T; therefore, the utilization U of the pipeline is given by

$$U = \frac{n-1}{n}\frac{t_f}{t_f + t_c}, \quad n \leq \bar{s}$$

$$= \frac{t_f}{nt_c}, \quad n \geq \bar{s}$$

where $\bar{s} = t_f/t_c + 2$.

To see this, consider first the case when $n \geq \bar{s}$. After the bubble leaves $loop_j$ (to $loop_{j-1}$), at least $(n-2)t_c$ time units elapse before the bubble arrives at $loop_{j+1}$. Therefore, processor f_j has completed its computation by the time the bubble appears at the lower-right input of C_j. The rate of progress is completely determined by the progress of the bubble, and the bubble spends exactly t_c time units in each loop. Thus, $T = nt_c$ justifying the throughput in this case. The other case, when $n \leq \bar{s}$, follows by a similar argument. As $n \to \infty$, the utilization of the pipeline with a single bubble is $\Omega(1/n)$.

To improve utilization, more bubbles must be added to the pipeline. Let s be the average spacing of bubbles, $s = n/b$. First consider a pipeline where all the bubbles are initially equally spaced. Because all processing and storage times are identical and the initial pipeline configuration is symmetric, each bubble will propagate identically (see Appendix 2, Corollary 2.1). Since the bubbles are indistinguishable, the analysis of the single bubble case applies with n replaced by s. Appendix 2,

Theorem 3 shows that for a large class of pipelines, including this case, the initial configuration of bubbles does not affect the utilization of a pipeline. We conclude

$$U = \frac{s-1}{s}\frac{t_f}{t_f + t_c}, \quad s \leq \bar{s}$$

$$= \frac{t_f}{st_c}, \quad s \geq \bar{s}$$

In either case, the throughput depends only on the average spacing of bubbles. In particular, it is independent of the length of the pipeline. Thus, for any fixed spacing, linear speed-up is achieved. Optimal performance is achieved by choosing $s = \bar{s}$, which yields

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

Thus, linear speed-up is achieved by pipelines with fixed processing and storage times. Furthermore, utilizations arbitrarily close to 100% can be achieved for $t_f >> t_c$.

6. Bounded Processing and Storage Times

We now consider pipelines where t_f and t_c are specified by their means and maxima. We derive lower bounds for throughput by deriving an upper bound for T. To do this, consider a pipeline where each computation requires time exactly $\max(t_f)$ and each storage requires time exactly $\max(t_c)$. We will call this the *slow pipeline*. Let U' and T' refer to the slow pipeline. By definition, $T' = \max(t_f)/U'$, and from Appendix 2, Corollary 3.2, we have $T \leq T'$ which yields $U \geq (E[t_f]/\max(t_f))U'$. Since these $\max(t_f)$ and $\max(t_c)$ are fixed, the analysis of the previous section applies, and U' can be calculated. This yields the bounds

$$U \geq \frac{s-1}{s}\frac{E[t_f]}{\max(t_f) + \max(t_c)}, \quad s \leq \frac{\max(t_f)}{\max(t_c)} + 2$$

$$\geq \frac{E[t_f]}{s\max(t_c)}, \quad s \geq \frac{\max(t_f)}{\max(t_c)} + 2$$

Again, linear speedup is achieved for any fixed s. If t_c is sufficiently small and $\max(t_f)$ is sufficiently close to $E[t_f]$, then utilizations arbitrarily close to 100% can be achieved for an appropriate choice of s. Upper bounds for U can be obtained in a similar manner using $\min(t_f)$ and $\min(t_c)$.

It may seem pessimistic to estimate the utilization based on the worst-case delays. One would hope for a pipeline whose performance is determined primarily by the average processing and storage times. However, we can propose pipelines where the bounds given above are tight in the limit as the number of processors goes to infinity. For example, consider a pipeline where t_f is $1/p$ with probability p and zero with probability $1 - p$, and $t_c = 0$. This approximates the behavior of a pipeline where most operations are fast, but infrequently computations take a long time. If the pipeline is operated with a single bubble, we can show that $U = p + 1/n$. In contrast, for a pipeline with processing times fixed at $E[t_f] = 1$ and one bubble the analysis of Section 5 applies, and $U_{fixed} = (n-1)/n$. In the limit as the number of processors goes to infinity,

$$\frac{U}{U_{fixed}} = \frac{E[t_f]}{\max(t_f)}$$

Thus, in the single-bubble case, linear speedup is achieved, but utilization is limited by the worst-case processing time. Better utilization can be achieved by adding more bubbles; however, this example shows that the performance of self-timed systems is not necessarily determined by average case-timings (as has often been claimed).

