Self-Timed Meshes Are Faster Than Synchronous

Peggy B.K. Pang Mark R. Greenstreet
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4
Canada
{ppang,mrg}@cs.ubc.ca

Abstract

This paper shows that self-timed meshes can achieve linear speed-up. The per-processor performance of a mesh is the average number of operations per processor per unit time. For synchronous processors, it has been shown that the per-processor performance of a mesh goes to zero as the size of the mesh goes to infinity. This paper shows that for self-timed meshes, the per-processor performance can be bounded below by a positive constant. Thus, self-timed meshes are asymptotically faster than synchronous ones. Furthermore, simulation and analytic results are used to show that analysis based solely on average case times can be optimistic and lead to poor design decisions.

1. Introduction

Two oft-mentioned advantages of self-timed designs are freedom from clock skew and achieving average-case performance in data dependent computations. Because self-timed designs do not have clocks, it is clear that they do not have clock skew. On the other hand, the performance of one part of a self-timed system can affect the performance of another part when one waits for a handshake from another. Through a sequence of such handshakes, two components that are widely separated in a design can influence each other's performance. To the best of our knowledge, only linear pipelines have been studied to determine how system size affects the performance of a self-timed system [6]. Likewise, most performance analysis for self-timed systems has been based on fixed processor times leaving claims for average case performance unsubstantiated. This paper analyses the performance of self-timed meshes with random processing times.

We measure performance as the average number of operations per processor per unit of time. A "processor" in this context represents a handshaking component, not necessarily a CPU. Thus, our analysis applies to large arrays of handshaking components, a model that is general enough to apply to a wide variety of self-timed designs. We do not consider performance issues that are specific to an application such as whether or not there is enough parallelism in the application to effectively utilise a large array.

Our analysis provides a basis for comparing the scalability of synchronous and self-timed designs. For this comparison, we consider "pure" synchronous designs. In such a design, all communication is controlled by a global clock. This clock is distributed to all components using a tree network with the clock generator at the root and components controlled by the clock at the leaves. The tree topology reflects the choice to determine all timing from a single clock – if there were multiple paths from the clock generator to a component, an additional timing component would be required to combine the clock signals from various paths to produce a single clock signal for the component. If two components communicate, there must be an interval within each clock cycle during which the sender can change its output and another interval during which the receiver can sample its input. For reliable communication these intervals must be disjoint. The clock-skew is the maximum uncertainty in the relative arrival times of the clock signal at two communicating components. To guarantee that the sending and sampling intervals are disjoint, the clock period must be greater than the worst-case skew.

The performance of large synchronous designs has been well studied. Fisher and Kung [5] analysed the asymptotic performance of synchronous processor arrays with this assumption. For meshes, they showed that worst-case skew grows with the square-root of the number of processors. Dikaiakos and Steiglitz [3] performed a similar analysis for expected skew and showed a bound of $n^{1/4} \log n$. Assuming that the clock period must be greater than the skew, per processor performance becomes arbitrarily small as the number of processors in the mesh increases. These theoretical results are matched by practical experience. There are few examples of globally synchronous designs with a large number of processors: most large scale parallel processors are a collection of synchronous processors with asynchronous interfaces to a communication fabric.

Given the limitations of pure synchronous designs, there have been several proposals of hybrid approaches. In general, these designs propose some kind of handshaking network that provides tightly correlated clock signals in the processor network. A simple example of this approach is Chapiro's proposal for clock-stretching [2]. When a receiver using this scheme is ready for input, it stops its own clock and waits for a "data-ready" signal from the transmitter. If a data-transfer is performed on each clock cycle, clock stretching reduces to four-phase, self-timed signaling with bundled completion [12].

Nowatzyk [10] and Pratt [11] have both proposed using arrays of phase locked loops to generate a global clock signal for large, multi-processor arrays. In such a design, care must be taken to avoid what Pratt refers to as "mode-lock" where a processor has N neighbours whose phases form an equilateral N-gon. Nowatzyck uses the limiting behaviour of a phase-frequency detector to introduce a non-linearity that breaks the symmetry of a mode-lock. Pratt solves the mode-lock problem by using a phase detector with negative (i.e. unstable) slope for phase differences greater than $\pi/2$ radians. We note that handshake circuits can be viewed as an array of coupled oscillators, and the choice of whether to view an array of phase locked loops as a synchronous or self-timed design is largely a matter of taste. The analyses of Nowatzyk and Pratt are specific to particular designs. Our analysis is for general handshaking circuits and is independent of the particular implementation.

Su [13] noted that a self-timed mesh can be used to distribute a clock signal. He reports on the performance of his simulator when simulating such a network, but he does not describe the performance of the network itself nor does he describe the timing model used for his simulations. Our work shows that such a clock network can operate as desired – the clock frequency can remain roughly constant as the array grows to arbitrary size.

Thiele [15] analysed the performance of self-timed processor arrays. Thiele's analysis assumes fixed processor times and uses Lawler's algorithm [9] to find the worst-case cycle of handshakes in the system. Burns [1] also used Lawler's algorithm to analyse the performance of self-timed systems. Burns's method uses weighted averages to compute expected performance when there are several possible paths of execution. This works well for processor designs where different timings can be associated with different instructions and the analysis can consider a typical instruction mix. For our current work, the number of possible cycles grows exponentially with the size of the mesh which precludes using Burns's approach to derive asymptotic results. Greenstreet and Steiglitz [6] analysed self-timed pipelines with exponential processing times. They showed a transformation of the pipeline performance problem into a queueing network problem. In the one-dimensional case that they considered, this produces a product-form queueing network, and they were able to give exact results. We use a similar approach to analyse two-dimensional meshes.

