Submitted to ASYNC 2001
Please do not redistribute

How to Achieve Worst-Case Performance

Mark R. Greenstreet and Brian de Alwis
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4
Canada
{mrg,bsd}@cs.ubc.ca

October 3, 2000

Abstract

"Average case performance" is an oft-cited motivation for self-timed design. In self-timed designs, computations proceed according to handshakes, and these handshakes can reflect the actual time required for operations rather than the worst-case time. The intuitive argument is that this should lead to systems whose performance reflects the average-case performance of their components. This paper shows that such intuition is often wrong.

This paper describes a connection between self-timed circuits and percolation networks. Percolation networks are a class of infinite graphs originally used to model critical phenomena arising from fluid flows in porous media. This paper shows how these techniques can be used to show the frequent existence of long chains of slow operations in self-timed designs. These chains can give rise to performance that is closer to worst-case than average-case.

This paper makes three contributions. First, it describes a fundamental connection between percolation networks and self-timed circuits. Second, it presents novel methods for studying percolation networks that arise in the analysis of self-timed circuits. Third, it gives examples of self-timed circuits whose performance is limited by percolation phenomena.

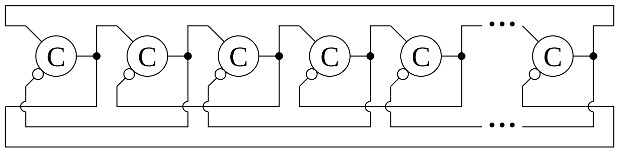

Figure 1: A ring of self-timed processors

1 Introduction

Consider a self-timed ring of processors as shown in figure 1. The C-elements in the figure represent the control path of a typical self-timed pipeline. When a stage is in the same state as its successor and in the opposite state as its predecessor, then the stage is enabled to change to the state of its predecessor. In this paper, we are concerned with the rate at which each C-element fires. We will assume that the time between when a C-element is enabled and when it fires is a random variable; that there is a separate such random variable for each operation performed by each C-element; and that these random variables are independent and identically distributed. By the symmetry of "bubbles" and "tokens" in the pipeline (see [4]), the highest throughput is obtained when the number of bubbles and tokens are equal.[1] In particular, if there are N C-elements in the ring, then

[1]This assumes that forward and reverse latencies are random variables with the same distribution. We will make this

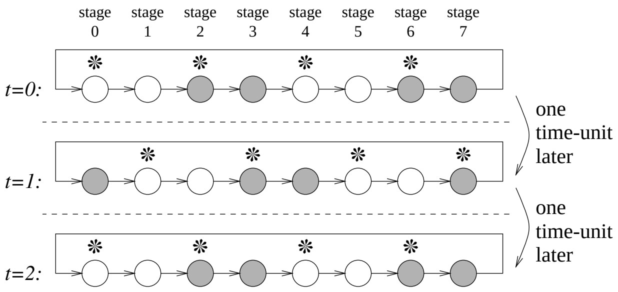

Figure 2: A Ring with C-elements with Unit Delay

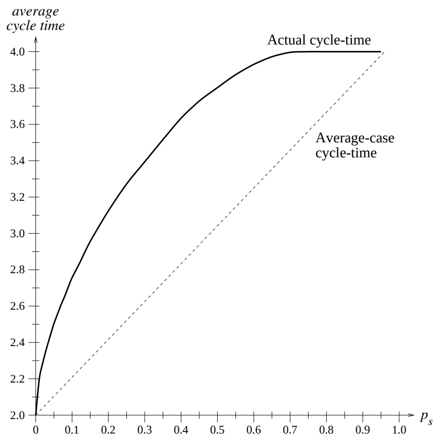

Figure 3: Cycle Time vs. p_s (probability of slow operations)

maximum performance is obtained when there are $N/4$ segments of adjacent C-elements whose outputs are high separated by $N/4$ segments of adjacent C-elements whose outputs are low. In the following, we will assume that this balance of tokens and bubbles holds.