7. Exponential Processing and Storage Times

We now consider a case where the worst-case processing and storage times are not bounded. We show that linear speed-up (relative to expected processing time) is still achieved. To exploit techniques from queuing theory, we make two simplifying assumptions:

1. The operations of the processor and the C-element are combined into a single operation. This could be achieved by taking the acknowledge signal to C_{j-1} from the output of f_{j+1} instead of the output of C_j.
2. The time for the combined operation of storage and computation $(t_c + t_f)$ is exponentially distributed with mean τ.

With this change, the *left* state is never visible. When $loop_{j-1}$ is in state *right* and $loop_j$ is in state *down*, a transition is enabled that brings $loop_{j-1}$ into state *down* and $loop_j$ into state *right*. Because this could be effected by delaying the acknowledge inputs of C-elements, the throughput of this modified pipeline is less than or equal to the throughput of the original pipeline (Appendix 2, Theorem 2).

A *segment* refers to a sequence of adjacent loops between bubbles plus the bubble on the right. 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 (no departure possible since bubbles cannot be destroyed). Let λ be the rate of arrivals to a queue; the average waiting time T is s/λ. 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, $\lambda = (1 - p_1)/\tau$, where p_1 is the probability that the length of a segment is 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. Furthermore [11], these lengths are independent random variables as $b \to \infty$. This is an M/M/1 queuing system [12]. The departure rate is $1/\tau$: whenever a segment includes more than two loops, the bubble can move. The expected length of an M/M/1 queue is $(1 - p_1)/p_1$; accordingly, the expected length of a segment is $1/p_1$. Since the average segment length must be $n/b = s$ by the definition of segments, we have $p_1 = 1/s$. From this, we have $T = \tau s^2/(s - 1)$ and

$$\lim_{n\to\infty} U = \frac{s - 1}{s^2} \frac{E[t_f]}{\tau}$$

From this result it follows that choosing $s = 2$ maximizes the asymptotic value of U, and that this maximum value is $U = E[t_f]/4\tau$. As mentioned above, this is a lower bound for utilization, because the simplifications made for the analysis were conservative. For any fixed s, linear speedup is realized.

We have shown that asymptotically good utilizations can be realized even when the worst-case processing time is unbounded. In contrast, the clock period of a synchronous system must be longer than the worst-case delay. The above result demonstrates that a self-timed pipeline can operate with performance determined by the average processing time instead of the worst-case.

8. Exponentially Distributed Processing Times with Fixed Offset

We now show that a self-timed pipeline can achieve high utilization even when processing times are unbounded. We consider processing times that are exponentially distributed with an offset:

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

To simplify the analysis, we assume $t_c = 0$. (The analysis of Section 7 corresponds to $t_f = 0$, t_c exponential with parameter τ.)

We cannot give exact analytic results for this case; instead, we derive approximate results for three cases depending on the spacing of bubbles, and we present the results of Monte-Carlo simulations that confirm these approximations.

1. Close spacing: $s << \lambda t_0$. Because there are many bubbles, a loop will most often proceed directly from the *right* to the *down* state without waiting. In the limiting case as s goes to zero, the average time spent in the *right* state is zero. By arguments similar to those in Section 5, $(s - 1)T_{down} = E[t_f]$. Occasional waiting in the *right* state can only slow down the pipeline; thus,
$$U \le (s - 1)/s, \qquad s << t_0 \qquad (1)$$
Monte-Carlo simulation shows that this bound is close to the actual utilization.
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
$$U < \lambda t_0/(1 + \lambda t_0), \qquad s \approx t_0 \qquad (2)$$
is an upper bound and reasonable approximation. This is the region of maximum utilization.
3. Large spacing: $s >> \lambda t_0$. Waiting for bubbles to arrive 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 $s >> \lambda t_0$. Since t_c is zero, a loop is a bubble for a positive amount of time only if it becomes a bubble while its predecessor is still computing a new result. Estimating the actual waiting time between computations with the average, T, this happens with probability approximately $e^{-\lambda(T-t_0)}$. When a bubble stops at a stage, the average length of the stay is $1/\lambda$. This yields:

$$\frac{s}{\lambda} e^{-\lambda(T-t_0)} \approx T$$

It is easy to verify that in the limit as s goes to infinity

$$T \approx t_0 + \frac{1}{\lambda} \log\left(\frac{s}{\log s + \lambda t_0}\right)$$

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

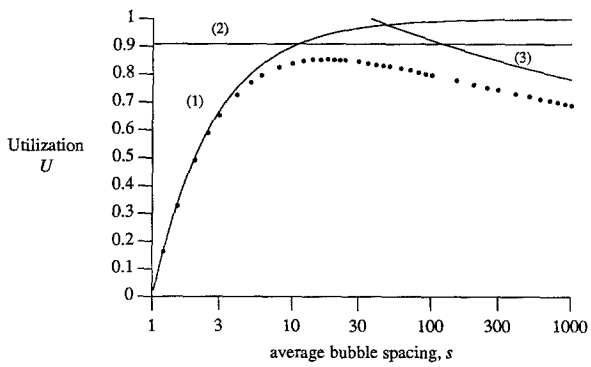


Fig. 4. Utilization vs. bubble spacing with $\lambda t_0 = 10$. Upper bounds shown by solid curves are labeled by equation number. The dots are the results of Monte-Carlo simulation.

limit as $s \to \infty$, $T \to \infty$, and the bubble arrives after t_0 with probability approaching one. This yields

$$U \approx \frac{\lambda t_0 + 1}{\lambda t_0 + \log\left(\frac{s}{\log s - \lambda t_0}\right)}, \quad s >> \lambda t_0 \qquad (3)$$

These three upper bounds provide a good approximation to the utilization of the self-timed pipeline. Figure 4 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.

9. 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, including unbounded distributions. The variation in processing time is absorbed by bubbles, processors which are temporarily idle. By introducing bubbles, the pipeline can operate at a rate which is closer to the average processing time than the worst-case.

The theorems of Appendix 2 provide a general framework for analyzing the throughput of self-timed pipelines. In particular using Corollary 3.2, performance bounds (both upper and lower) for many other processing time distributions can be derived from the results presented in Sections 5–8. The C-element protocol has a natural extension to higher dimensions. The bounds from Section 6 (bounded distributions) also apply to this case. In particular, assuming bounded processing and communication time, two-dimensional processor networks can also realize linear speedup. For higher dimensional networks, the delays of non-nearest neighbor communication may limit performance to less than this. Our future work will include investigation of higher dimension cases, especially for more general distributions.

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 in a pipeline with locally synchronous stages. 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 Appendix 1. This approach combines the simplicity of synchronous design (for the individual stages) and the robustness of self-timed designs (for interstage timing).

Acknowledgments

We thank Ehran Çinlar, Marios Dikaiakos, Claire Kenyon-Mathieu, and F. Miller Maley for helpful comments.

Appendix 1. Synchronous Pipelines

In this appendix, we examine clock buffering for synchronous designs and show that many typical designs cannot guarantee both linear speedup and reliable pipeline operation. The arguments are adversarial, so the results show that failure is possible, although not necessarily probable. More work needs to be done to understand when the limits of synchronous clocking are reached in practical systems.

In [13], Fisher and Kung present several clock distribution designs. Because signals are necessarily attenuated when propagated large distances, these designs must be implemented with chains of buffers, as illustrated in figure 5. They require the skew introduced by each buffer to be 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 ([13], Assumption A8).

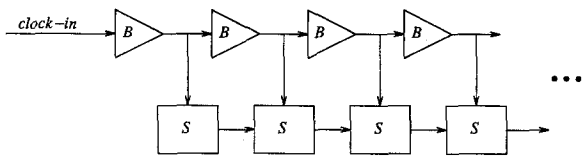


Fig. 5. A linear array of processing stages, clocked by a linear array of buffered clock signals, after [13]. The buffers are labeled "*B*," and the processing stages "*S*."