This paper presents handshake protocols for self-timed meshes in section 2. Both three- and four-connected meshes are described. In the four-connected mesh, each node receives data from its south and west neighbours and acknowledgements from its north and east neighbours. In the three-connected mesh, the north and east acknowledgements are combined into a single acknowledgement from the north-east neighbour. This modification allows the mesh to be viewed as a queueing network, and we show that the performance bounds for the three-connected mesh provide lower bounds for the more familiar four-connected mesh. The queueing network model is analysed in section 3 where we show that the average processor utilisation of the three-connected mesh is at least 1/12. This establishes that self-timed meshes indeed scale better than synchronous designs. The results in section 2 and 3 are based on exponential distribution of processing times. Section 4 presents Monte-Carlo simulations where other distributions are considered along with examining the impact of using buffers between processors.

2. Self-Timed Meshes

One-dimensional, self-timed pipelines have been well-described in the literature (e.g. [6, 14, 16]. For the sake of comparison with two-dimensional meshes, we state the rules that govern the operation of a Muller style pipeline. It is sufficient to consider only the state of the processor in the handshake protocol. Each processor has a predecessor and a successor. When the processor and its predecessor are in opposite states (i.e. the predecessor is offering new data), and the processor and its successor are in the same state (i.e. the successor has received the latest value), then the processor may move to the state of the predecessor. We equate this move with performing a computation and assume that the time between enabling and firing a processor is a random variable. If the pipeline is a ring, then at any instant, the ring can be partitioned into maximal segments of processors in the same state. It was shown in [6] that the number of such segments is an invariant maintained by the ring.

2.1. A four-connected mesh

Figure 1 shows a self-timed mesh where each processor communicates with its north, south, east, and west neigh-

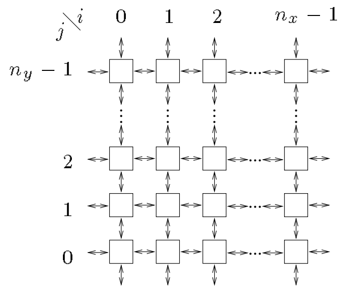

Figure 1. A four-connected mesh

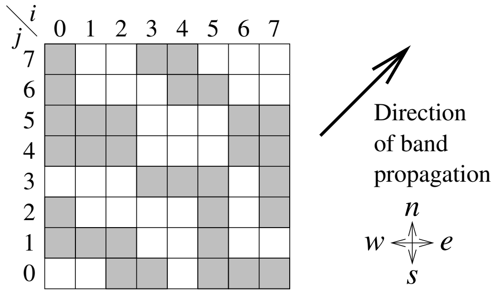

Figure 2. Bands on the four-connected mesh

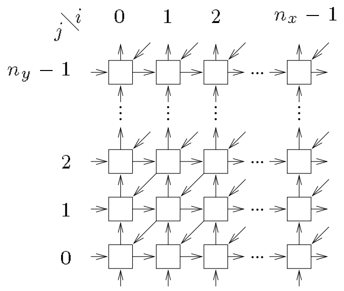

Figure 3. A three-connected mesh

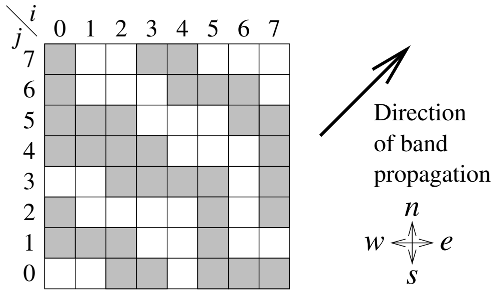

Figure 4. Bands on the three-connected mesh

bours. To avoid boundary conditions, we embed the mesh on a torus: the east neighbour of the right-most processor in a row is the left-most processor of the same row, and the north neighbour of the top processor in a column is the bottom processor in the same column. The basic principles of the Muller pipeline can be used to define a handshake protocol for the mesh: when a processor is in the same state as its north and east neighbours and in the opposite state as its south and west neighbours, it is enabled to change state. As with the original Muller pipeline, this handshake protocol is speed-independent: once a processor is enabled to fire, it remains enabled with unchanging inputs until it fires.

If we consider north-south, east-west, and northwest-southeast pairs in the same state as contiguous, then we can identify maximal contiguous regions of processors in the same state. As figure 2 shows, these contiguous regions form bands around the torus that correspond to the segments of one-dimensional pipelines. The number of bands is an invariant maintained by the mesh, and bands propagate to the northeast. For simplicity, we only consider configurations where each band wraps once around the torus in the north-south direction and once around the mesh in the east-west direction. This property is also an invariant of the mesh.

This protocol allows bands to be composed of pieces that only meet at the corners of diagonally adjacent processors, for example, processors $p(2,4)$ and $p(3,3)$ in figure 1 (where $p(i,j)$ denotes the processor at location (i,j)). To determine a lower bound on the performance of such a mesh, we found it helpful to consider a modified protocol that avoided these pinched bands. This protocol corresponds to a three-connected mesh as described in the next section.

2.2. A three-connected mesh

Figure 3 shows a three-connected mesh. A processor is enabled if it is in the opposite state as its south and west neighbours and in the same state as its northeast neighbour. We assume that the times between when a processor becomes enabled and when it fires are independent, exponentially distributed random variables. Without loss of generality, we assume that the mean firing time is one. It is straightforward to show that an invariant of this protocol is that if $p(i,j)$ is the left-most processor in band β and row j, and $p(i',j+1)$ is the right-most processor in row $j+1$ of the same band, then $i \leq i'$. This means that, unlike the four-connected mesh, no band of the three-connected mesh has pieces that meet only at a corner (see figure 4). A related invariant is that whenever a processor is in the same state as its northeast neighbour, then it is in the same state as its north and east neighbours as well. A direct consequence of the second invariant is that the three-connected mesh is speed-independent.