The simplest timing model to analyze is one where each C-element takes exactly one time unit to fire after it becomes enabled. Figure 2 shows this scenario. Enabled processors are marked with a ❊. Initially, all of the even-indexed stages are enabled. One time unit later, these stages fire, and all of the odd-indexed stages become enabled. One more time unit later, the odd-indexed stages fire, and the even-indexed stages become enabled again. Thus, each stage has a cycle time of two time units.

The performance of self-timed rings have been studied in various contexts. For example, [14] analyzed various, regular arrays of self-timed processors, assuming each processor has a fixed time for each operation. Self-timed rings with exponentially distributed processing times were analyzed in [4], and self-timed meshes in [11]. Xie and Beerel have developed tools that analyze general networks of self-timed processors for general probabilistic models [15, 16].

In this paper, we focus on the case where the times for operations are Bernoulli random variables [2] (suitably shifted and scaled): each processor has two possible processing times, "fast" and "slow", where the probability of an operation being fast is p_f, and therefore the probability of an operation being slow is $p_s = 1 - p_f$. The reader can think of this as flipping a weighted coin each time the processor is enabled – if the coin comes up heads, then the operation is fast; if the coin comes up tails, then the operation is slow. We are interested in how the performance varies with p_f.

As an example, consider the case where a fast operation takes one time unit and a slow operation takes two time units. Figure 3 shows the average cycle time as a function of p_s, the probability that an operation is slow. When $p_s = 0$, all operations take one time unit, and the average cycle time is two time units as described above. Likewise, when $p_s = 1$, all operations take two time units, and the average cycle time becomes four time units. The dashed line shows the cycle times that would be achieved if cycle time was determined by the average delay of a C-element. The solid curve shows the actual cycle times observed by Monte-Carlo simulation.

This curve has three salient features:

1. The actual performance only matches the average-case scenario for the deterministic pro-

assumption to simplify the analysis. Similar results apply with asymmetric latencies.

cessing time scenarios. For most values of p_s, the actual cycle time is much larger than predicted by a simple, average-case model.

2. The curve reaches the maximum cycle time for a value of p_s near 0.72. For p_s above this value, the cycle time is exactly the time that would be observed if all processors were always slow. In other words, decreasing the delay for up to 28% of the operations has no effect on the overall performance! This is due to *critical phenomena* in the behavior of the self-timed ring.

3. The curve is vertical at $p_s = 0$.

In the remainder of this paper, we examine the causes for these phenomena and explore their implications for self-timed design.

2 Task Graphs and Percolation Networks

This section presents the two modeling concepts behind our analysis: task graphs and percolation networks. Task graphs model the precedence relations between operations in parallel processes. Percolation networks are infinite, random graphs, where the basic question is whether or not the graph contains a connected component of infinite extent. We show that when the times for operations are Bernoulli random variables, then we can view the resulting task graph as a percolation network. If this percolation network has an infinite, connected component, then the original self-timed system operates with worst-case performance.

2.1 Task Graphs

A task graph is a way of visualizing the operations of a a parallel process [3]. The graph has one vertex for each operation of each processor. In particular, let vertex $v_{i,j}$ correspond to the j^{th} operation of processor i. Edges in the graph are directed, with an edge from vertex v_{i_1,j_1} to vertex v_{i_2,j_2} if the ${j_2}^{th}$ operation of processor i_2 depends directly on the ${j_1}^{th}$ operation of processor i_1. For example, figure 4 shows the

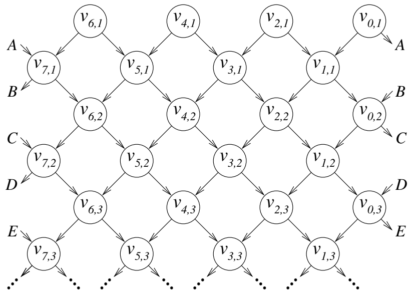

Figure 4: Task Graph for a Self-Timed Ring

task graph for the self-timed ring depicted in figures 1 and 2. More generally, consider a ring with N stages, where N is even. The set of edges in the task graph is:

$$\begin{array}{l} \{(v_{i,j}, v_{i\oplus 1,j}), (v_{i,j}, v_{i\oplus 1,j}) | i \text{ even}, \, j \geq 0\} \cup \\ \{(v_{i,j}, v_{i\oplus 1,j+1}), (v_{i,j}, v_{i\oplus 1,j+1}) | i \text{ odd}, \, j \geq 0\} \end{array} \quad (1)$$

where $\oplus$ and $\ominus$ denote addition and subtraction modulo N.