Consider differences in delay between leading and falling edges in the buffers of figure 5. 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 interchanged). 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 in the worst-case, 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 random variables. We assume that they are bounded 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. If subsequent stages continue to delay the first edge relative to the others, the interval between the second and third edges will be reduced, and so on. Thus we can produce an arbitrarily long sequence of successive pulses separated by w.

The second step of the construction exploits variations in w. Suppose that such a sequence of pulses, separated by $\min(w)$, encounters a stage which imposes a separation of $\max(w)$. Because a one-shot must have finite memory, the stage must ultimately lose a pulse.

Appendix 2. Properties of Self-Timed Pipelines

In this appendix, we derive three theorems for pipelines. First we present a sufficient condition for liveness. Then we give a proof that the waiting time cannot be decreased by increasing the delay of one or more operations in a realization. Finally, we show sufficient conditions under which the average waiting time T is well defined.

Let n be the number of processors (and therefore the number of storage elements) in the pipeline, b be the number of bubbles, and $s = n/b$ be the average spacing of bubbles. All operations on processor, storage element, and loop indices are implicitly modulo n. Each loop (as described in Section 3) can be in one of the three states *down*, *left*, or *right*.

THEOREM 1. For any pipeline with $1 < s \leq n$, and for any distributions for t_c and t_f, at least one processor or C-element is active at any given time.

By the definition of s, $1 < s \leq n$ is equivalent to $1 \leq b < n$, which means that there is at least one loop that is a bubble and at least one loop that is not a bubble. Since the loops are arranged in a ring, we can find a j such that $loop_j$ is in state *down* and $loop_{j-1}$ is in state *left* or *right*. If $loop_{j-1}$ is in state *left*, then processor f_{j-1} is active. Otherwise, $loop_{j-1}$ is in state *right* and C-element C_{j-1} is active. In both cases the claim is established.

The next theorem shows that the throughput of a self-timed pipeline cannot be increased by slowing down any one or more operations. In general, the times for computation and storage are random variables. A *realization* is an initial state and a delay for each operation. The initial state can be specified by a function q where $q(j)$ is the initial state (*down*, *left*, or *right*) of loop j. The delays of computation and storage operations can be specified by a positive (but not infinite), real valued function, *delay*(i, j, *what*), where i is the iteration, j is the position in the ring, and *what* is either *processor* or *C-element*. In particular, *delay*(42, 17, *C-element*) =

1.23 means that C_{17} takes 1.23 time units (after its inputs are both available) to perform its 42^{nd} storage operation. Likewise, we define *start*$(i, j, what)$ to be the time at which the inputs become available for $what_j$ to start its i^{th} operation. We write $delay_1 \leq delay_2$ to denote that for all i, j, and *what*, $delay_1(i, j, what) \leq delay_2(i, j, what)$. We define $start_1 \leq start_2$ in the same manner.

THEOREM 2. Let $R_1 = (q_1, delay_1)$ and $R_2 = (q_2, delay_2)$ be two realizations of a pipeline. If $q_1 = q_2$ and $delay_1 \leq delay_2$, then $start_1 \leq start_2$.

Assume otherwise. Then, of all operations that start earlier in R_2 than in R_1, let $(i, j, what)$ be the first.

If *what* = *processor*, this means that the output of the preceding C-element changed sooner in R_2 than in R_1. By the choice of $(i, j, what)$ to be the first violation of the claim, this C-element started its operation no earlier in R_2 than in R_1. Furthermore, the C-element in R_2 took at least as long to complete its operation as the one in R_2 by the hypothesis $delay_1 \leq delay_2$. Thus, the input to f_j became available no earlier in R_2 than in R_1. A contradiction.

If *what* = *C-element*, a similar argument leads to the required contradiction.

Corollary 2.1. Let $R_1 = (q_1, delay_1)$ and $R_2 = (q_2, delay_2)$ be two realizations of a pipeline. If $q_1 = q_2$ and $delay_1 = delay_2$, then $start_1 = start_2$.