Proposition 1 *The performance of a four-connected mesh is greater than or equal to the performance of a three-connected mesh of the same dimension, number of bands, and distribution of processor firing times.*

Proof: Executions of a three-connected mesh to can be mapped to executions of a four-connected mesh such that each operation in the four-connected mesh occurs no later than it did in the three-connected mesh. This mapping is possible because in any mesh configuration, the set of processors enabled according to the three-connected protocol is a subset of those enabled according to the four-connected protocol. Thus, operations in the four-connected mesh become enabled no later than their counterparts in the three-connected mesh. □

An immediate consequence of proposition 1 is that any lower bound for the performance of the three-connected mesh is also a lower bound for the four-connected mesh.

The three-connected mesh has a tractable queueing network model. Let n_x and n_y be the circumference of the torus in the east-west and north-south directions respectively. Let n_B be the number of bands. Because the handshake protocol has two states, n_B must be even. Let β be a band, and let j be the index of a row in the mesh, and let $p(i,j)$ be the left-most processor in band β and row j. We identify two queue-like structures rooted at $p(i,j)$:

The southeast queue: Let $p(i_{se}, j-1)$ be the left-most processor in row $j-1$ and band β. We note that if $i_{se} - i = 0$, then $p(i,j)$ is not enabled because it is in the same state as its south neighbour. We say that $i_{se} - i$ is the number of processors waiting in the southeast queue for row j of band β. If $p(i,j)$ is enabled, then firing it corresponds to a departure from this queue. Likewise, firing $p(i_{se}, j-1)$ constitutes an arrival.

The northeast queue: Let $p(i_{ne}, j+1)$ be the right-most processor in row $j+1$ and band β. As noted above, $i_{ne} \geq i$. If $i_{ne} - i = 0$, then $p(i,j)$ is not enabled. Accordingly, $i_{ne} - i$ is the number of processors waiting in the northeast queue for row j of band β. Firing $p(i,j)$ causes a departure from this queue, and firing $p(i_{ne}+1, j)$ causes an arrival.

In figure 3, $p(2,3)$ is the left-most processor of a band in row 3. The southeast queue for $p(2,3)$ extends to $p(5,2)$; there are three processors waiting in this queue. The northeast queue for $p(2,3)$ extends to $p(3,4)$; there is one processor waiting in this queue. Thus, $p(2,3)$ is enabled to fire. After $p(2,3)$ fires, $p(3,3)$ will become the left-most processor for this band in row 3. Its northeast queue will be empty until $p(4,4)$ fires. The average length of the southeast queues is n_x/n_y, and the average length of the northeast queues is $(n_x/n_B) - (n_x/n_y) - 1$.

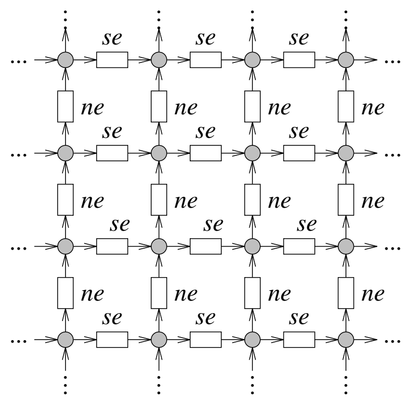

Figure 5. A queueing network model for the three-connected mesh

Figure 5 shows the queueing network corresponding to a three-connected mesh. Queues are labeled ne or se according to whether they are northeast or southeast queues of the mesh. At each junction between queues, there is a fork/join node, indicated by a filled circle. When the queues to the left and below a fork/join node are non-empty, the node may remove one element from each of these queues and insert one element into the queue above it and one element into the queue to its right. Service times of the fork/join nodes are independent, exponentially distributed random variables with mean one. Operations of the fork/join nodes correspond to processor actions in the original mesh.

A few comparisons of the queueing network and the three-connected mesh are in order. The interconnection of the queueing network forms a twisted torus. The twist does not affect our analysis which depends only on local properties of the network. Fork/join operations move elements between queues, and the total number of elements in either type of queue is fixed. Insert and remove operations displace the ends of the queues on the original three-connected mesh. Each southeast queue crawls around the mesh a the southeast direction, and likewise for northeast queues.

The analysis in the next section is simplified if the average queue lengths for southeast and northeast queues are equal and if initially each queue holds exactly this many elements. This is achieved, for example, if n_x is a multiple of six, $n_y = n_x$, and $n_B = n_x/3$ in which case the average length for all queues is one. If the mesh is started in the state where

$$p(i,j).v \quad = \quad ((i-j) \bmod 6) < 3 \qquad (1)$$

then the corresponding queuing network will start with one element in each queue (we write $p(i,j).v$ to denote the value held by processor $p(i,j)$).

Proposition 2 *Let M be a three-connected mesh with n_x a multiple of six, $n_y = n_x$, and $n_B = n_x/3$ with processors initialised as in equation 1. Let Q be a mesh connected queueing network with $n_x^2/3$ queues of each type as shown in figure 5 where each queue initially holds one element. If the average rate of departures from southeast queues is μ, then, the per processor performance is $\mu/3$.*

Proof: Each operation of a fork/join node corresponds to an action of some processor in the three-connected mesh. There are $n_x^2/3$ fork/join nodes in the queueing network and n_x^2 processors in the three-connected mesh. Thus, the per processor performance is $\mu/3$ as claimed. □

3. Arrays of Queues

This section presents a lower bound for μ that is independent of the size of the network. Our analysis is by induction on the structure of the network. The base case is a square of four queues where three of the four fork/join nodes are replaced by simple exponential servers. This replacement makes the analysis tractable, and we show that the simplified model provides a lower bound for performance. The induction step adds a square of nodes to the network. Again we show that the construction is conservative. Finally, we join the edges of the network to form the twisted-torus of the actual queueing network. This too is a conservative construction giving us a lower bound for the performance of the queueing network.