With each vertex, $v_{i,j}$, we associate $t(v_{i,j})$, the time at which the operation takes place, and $\delta(v_{i,j})$ the delay between enabling the process and performing it. We also write $t_{i,j}$ and $\delta_{i,j}$ to denote $t(v_{i,j})$ and $\delta(v_{i,j})$ respectively. The recurrence below relates these two quantities to each other:

$$t_{i,j} = \begin{cases} \delta_{i,j}, & i \text{ even}, \; j = 0 \\ \delta_{i,j} + \max(t_{i\ominus 1,j-1}, t_{i\oplus 1,j-1}), & i \text{ even}, \; j > 0 \\ \delta_{i,j} + \max(t_{i\ominus 1,j}, t_{i\oplus 1,j}), & i \text{ odd} \end{cases} \quad (2)$$

The first case with i even and $j = 0$ describes the firing times of the processors that are initially enabled. The other two cases state that a processor becomes enable at the latter of receiving data from its predecessor ($t_{i\ominus 1,\ldots}$) and receiving an acknowledgement from its successor ($t_{i\oplus 1,\ldots}$, and that the processor fires $\delta_{i,j}$ time units after becoming enabled.

For our particular problem, $\delta_{i,j}$ is derived from a Bernoulli random variable. In particular, let δ_s be the time for a "slow" operation and δ_f be the time for a "fast" operation with $\delta_s \geq \delta_f$. The time for processor i to perform its j^{th} operation is

$$\delta_{i,j} \quad = \quad t_f + (t_s - t_f)\beta(v_{i,j}) \tag{3}$$

where the $\beta(v_{i,j})$'s are independent Bernoulli random variables that are one with probability p_s and zero with probability $p_f = 1 - p_s$. We also write $\beta_{i,j}$ to denote $\beta(v_{i,j})$.

An alternative way to write equation 2 is as a maximum over all paths from an initial vertex. Let $\Gamma_{i,j}$ be the set of all paths from an initial vertex (i.e. one with i even and $j = 0$) to vertex $v_{i,j}$:

$$\Gamma_{i,j} \;=\; \begin{cases} v_{i,j}, & i \text{ even},\ j = 0 \\ s \cdot v_{i,j}, & s \in (\Gamma_{i\ominus 1,j-1} \cup \Gamma_{i\oplus 1,j-1}), \\ & i \text{ even},\ j > 0 \\ s \cdot v_{i,j}, & s \in (\Gamma_{i\ominus 1,j} \cup \Gamma_{i\oplus 1,j}), \\ & i \text{ odd},\ j > 0 \end{cases} \tag{4}$$

Using Γ, equation 2 can be rewritten written as:

$$t_{i,j} \quad = \quad \max_{s \in \Gamma_{i,j}} \sum_{v \in s} \delta(v) \tag{5}$$

For the case where δ is defined as in equation 3,

$$t_{i,j} \quad = \quad (2 * i + (j \bmod 2))\delta_f + \max_{s \in \Gamma_{i,j}} \sum_{v \in s} \beta(v) \tag{6}$$

In English, this says that the time at which processor i completes is determined by the path from an initial vertex through $v_{i,j}$ with the maximum number of slow operations. In the next section, we will show that for p_s greater than a critical value, the probability that for every j there exists some i such there is a path from an initial state through vertex $v_{i,j}$ where every vertex on the path is slow. This gives rise to worst-case performance.