Proof. Since $delay_1 \leq delay_2$, Theorem 2 implies $start_1 \leq start_2$. Likewise, $delay_2 \leq delay_1$ implies $start_2 \leq start_1$. Therefore, $start_1 = start_2$ as claimed.

Corollary 2.1 shows that the behavior of a self-timed pipeline (as described by starting times of operations) is completely determined by the initial state and the delay function.

We now give a condition that guarantees that utilization depends only on the distributions of processing and storage times and the number of bubbles in the pipeline.

Condition 1. For any fixed i, j, and *what*, *delay*$(i, j, what)$ is a random variable. Condition 1 is satisfied if

1. All of these random variables are independent.
2. For fixed j and *what*, the resulting family of random variables are identically distributed.

The first condition requires that the time to perform any given computation or storage operation is independent of the time to perform other operations. The second condition requires that any given processor (or storage element) has the same distribution of computation (or storage) times for all iterations.

We now introduce the idea of *restarting* that will allow us to analyze aggregate properties of (almost) all realizations. For any integer k, after each processor and C-element has completed exactly k operations, the state of the pipeline is identical to the initial state. The *restart-at-k* of a realization $R = (q, delay)$ is a realization $R' = (q, delay')$ such that $delay' \leq delay$ and all processors and C-elements complete their k^{th} operation at the same time. In particular, we consider the first processor or C-element to complete its k^{th} operation; let this occur at time τ_k. We reduce delays (in *delay'* relative to *delay*) for operations in progress at this instant so that they will also complete at time τ_k. Finally, we set the delays of all remaining operations of iterations at or before k to zero to guarantee that each processor and C-element completes its k^{th} operation at time τ_k. By construction $delay' \leq delay$ and by Theorem 2, $start' \leq start$. Since the pipeline is restarted to a fixed state (independent of the state before the restart operation), and the delays of computation and storage operations are independent random variables (assuming Condition 1), the behaviors of the pipeline before and after the restart are independent. The *restart-every-k* of a realization $R = (q, delay)$ is a realization $R' = (q, delay')$ such that $delay' \leq delay$ and all processors and C-elements complete their mk^{th} operation at the same time for all integer $m \geq 1$.

THEOREM 3. If a pipeline satisfies Condition 1, then the limit

$$T = \lim_{m \to \infty} \frac{start(m, j, what)}{m}$$

exists almost surely and has the same value for all j and *what*, and all initial states with the same number of bubbles.

Proof. In the following, we assume $1 \leq b < n$ to guarantee that the pipeline is alive. Otherwise, the pipeline deadlocks, and the limit is infinity for all realizations. We first show that the limit exists for any fixed initial state and for any integer k, when each realization is replaced by its restart-every-k version. We then show that in the limit as k goes to infinity, the limit for the restart-every-k version is the limit for the original pipeline. Finally, we show that this limit is independent of the initial state, j, and *what*.

Let $start_k$ be the starting times of a restart-every-k version of a realization. We consider

$$T_k = \lim_{m\to\infty} \frac{start_k(m, j, what)}{m}$$

Let $\delta_k(1) = start_k(1, j, what)$ and $\delta_k(m) = start_k(m, j, what) - start_k(m-1, j, what)$. Then we can write the above limit as

$$T_k = \lim_{m\to\infty} \frac{1}{m} \sum_{i=1}^{m} \delta_k(i) = \lim_{m\to\infty} \frac{1}{m} \sum_{i=0}^{m-1} \frac{1}{k} \sum_{h=1}^{k} \delta_k(ik + h)$$

Because the operation of the pipeline before any restart is independent of the operation after the restart, the sums over h are independent random variables. Thus, by the strong law of large numbers [10], the limit exists almost surely. Since $start_k \le start$, we have

$$\liminf_{m\to\infty} \frac{start(m, j, what)}{m} \ge T_k$$

We now derive an upper bound, and show that these two are equal in the limit as k goes to infinity. In a pipeline with b bubbles, the difference at any time in the number of operations performed by any two processors or C-elements is at most $b + 1$. Therefore, the number of processor delays that are decreased by each restart is at most $n(b + 1)$. Likewise, at most $n(b + 1)$ C-element delays are decreased in the construction of *delay'*. The largest difference in starting times between the original realization and its restarted counterpart would occur if these operations were performed sequentially in the original. Let

$$gap_i = \sum_{h=(i+1)k-b}^{(i+1)k} \sum_{j=0}^{n-1} \Big(delay(h, j, processor) + delay(h, j, C\text{-}element) \Big)$$