3.1. Probability primer

We first review some concepts from probability that will be used in the analysis. If g is a Boolean expression, $P\{g\}$ denotes the probability that g holds. Let α be a random variable. The expected value (i.e. mean) of α is denoted by $E[\alpha]$. The distribution function of α is written $\mathrm{F}[\alpha]$ where

$$\mathrm{F}[\alpha](u) \quad = \quad P\{\alpha \leq u\} \qquad (2)$$

If α_1 and α_2 are random variables such that for all u, $\mathrm{F}[\alpha_1](u) \leq \mathrm{F}[\alpha_2](u)$, then α_1 *stochastically dominates* α_2, and we write $\alpha_1 \geq_{\mathrm{st}} \alpha_2$. If $\alpha_1 \geq_{\mathrm{st}} \alpha_2$, then $E[\alpha_1] \geq E[\alpha_2]$. Stochastic dominance extends readily to sequences. If $a(0), a(1), \ldots$ and $b(0), b(1), \ldots$ are sequences of random variables such that for all $i \geq 0$, $a(i) \geq_{\mathrm{st}} b(i)$, then a stochastically dominates b, and we write $a \geq_{\mathrm{st}} b$.

The joint distribution function for $\alpha_1, \alpha_2, \ldots, \alpha_m$ is $\mathrm{F}[\alpha_1, \ldots, \alpha_m]$ with

$$\begin{aligned} & \mathrm{F}[\alpha_1, \ldots, \alpha_m](u_1, \ldots, u_m) \\ = \quad & P\{(\alpha_1 < u_1) \wedge \ldots \wedge (\alpha_m < u_m)\} \end{aligned} \qquad (3)$$

Variables $\alpha_1 \ldots \alpha_m$ are independent iff

$$\mathrm{F}[\alpha_1, \ldots, \alpha_m](u_1, \ldots, u_m) \quad = \quad \prod_{i=1}^{m} \mathrm{F}[\alpha_i](u_i) \qquad (4)$$

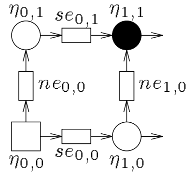

Figure 6. A simple, square queuing network

If α is an exponentially distributed random variable with mean m, then $\mathrm{F}[\alpha](u) = 1 - e^{-u/m}$ for $u \geq 0$ and zero otherwise. We write x_m to denote a exponentially distributed random variable that is independent of any other variables in the analysis. A Poisson process with rate λ is a sequence, $a_1, a_2, \ldots$, such that for i greater than zero, the random variable $a_{i+1} - a_i$ is exponentially distributed with mean $1/\lambda$, and these variables are independent. We write p_m to denote a Poisson process with mean m whose interarrival times are independent of any other variables in the analysis.

3.2 Analysing the queuing network

We first consider the simple network shown in figure 6. The filled circle is a fork/join node whose processing times are independent, exponentially distributed random variables with mean one. The empty circles are server nodes whose processing times are independent, exponentially distributed random variables with mean two. The rectangular components are queues that initially hold one element, and the square component is a Poisson source with rate $1/4$. We use the label of a component to refer to the sequence of random variables corresponding to the times of output events of the component.

Lemma 3 *Given a queuing network as depicted in figure 6, the sequence of output events for each node (source, server, or fork/join) is stochastically dominated by a Poisson process with rate $1/4$. Furthermore, the time between the arrival of an element at the input of the fork/join node from queue $se_{0,1}$ and the subsequent operation of the fork/join is stochastically dominated by a exponential variable with mean two. Likewise for arrivals from queue $ne_{1,0}$.*

Proof: By the symmetry of the network, the probability that an element from queue $se_{0,1}$ arrives at the fork/join node, $\alpha_{1,1}$, before the corresponding element has arrived from $ne_{1,0}$ is at most one half. If the element from queue $se_{0,1}$ arrives before the element from $ne_{1,0}$, then the element from $ne_{1,0}$ must be waiting at the input of server $\eta_{1,0}$ because both elements were output by source $\eta_{0,0}$ at the same time. Thus, if the element from queue $se_{0,1}$ arrives first, it will wait for the exponential service time with mean 2 of server $\eta_{1,0}$ and then wait for the exponential service time with mean 1 of

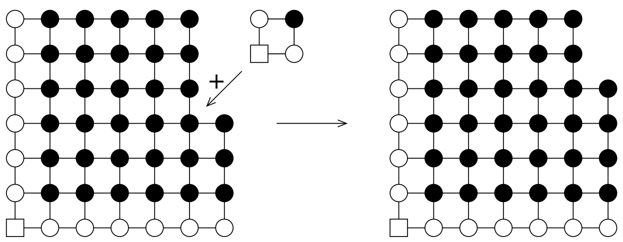

Figure 7. Constructing a larger network

server $\eta_{1,1}$. As noted above, this happens with probability at most one half. The other possibility is that the element from queue $se_{0,1}$ arrives at the same time or later than the element from queue $ne_{1,0}$. In this case, the element from queue $se_{0,1}$ waits only for the exponential service time with mean 1 of server $\eta_{1,1}$. From these observations, we can show that the waiting time for the element from queue $se_{0,1}$ is stochastically dominated by an exponential process with mean two. This establishes the claim about service times. Details are provided in appendix A.2. Noting that the source is Poisson with rate 1/4 and that the servers are stochastically dominated by exponential servers with mean 2, standard queueing theory shows that the output processes for the servers are stochastically dominated by a Poisson process with rate 1/4. Again, details are provided in appendix A.2. □