2.2 Percolation Networks

Percolation networks [6] are "crystalline lattices": graphs that are formed by repetition of a finite pattern graph. With each edge or vertex of the pattern graph, there is a probability given for whether or not that edge or vertex is present in any particular instance of the pattern graph in the entire graph. For example, figure 2.2 shows a percolation network

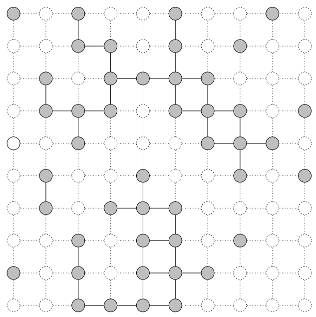

Figure 5: A Percolation Network

based on a square mesh where each possible vertex has a 0.5 probability of inclusion in the network. In this figure, included vertices are drawn shaded with a solid boundary; non-included vertices are drawn with a dotted boundary. Likewise, edges between two included vertices are as drawn solid arrows, and edges that have one or both endpoints an a non-included vertex are drawn dotted.

Percolation theory originally arose in the study of the flow of coal dust particles through gas masks and the study of fluid flows through porous rocks [7], hence the name "percolation." Percolation models are characterized by whether they are "site models" in which the probabilities apply to the vertices of the pattern graph, or "bond models" in which the probabilities apply to the edges. Furthermore, a distinction is made between oriented models in which edges are directed, and non-oriented models in which they are not. Non-oriented, bond models have been the most intensely studied (see [5]).

Percolation networks are characterized by phase transitions. Let p be the probability that a site (or bond) is present in a percolation network. What is the probability that there is a connected component spanning the network? For the fluid flow problems mentioned above, the presence of a bond models an open pore in a rock. A spanning network of open pores indicates that fluid can seep through the rock. In particular, we can study this question in the limit that the size of the network goes to infinity assuming that the network grows in all dimensions at roughly the same rate. In this limit, percolation networks are characterized by a critical probability, p_c, whose value depends only on the pattern. The probability of a connected component that spans the network is zero for $p < p_c$ and one for $p > p_c$.

Critical probabilities have been published for a variety of percolation networks. The most relevant such results to our current work are Kesten's famous proof of a critical probability of 0.5 for a non-oriented, bond model on a square, two-dimensional mesh [9], and Ziff's numerical derivation of a bound of ≈ 0.5927460 for a non-oriented, site model network on the same square mesh [17]. Kesten used arguments based on the graph and its dual, noting that both are square meshes. Ziff's used hull-walking techniques to generate Monte-Carlo estimates and to derive coefficients for renormalization methods. For a summary of some recent results in percolation theory, Stauffer [13] provides a short and excellent survey.

2.3 The Connection between Task Graphs and Percolation Networks

Consider again the task graph for a ring of self-timed processors as illustrated by figure 4. We construct a site-model, oriented percolation network from this graph in the obvious way: sites of the percolation network correspond to vertices of the task graph; bonds correspond to edges; and a site is included if the corresponding operation is slow. Note that the task-graph has cylindrical topology; thus, our percolation network is a square mesh on a cylinder. Because we are concerned about the limits as the size of the network goes to infinity, the distinction between a mesh on a plane and a mesh on a cylinder is insignificant. In the next section, we examine this network and estimate its critical probability, p_c.

If p_s, the probability of a processor being slow is greater than p_c, then the probability of the existence of a path that spans the network and consists of only slow processors goes to one as the size of the ring goes to infinity. Let N be the number of processors in the ring, and G be the number of generations considered. Note that the width of the task graph is $N/2$ and the height is $2G$. The existence of a path of all slow processors says that with high probability there is some processor in generation G that completes its operation at time $2G\delta_s$. Note that every task in generation G depends on every task in generation $G - N/2$. If G/N is large but bounded (e.g., 1,000,000),[2] then with high probability no task in generation G completes before $(G - N/2)\delta_s + (N/2)\delta_f$. Thus, the performance of the ring is very close to worst-case performance.