For each i, gap_i is an upper bound on the decrease of starting times due to the i^{th} restart. This yields

$$\limsup_{m\to\infty} \frac{start(m, j, what)}{m} \le \lim_{m\to\infty} \frac{1}{m} \sum_{i=0}^{m-1} \frac{1}{k} \left(gap_i + \sum_{h=1}^{k} \delta_k(ik + h) \right)$$

Applying the strong law of large numbers again, the right side of the inequality becomes

$$\lim_{m\to\infty} \frac{1}{m} \sum_{i=0}^{m-1} \frac{1}{k} (E[gap] + T_k)$$

Taking limits as k goes to infinity yields

$$\limsup_{k\to\infty} T_k \le \liminf_{m\to\infty} \frac{start(m, j, what)}{m} \le \limsup_{m\to\infty} \frac{start(m, j, what)}{m} \le \liminf_{k\to\infty} \left(\frac{E[gap]}{k} + T_k \right)$$

almost surely. $E[gap]$ is bounded by $n(b + 1)(E[t_f] + E[t_c])$; therefore, $\lim_{k\to\infty} E[gap]/k = 0$, and we conclude

$$\liminf_{m\to\infty} \frac{start(m, j, what)}{m} = \limsup_{m\to\infty} \frac{start(m, j, what)}{m},$$

almost surely. Therefore, for any fixed initial state the limit exists almost surely as claimed.

The final step is to show that this limit is the same for all initial states with the same number of bubbles, and for all j and *what*. Let q_1 and q_2 be two states with the same number of bubbles. Let T_1 and T_2 be their respective values of the above limit. We note that a pipeline initially in state q_1 can be brought into state q_2 by setting the delays of a bounded, $O(nb)$, number of operations to zero. Thus, $T_2 \le T_1$. By symmetry, we have $T_1 \le T_2$ and therefore $T_2 = T_1$. As noted above, the difference in the number of operations performed by any two processors or C-elements is at most $b + 1$. Because each loop completes any finite number of operations in a finite amount of time, the difference between starting times of different loops and processors is insignificant in the limit. Thus, the limit exists and is the same for all initial states, and for all j and *what*.

Corollary 3.1. If a pipeline satisfies Condition 1, then its utilization is defined.

Proof. The average waiting time is

$$T = \lim_{m\to\infty} \frac{start(m, j, what)}{m}$$

By Theorem 3, this limit exists almost surely. Therefore, the utilization $U = E[t_f]/T$ is defined.

Corollary 3.2. Let P_1 and P_2 be two pipelines that satisfy Condition 1 and have the same number of processors and bubbles. Let F_1 and C_1 be the distribution functions for processing and storage times respectively in P_1, and F_2 and C_2 be the same for P_2. If $F_1 \ge F_2$ and $C_1 \ge C_2$, then $T_1 \le T_2$.

Proof. By Theorem 3, T_1 and T_2 are defined. By Theorem 2 and the monotonicity of averages, it is sufficient to exhibit a mapping from realizations of P_1 to realizations of P_2 with the following two properties: (1) the realization in P_2 has a *delay* function that is greater than or equal to that of the realization in P_1, and (2) the mapping preserves the probability of measurable sets. If F_1, C_1, F_2, and C_2 are continuous this can be achieved for each realization and by mapping $delay_1(i, j, processor)$ to $F_2^{-1}(F_1(delay_1(i, j, processor)))$ and likewise for C-element delays. Since $F_1 \geq F_2$

$$F_2^{-1}(F_1(delay_1(i, j, what))) \geq delay_1(i, j, what)$$

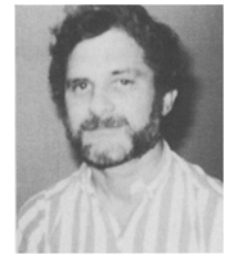
as required. If one or more of the distributions are not continuous, suitable variations of this simple mapping will produce the required result.