To extend the results for the square to larger networks, we construct larger networks from a simple square merging a single square at a time as shown in figure 7. In the merging process, fork nodes are promoted to fork/join nodes and sources are eliminated in the obvious manner. Details of the construction are given in appendix A.3. The key result is stated below:

Lemma 4 *Let n_x and n_y be natural numbers. For $0 \leq i < n_x$ and $0 \leq j < n_y$, let $\alpha_{i,j}$ be a node where $\alpha_{0,0}$ is a Poisson source with rate $1/4$; for i and j greater than zero, $\alpha_{0,j}$ and $\alpha_{i,0}$ are exponential servers with mean service time two, and $\alpha_{i,j}$ is a fork/join node with mean service time one; and the nodes are connected in a rectangular array by queues which each initially hold one element.*

The sequence of output events for each node (source, server, or fork/join) is stochastically dominated by a Poisson process with rate $1/4$. The time between the arrival of an element at a fork/join node and the subsequent operation by the fork/join is stochastically dominated by a exponential random variable with mean two.

Proof: If n_x or n_y is zero, then the network is empty, and the claims are trivially satisfied. If n_x or n_y is one, then the network is a chain of queues for which the analysis is straightforward. We focus on the case where n_x and n_y are both at least two. Any such mesh can be constructed starting with a square and merging successive squares on the right or top boundaries of the existing network. Accordingly, we induct over the size of the network. For each merge operation in the construction, the performance of nodes in the original network is unchanged, and the performance of nodes in the square added to the network is unchanged or improved. Details are given in appendix A.3 □

Finally, we need to identify opposite edges of the array to form the twisted torus. This produces a closed network, promoting the remaining Poisson source and the remaining exponential servers to fork/join nodes. The challenge with this operation is that we identify two edges of a large network, rather than adding a simple square. The times of operations on opposite edges of the rectangular network are not independent random variables. However, an induction argument over the time of the operations shows that the results for the array also apply to the twisted-torus.

Proposition 5 *Given integers n_x and n_y, let Q be a queuing network composed of an n_x by n_y array of fork/join nodes connected by queues in a twisted-torus topology. The sequence of output events for each node is stochastically dominated by a Poisson process with rate $1/4$.*

Proof: See appendix A.4.
We can now prove the main claim of this paper.

Proposition 6 *Let n be a positive multiple of three. Let M be a three-connected or four-connected mesh with $n_x = n_y = n$ and $n_B = n/3$. If the mesh is started in the state that satisfies equation 1, then the per-processor performance of the mesh is at least $1/12$.*

Proof: The claim for a three-connected mesh follows directly from propositions 2 and 5. The result extends to a four-connected mesh by proposition 1.

These bounds are conservative; they underestimate the performance of the mesh. In the next section, we present performance measurements from simulations of particular meshes.

4. Simulation Results

Figure 8 shows results from Monte-Carlo simulations of three- and four-connected meshes. In both plots, u denotes processor utilisation, the average number of operations per processor per unit time. The left plot shows the relation between n_x/n_B (number of processors in a row or column per band) and performance for a 72 by 72 mesh. The highest performance for the three-connected mesh is 0.127 which is about 50% higher than the lower bound derived in the previous section. The lower bound is conservative, largely because the analysis for the bound overestimates the probability that an element at a fork/join node waits for an arrival

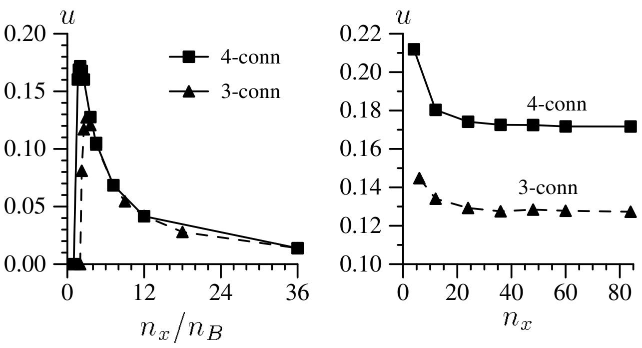

Figure 8. Processor utilisation vs. average band width and mesh size

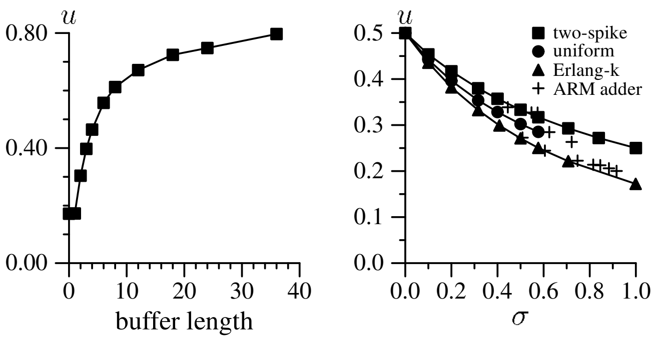

Figure 9. Effects of buffering and processing time distribution on processor utilisation

from the other queue. The highest performance occurs when $n_x/n_B = 3$, which is when the average length for the northeast and southeast queues are equal at one. The highest performance for the four-connected mesh is 0.172 which occurs when $n_x/n_B = 2$. This is the point where the four inputs to the processor node are equally likely to be last.