3 The Critical Probability

In this section, we present two methods for estimating the critical probability of a site model, oriented, percolation network. First, we present an analytical approach where we construct Markov chains that approximate the behavior of the percolation network. Second, we make an estimate based on Monte Carlo simulations. The Monte Carlo simulations give a more accurate estimate. The Markov chain approach has the advantage that the estimates it provides are strictly lower bounds.

3.1 A Markov Chain Approach

Consider again the task graph from figure 4. To start our analysis, it is helpful to relabel the tasks. Let

$$u(i,j) \quad = \quad v(2*i+1-j,\ \lfloor j/2 \rfloor) \qquad (7)$$

With this relabeling, the edges of the task graph from figure 4 become

$$\{(u(i,j), u(i,j+1)), (u(i,j), u((i+1) \bmod N, j+1))\} \qquad (8)$$

Figure 6 shows the relabelled task graph. We view each row in figure 6 as a generation. We will say that a task is "critically slow" if the task is slow and either the task is in the first generation, or the task has at least one predecessor that is critically slow. If a task in generation j is critically slow, then it completes its operation at time $j\delta_s$. We are interested in the probability of finding an arbitrarily long path of critically slow processors.

[2] The requirement that G/N be bounded prevents scenarios where G is exponentially larger than N (e.g. $(u/p_f)^N$) for some $u > 0$, in which case the network is spanned with probability at most e^{-u}.

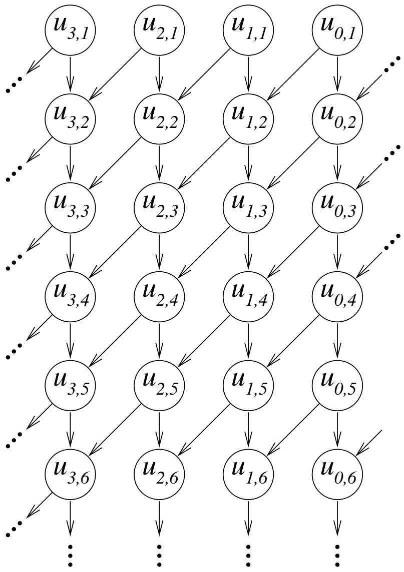

Figure 6: Relabelled Task Graph

We associate with each generation a configuration column vector: an element of this vector is '1' if the task in the corresponding position in the task graph is critically slow; otherwise the element is '0'. The process that goes from the state vector for one generation to the state vector for the next is a Markov process [2]. Let M be transition probability matrix for this process. In the limit as N goes to infinity, M has an uncountable number of states. To compute lower bounds on the critical probability, we approximate this Markov process with a finite one. To describe this approximation, we first introduce the notions of equivalent and canonical states.

Let x, x_1, x_2, and x' be configurations. Let $\mathsf{rotate}(x, k)$ be the configuration obtained by rotating the elements of x by k positions to the right. Let $\mathsf{reverse}(x)$ the the configuration obtained by reversing the order of the elements of x. The following

```
Configuration canon(Configuration x) {
  if(x ≡ 0*) return(0*);
  while(x[0] ≠ 1) x = rotate(x, 1);
  x = fill(x);
  if(x > reverse(x)) x = reverse(x);
  return(x);
}
```

Figure 7: A function for computing canonical states

equalities are straightforward to prove:

$$\begin{aligned} P\{x \to x'\} &= P\{\mathsf{rotate}(x,k) \to \mathsf{rotate}(x',k)\} \\ P\{x_1 101 x_2 \to x'\} &= P\{x_1 111 x_2 \to x'\} \\ P\{x \to x'\} &= P\{\mathsf{reverse}(x) \to \mathsf{reverse}(x')\} \end{aligned} \tag{9}$$

where $P\{x \to x'\}$ is the probability that x' is the next configuration given that x is the current configuration. These equalities say that configurations can be rotated, reversed, or filled without changing the probability of the existence of a path of critically slow processors. We write $\mathsf{fill}(x)$ to indicate the state obtained from x by replacing every occurrence of the substring 101 with the substring 111.