References

1. H.T. Kung, L.M. Ruane and D.W.L. Yen, "A Two-Level Pipelined Systolic Array for Convolution," *Proc. of the CMU Conf. on VLSI Systems and Computation*, Pittsburgh, PA, 1981.
2. R.J. Lipton and D. Lopresti, "A Systolic Array for Rapid String Comparison," *Proc. of the 1985 Chapel Hill Conf. on VLSI*, Chapel Hill, NC, 1985.
3. S.D. Kugelmass and K. Steiglitz, "A Scalable Architecture for Lattice-Gas Simulations," *J. Computational Physics*, vol. 84, 1989, pp. 311–325.
4. U. Frisch, B. Hasslacher and Y. Pomeau, "A Lattice Gas Automaton for the Navier-Stokes Equation," *Phys. Rev. Lett.*, vol. 56, 1986, pp. 1505–1508.
5. C.L. Seitz, "System Timing," in *Introduction to VLSI Systems*, C.A. Mead and L.A. Conway, Reading, MA: Addison-Wesley, 1980, pp. 245–258.
6. I.E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, June 1989.
7. D.L. Dill, S.M. Nowick and R.F. Sproull, *Specification and Automatic Verification of Self-timed Queues*, Technical Report CSL-TR-89-387, Computer Systems Laboratory, Stanford University, Stanford, CA, 1989.
8. R.E. Miller, *Switching Theory*, New York: Wiley, 1965.
9. M.R. Greenstreet, T.E. Williams and J. Staunstrup, "Self-Timed Iteration," *VLSI '87: Proc. of the Int. Conf. on VLSI*, Vancouver, 1987.
10. W. Feller, *An Introduction to Probability Theory and Its Applications, Vol. 1*, New York: Wiley, 1968.
11. J. Kao, personal communication.
12. A.O. Allen, *Probability, Statistics, and Queuing Theory, with Computer Science Applications*, New York: Academic Press, 1978.
13. A.L. Fisher and H.T. Kung, "Synchronizing Large VLSI Processor Arrays," *IEEE Trans. on Computers*, vol. C-34, 1985, pp. 734–740.

Mark Greenstreet received the B.S.E.E. degree (with honors) from the California Institute of Technology, Pasadena, CA, in 1981.

From January 1982 through July 1985, Mr. Greenstreet was employed by ESL, Inc., Sunnyvale, CA, where he designed custom VLSI chips for signal processing applications. He was awarded two patents related to these designs. In August, 1985, Mr. Greenstreet joined the newly formed VLSI research group at Aarhus University, Aarhus, Denmark, where his research included parallel computation, VLSI design, and design verification. Since September 1987, Mr. Greenstreet has been in the Ph.D. program in computer science at Princeton University.



Kenneth Steiglitz received the B.E.E. (magna cum laude), M.E.E., and Eng.Sc.D. degrees from New York University, New York, NY, in 1959, 1960, and 1963, respectively.

Since September 1963 he has been at Princeton University, Princeton, NJ, where he is now Professor of Computer Science, teaching and conducting research on highly parallel architectures, optimization algorithms, and the foundations of computing. He is the author of *Introduction to Discrete Systems* (New York: Wiley, 1974), and co-author, with C.H. Papadimitriou, of *Combinatorial Optimization: Algorithms and Complexity* (Englewood Cliffs, NJ: Prentice Hall, 1982).

Dr. Steiglitz is a member of the VLSI Committee of the IEEE Signal Processing Society, is chairman of the Society's Technical Direction Committee, served two terms as member of their Administrative Committee, as member of the Digital Signal Processing Committee, and as Awards Chairman of that Society. He is an Associate Editor of the journal *Networks*, and is a former Associate Editor of the *Journal of the Association for Computing Machinery*. A member of Eta Kappa Nu, Tau Beta Pi, and Sigma Xi, he was elected Fellow of the IEEE in 1981, received the Technical Achievement Award of the Signal Processing Society in 1981, their Society Award in 1986, and the IEEE Centennial Medal in 1984.