The right plot of figure 8 shows the dependence of processor performance on the mesh size. Optimal values of n_x/n_B were used for the simulations: for the three-connected mesh, $n_x/n_B = 3$, and for the four-connected mesh, $n_x/n_B = 2$. For both meshes, performance rapidly approaches the limit. Small meshes perform somewhat better than large meshes because the timing correlations across a small mesh are stronger than those across a large one. These correlations contribute to higher performance.

The left plot of figure 9 shows the performance of a four-connected mesh with buffers between processors. Each connection between processors is buffered by a FIFO queue using the usual Muller protocol. The times for an active stage to perform an operation are exponential random variables with mean 0.1. If throughput were determined only on average-case time, then Thiele's analysis [15] shows that optimal performance of one operation every $1/1.1$ time units would be achieved with two-stage FIFO buffers. As shown in figure 9, the variance in processing time is significant in overall performance, and higher performance is achieved with larger buffers. Of course, using larger buffers increases the latency of the pipeline, and the designer must address the trade-offs of throughput and latency. If a designer only considered average case processing times, there would be no reason to use buffers with more than two stages. Such a design is likely to have less than optimal performance for real processing time distributions.

The right plot of figure 9 shows the performance of a 72 by 72 mesh for various distributions of processing times. Each distribution has a mean of one, and different distributions in the same family are characterised by their standard deviations, σ. A random variable with the two-spike distribution with standard deviation σ takes on value $1 - \sigma$ with probability 0.5 and value $1 + \sigma$ with probability 0.5. As negative processing times are non-sensical, we require $0 \leq \sigma \leq 1$. A uniformly distributed random variable with standard distribution σ is uniformly distributed in the interval $[1 - \sqrt{3}\sigma, 1 + \sqrt{3}\sigma]$. To avoid negative processing times, σ must be between 0 and $1/\sqrt{3}$. The Erlang-k distribution is obtained as the sum of k exponentially distributed random variables, and $\sigma = 1/\sqrt{k}$ (see [4]). For large values of k, the central-limit theorem shows that this distribution is close to a normal distribution with the same standard deviation; however, a random variable with an Erlang-k distribution always takes on non-negative values. Finally, the ARM-adder distribution was obtained from trace data for the ARM microprocessor (see [17]). We obtained traces for three programs, with each trace providing carry chain length distributions for branches, addresses calculations, general arithmetic, and totals. We assumed delay proportional to carry chain length, and obtained twelve distributions for our simulations (four from each program trace).

From figure 9, it is apparent that performance is determined largely by the variance of the processing time distribution. This is consistent with the experience of the performance analysis community that many properties of queueing networks can be estimated from the first and second moments of service time distributions (see [8]). The performance for the trace data is similar to that for the artificial distributions, suggesting that reasonable estimates can be made for real designs once the mean and variance of the processing time distribution is known. We also noted that the mean carry-chain length from the trace data varied from 5.0 to 21.0, with the total values for each trace between 9 and 11. This is much larger than the values for an adder with uniformly distributed inputs. Thus, although logarithmic expected time for addition is often mentioned as an advantage of self-timed design, we don't expect that this "advantage" will be realised in real processor designs.

The data for the uniform and twin-spike distributions fit simple formulas. In particular, processor utilisation with uniform processor times fits

$$u_{\text{uniform}}(\sigma) \quad = \quad \frac{1}{2(1+1.31\sigma)} \tag{5}$$

Processor utilisation with the twin-spike distribution fits

$$u_{\text{twin-spike}}(\sigma) \quad = \quad \frac{1}{2(1+\sigma)} \tag{6}$$

We note that $1 + \sigma$ corresponds to the maximum delay of the twin-spike distribution. This means that the expected performance corresponds exactly to worst-case processing time! We note that each operation in the process dependency graph has four successors. Each of these successors has probability of 0.5 of taking place after the maximum delay. Thus, the probability that at least one of these operations takes place after the maximum delay is $15/16$, and the expected number of these operations that take place after the maximum delay is 2. We suspect that this ensures that with high probability, there will be chains of operations in the process dependency graph where almost all operations take the maximum delay. At the time of writing, we don't have a formal proof of this conjecture.

The observations about the twin-spike design have implications for self-timed design. For example, consider a design where the time for a component to perform an operation can take on one of two values depending on the value of its operands; in other words, it has a fast-mode for some operands and a slow mode for others. If long and short operations occur with equal probability, then a mesh of such components will operate at the rate corresponding to all operations being slow. Thus, adding hardware to distinguish between slow and fast operands only increases the complexity of the design and cannot improve performance. Our simulations indicate that with a bimodal processing time distribution, at least 75% of the operations must be fast before the performance is not determined solely by the time for the slow operation.

5. Conclusions

Two oft-mentioned advantages of self-timed designs are freedom from clock skew and achieving average-case performance in data dependent computations. We have demonstrated that self-timed meshes can achieve a per-processor throughput that is independent of the size of the array. In contrast, the clock period for a synchronous design must grow asymptotically with the diameter of the network. In this asymptotic sense, self-timed arrays are faster than their synchronous counterparts.

Our analysis was based on a queueing network model for a simple array assuming exponentially distributed processing times with mean one. Using this model, we showed that the average number of operations per processor per unit time is at least $1/12$. This is a lower bound, and simulations suggest that more accurate estimates are $1/7.85$ and $1/5.82$ for three-connected and four-connected meshes respectively.