Using this equalities, we define the notion of a canonical configuration. Let canon be defined as shown in figure 7. A configuration, x is canonical if $x = \mathsf{canon}(x)$. Two states, x, and x', are equivalent if $\mathsf{canon}(x) = \mathsf{canon}(x')$. We write $x \sim x'$ to indicate that x and x' are equivalent. Using canonical states significantly reduces the size of the state space. For example, with $N = 4$, there are a total of 16 configurations, but only six canonical ones: 0000, 0001, 0011, 0111, 1001, and 1111. With $N = 12$, there are a total of 4096 configurations, but only 265 are canonical.

To approximate the percolation network for the self-timed ring, we chose a set of canonical configurations, W. We assume that $W \supseteq \{0*,\ 1\}$. Let W' be the set of all successors, not necessarily canonical, of configurations in W, and let M_W be the $|W'| \times |W|$ transition probability matrix from states in W to states in W'. It is straightforward to show that there are configurations in W whose canonical equivalents are not in W; therefore $|W'| > |W|$, which means that M_W is not square. We handle these troublesome successors that don't have representations in W by decomposing each such configuration into two configurations, each with its canonical version in W. We then treat these two configurations independently.

One can think of these decompositions as a game against an adversary. Our goal is to drive all configurations to the $0*$ configuration, thus showing that their are no contiguous components spanning the percolation network. To do this we create a Markov chain whose states are the elements of W. When a successor of a configuration in W is outside of W, we break the successor into two configurations, and hand one of these pieces to the adversary who can put it anywhere in the successor generation. The adversary's goal is to maximize the probability of there being a chain of critically slow sites that span the network. To do this, the adversary places the piece that we relinquished arbitrarily far from any other slow sites. This minimizes the likelihood of a two critically slow paths from meeting and merging.

Now, we'll restate the previous paragraph more formally. We create a $|W| \times |W'|$ matrix, $Q_{W'}$ to map configurations in W' back to canonical configurations in W. For each configuration, $w' \in W'$ such that $\mathsf{canon}(w') \notin W$, we choose two configurations, w'_1 and w'_2 such that $w' = w'_1 w'_2$ and $\mathsf{canon}(w'_1) \in W$ and $\mathsf{canon}(w'_2) \in W$. We now consider w'_1 and w'_2 independently. We set the entries in column w by the rule:

$$Q_{W'}(x, w') \;=\; |w'_1 \sim x| + |w'_2 \sim x| \qquad (10)$$

where $|i = j|$ is one if $i = j$ and zero otherwise. For $w' \in W'$ such that $\mathsf{canon}(w') \in W$, we simply set $Q_{W'}(\mathsf{canon}(w'), w')$ to one, and all other entries in column w to zero.

As an example, let W be the set of all canonical configurations that can be represented in a window of width nine. Let w = 111001111; w is a configuration in W. The set of successors of w are w'_0 = 1111011111 and any configuration that can be obtained by replacing a 1 element in w'_0 with a 0. For example, w' = 1011001001 is a successor of w. $\mathsf{canon}(w')$ = 1001001111 is not in W (it requires a window of width 10). Let w'_1 = 110100100 and w'_2 = 1.

w	bound	w	bound
2	0.5825	9	0.6580
4	0.6258	10	0.6612
6	0.6438	11	0.6639
8	0.6542	12	0.6663

Table 1: Lower bounds for the critical probability for various window sizes

To obtain the canonical configuration for w'_1 we rotate w'_1 to places to the left to obtain 1101001. Next, we replace 101 substrings with 111 to obtain 1111001. Finally, we note that 1111001 $>$ 1001111. Thus, the canonical configuration for 110100100 is 1001111. The canonical configuration for 1 is 1. Both of these canonical configurations are in W. Using this decomposition for 1101001001, $Q_{W'}(1, 1101001001) = 1$ and $Q_{W'}(100111, 1101001001) = 1$.