Although the asymptotic performance of a self-timed mesh is within a constant factor of the average case performance for its components, the variance in processing time contributes significantly to the overall performance. This suggests that performance analysis techniques should be developed that take into account the variance of processing times. Based on our simulations, the sum of the mean delay and the standard deviation provides a much better estimate of performance than the mean delay alone. We showed that the effects of processing time variance can strongly affect design decisions regarding the size of buffers to use between processing elements. For bimodal processing time distributions, performance can be completely determined by the time required for the slow operation. Unless the variance of processing time is considered, claims of "average-case" performance can be seriously misleading.

Acknowledgements

We have enjoyed interactions with many excellent colleagues. Ivan Sutherland revived our interest in the performance of self-timed meshes with his own enthusiasm for self-timed pipelines and their variations. We benefited from Nick Pippenger's insightful comments about stochastic inequalities. Peter Beerel provided us with trace data for the ARM that allowed us to simulate meshes with "real-world" processing time distributions.

References

[1] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, Computer Science Department, California Institute of Technology, Pasadena, CA, Jan. 1991. Tech. Report Caltech-CS-TR-91-01.

[2] D. M. Chapiro. *Globally-Asynchronous, Locally-Synchronous Systems*. PhD thesis, Department of Computer Science, Stanford University, Oct. 1984. Tech. Report STAN-CS-84-1026.

[3] M. D. Dikaiakos and K. Steiglitz. Comparison of tree and straight-line clocking for long systolic arrays. *Journal of VLSI Signal Processing*, 3(4), 1991.

[4] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John-Wiley and Sons, 1950.

[5] A. L. Fisher and H. Kung. Synchronizing large VLSI processor arrays. *IEEE Transactions on Computers*, C-34(8):734–740, Aug. 1985.

[6] M. R. Greenstreet and K. Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI and Signal Processing*, 2(3):139–148, Nov. 1990.
[7] L. Kleinrock. *Queueing Systems, volume 1: theory*. John Wiley and Sons, 1975.
[8] L. Kleinrock. *Queueing Systems, volume 2: applications*. John Wiley and Sons, 1976.
[9] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, New York, 1976.
[10] A. Nowatzyk. *A Communication Architecture for Multiprocessor Networks*. PhD thesis, School of Computer Science, Carnegie Mellon University, Dec. 1989. See chapter 3.3. Tech. report CMU-CS-89-181.
[11] G. A. Pratt and J. Nguyen. Distributed synchronous clocking. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):314–328, Mar. 1995.
[12] C. L. Seitz. System timing. In *Introduction to VLSI Systems* (Carver Mead and Lynn Conway), chapter 7, pages 218–262. Addison Wesley, 1979.
[13] W.-K. Su. *Reactive-Process Programming and Distributed Discrete Event-Simulation*. PhD thesis, Computer Science Dept., California Institute of Technology, Oct. 1989. See chapter 8.1. Tech. report Caltech-CS-TR-89-11.
[14] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. Turing Award lecture.
[15] L. Thiele. On the analysis and optimization of self-timed processor arrays. *INTEGRATION*, 12(2):167–187, Dec. 1991.
[16] T. E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, May 1991.
[17] K. Y. Yun, A. E. Dooply, et al. The design and verification of a high-performance, low-control-overhead asynchronous differential equation solver. In *Proceedings of the 1997 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 1997.

A Proofs

A.1 Computation graphs for queueing networks

The queueing networks that we analyse have simple computation graphs. These graphs have a vertex corresponding to each operation for each node. If the operation for vertex v_j consumes an element output by the operation for vertex v_i, then there is an edge from v_i to v_j. We add a node, v_0, with edges to all vertices for operations that are enabled in the initial state. This computation graph is rooted (at v_0), directed, and acyclic. Because we consider meshes with no final state, our computation graphs are infinite. We associate a non-negative random variable with each vertex which is the time that the corresponding node takes to perform this operation after all of its inputs are available. We assume that these random variables are independent. The vertex v_0 represents the "initial operation," and we define this to occur at time zero. The time at which a node performs on operation is the maximum of the time for any of its inputs plus the time for the operation.

The max and sum operators are strictly increasing in their arguments. This gives rise to the following property that we use throughout our analysis. The proof is omitted for brevity.

Property 1 *If two graphs have the same structure, and the distributions of the random variables for delay times of the first graph stochastically dominate those of the second, then the distributions of the times at which operations occur in the first graphs stochastically dominate those of the second.*

A.2 Proof of lemma 3

Recall that the lemma pertains to the queueing network shown in figure 6. The output of source component, $\alpha_{0,0}$ is Poisson with rate $1/4$, so the claims are trivially satisfied for this component. The sequence of components $\alpha_{0,0} \rightarrow se_{0,0} \rightarrow \alpha_{1,0}$ forms a $M/M/1$ queue with arrival rate $1/4$ and mean service time 2. This is a well-studied system (see [7]). In steady state, the average queue length is one, and the departure process, $\alpha_{1,0}$, is Poisson with rate $1/4$. The condition that the queue initially holds one element guarantees that the transient behaviour of the output process is stochastically dominated by the Poisson process. Thus, the outputs of node $\alpha_{1,0}$ satisfy the claims. A symmetric argument applies for the outputs of node $\alpha_{0,1}$.

Now, consider node $\alpha_{1,1}$ at the moment when an element arrives from queue $se_{0,1}$. Call this element e_{se}. Let τ_{se} be the time of this arrival; note that this time can be determined either by the time at which e_{se} was inserted into queue $se_{0,1}$ by node $\alpha_{0,1}$, or by the time at the previous element was removed from queue $se_{0,1}$ by node $\alpha_{1,1}$. Let $\tau_{1,1}$ be the time at which element e_{se} is processed by node $\alpha_{1,1}$, and let $\delta_{se} = \tau_{1,1} - \tau_{se}$. Let e_{ne} be the corresponding element from queue $ne_{1,0}$, and let τ_{ne} denote its time of arrival at the input of $\alpha_{1,1}$. By the symmetry of the network, $P\{\tau_{se} < \tau_{ne}\} = P\{\tau_{se} > \tau_{ne}\}$. Thus, $P\{\tau_{se} < \tau_{ne}\} \leq \frac{1}{2}$. Accordingly,

$$F[\delta_{se}] \quad \leq_{\text{st}} \quad \tfrac{1}{2}F[\delta_{se}|\tau_{se} < \tau_{ne}] + \tfrac{1}{2}F[\delta_{se}|\tau_{se} \geq \tau_{ne}]$$

We first consider the case when $\tau_{se} < \tau_{ne}$. Note that e_{se} and e_{ne} were output by the same operation of source node $\alpha_{0,0}$. Furthermore, their predecessors have already been consumed by fork/join node $\alpha_{1,1}$. Accordingly, element e_{ne} must be at the input of node $\alpha_{1,0}$. In this case, e_{se} will wait at the input of $\alpha_{1,0}$ for exponential mean two time waiting for e_{ne} to arrive, and an additional exponential mean one time waiting for $\alpha_{1,1}$ to perform its operation. This yields:

$$F[\delta_{se}|\tau_{se} < \tau_{ne}](u) \quad = \quad 2(1 - e^{-u/2}) - (1 - e^{-u})$$

If $\tau_{se} \geq \tau_{ne}$, then δ_{se} is exponentially distributed with mean one:

$$F[\delta_{se}|\tau_{se} \geq \tau_{ne}](u) \quad = \quad 1 - e^{-u}$$

Combining these observations, we obtain

$$F[\delta_{se}](u) \quad \leq_{\text{st}} \quad 1 - e^{-u/2}$$

which means that δ_{se} is stochastically dominated by an exponential variable with mean two, as claimed. A symmetric argument applies for δ_{ne}.

Finally, consider the queueing system composed of components $\alpha_{0,1} \to se_{0,1} \to \alpha_{1,1}$. Analysis similar to that for $\alpha_{0,0} \to se_{0,0} \to \alpha_{1,0}$ and property 1 show that the outputs of $\alpha_{1,1}$ are dominated by a Poisson process with mean two. □.

A.3 Proof of lemma 4

We consider networks that have n_x columns and n_y rows of nodes connected by queues. We assume that n_x and n_y are both at least two. The first two columns have exactly n_y nodes, and the number of nodes in a column is a non-increasing function of the column index. Likewise, the first two rows have exactly n_x nodes, and the number of nodes in a row is is a non-increasing function of the row index. Any rectangular mesh can be constructed by starting with a simple square and merging additional squares one at a time as shown in figure 7 such that each intermediate network satisfies these constraints. In the merging process, fork nodes are promoted to fork/join nodes and Poisson sources are eliminated in the obvious manner.

Let Q be a network as described above, and consider the addition of a square to a column. The argument when adding a square to a row is analogous. There are two cases to consider: either the square is added at the bottom of a new column, or the square is added at the top of an existing column of Q. In either case, no inputs of nodes of Q are modified, so the performance of nodes of Q is unchanged.

Consider the case when the node is added at the bottom of a new column. The nodes on the left edge of the square are replaced by nodes of Q. Therefore, the claims are satisfied for these nodes. The node in the lower-right corner of the square had received its inputs through a queue from a Poisson source. After the merge, this node receives its inputs from the server node of Q. The service times of the server node are stochastically dominated by the service times of a Poisson source. By property 1, output times of the the lower-right node after the merge are dominated by those of the node from before the merge. By lemma 3, the lower-right node of the square satisfies the requirements of this lemma. Therefore, the node after the merge does as well. A similar argument applies for the upper-right node (the fork/join node).

The case when the node is added on top of an existing column is similar. In this case, the lower-right node of the square is replaced by a node from Q; therefore it satisfies the claims of this lemma. □

A.4 Proof of property 5

We "sew" opposite edges of the array together using queues that initially hold one element.

Let α_{dst} be the Poisson source of the array, and let α_{src} be the row-predecessor of this node in the torus. The first stitch of our construction is a queue with input α_{src} and output α_{dst}. This promotes α_{dst} from a Poisson source to an exponential server with mean two. The time of the k^{th} output of α_{dst} depends on the time of the first $k-1$ outputs of α_{src}. The time of the $k-1^{st}$ output of α_{src} depends on the time of the first $k-n_x$ outputs of α_{dst}. Therefore, if α_{src} is stochastically dominated by a Poisson process with rate $1/4$ for its first $k-n_x$ outputs, then, by lemma 4, α_{dst} will be as well for its first $k-1$ outputs. This implies that α_{src} is stochastically dominated by a Poisson process with rate $1/4$ for its first k outputs. Because $n_x \geq 2$, this provides an induction argument that all outputs of α_{src} and α_{dst} are dominated by Poisson sources with rates $1/4$. Property 1 ensures that the network with this first queue added satisfies the claims of the proposition.

The next $n_y - 1$ queues are added to connect the left and right edges of the array. Each new queue promotes an exponential server node of the left edge to a fork/join node. Arguments like those for lemmas 3 and 4 show that the outputs of the new networks are stochastically dominated by those of the array. Now, we have a cylinder with server nodes along its bottom edge and fork/join nodes everywhere else.

We now add the n_x queues to join the top and bottom edges to form a torus. These create n_x new squares. The output times for the torus are stochastically dominated by those for the cylinder which can be shown by an induction argument over time similar to the one above. □