Let $A_W = Q_{W'} M_W$. Matrix A_W is $|W| \times |W|$. Let $w \in R^W$ be a vector, such that $w(i)$ is the expected number of independent local occurrences of i in the current global configuration. Then, $A_W\, w$ is a vector whose elements are an upper bound on expected number of independent local occurrences of the configurations in W of the successor of w. The configuration $0*$ is a sink: $P\{0* \rightarrow 0*\} = 1$. Therefore, A_W has an eigenvalue of 1 whose eigenvector corresponds to this configuration. If all other eigenvalues of A_W have magnitudes less than 1, then all configurations in our approximate system lead almost surely to the $0*$ configuration. Since our approximate system overestimates the number of critically-slow sites, this provides a bound for the original percolation network. In particular, we find the largest value of p_s such that all eigenvalues of A_W other than the one for $0*$ have magnitudes less than one. This value of p_s is a lower bound for p_c.

We considered sets for W that consisted of all canonical configurations that could be represented in a window of width w sites. If a successor configuration exceeded this window, we split of the leftmost 1 as a separate configuration. Table 1 shows the estimates we obtained for various window sizes.

3.2 Monte Carlo Simulations

In addition to our analytical approach described above, we estimated the critical probability using Monte Carlo simulations. Simulating a ring with 1000 processors for 4000000 time steps with various values of p_s we conclude that $p_c \approx 0.72$. The Monte Carlo estimate is certainly more accurate than the lower bound computed above. We see the two methods as complementary. The analytical approach provides a proven bound and provides some insight into the behavior of the percolation network. Our Monte Carlo simulations provide a more accurate estimate but no guarantees.

4 Other Distributions

The Bernoulli type distributions that we considered in the earlier sections are very simplistic. It is natural to ask if these results apply to other processing time distributions as well.

A simple result is that if a regular array of processors has a corresponding percolation network with critical probability p_c, and t_0 is chosen such that

$$P\{\text{processor delay} > t_0\} \quad \geq \quad p_c \tag{11}$$

then the array operates no faster than it would if all operations take constant time t_0.

The propensity of slow operations to dominate performance shows up for other distributions as well. For example, figure 8 shows the distribution function for the maximum carry chain length in additions performed by the ALU of an ARM microprocessor executing the Dhrystone benchmark.

We simulated an eight processor ring where processing times were independent and had the same distribution as the carry chain lengths for the ARM, i.e. a carry chain length of k corresponds to a processing time of k time units. The average delay for such a processor is 21.0 time units, which would give rise to a cycle time of 42.0 time units if average case delay determined performance. Simulation indicates that the actual delay is 54.9 time units which is 30% lower than predicted by the average case values. The worst-case value of 64 time units per cycle is a more

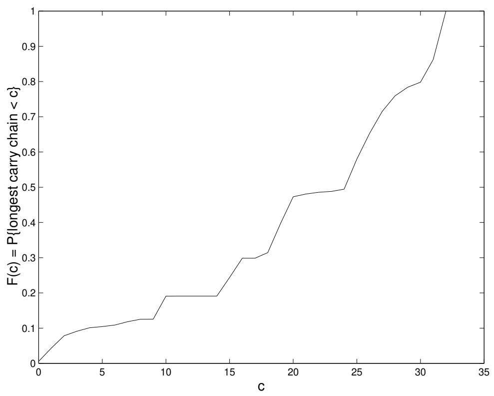

Figure 8: Distribution of length of longest carry chain for an ARM ALU executing the Dhrystone benchmark

accurate estimate of the actual performance than the estimate based on the average adder delay.

Figure 9 shows the sensitivity of the eight processor ring described above to the distribution of carry chain lengths. We computed the sensitivities by performing Monte Carlo simulations for 32 distributions that were linearly independent, small perturbations of the distribution shown in figure 8. We then solved for the cycle time sensitivities for each carry chain length. For example a sensitivity of -1.4 for length c indicates that increasing the distribution function by ϵ at for carry chains of length c decreases the cycle time by 1.4ϵ time units. If performance were determined by the average adder delay, then the sensitivity would be the dotted line shown in the figure. Instead, we observe that the fast cases contribute very little to performance, while carry chains of lengths from 25 to 32 have a very strong effect. For example consider an alternative design where all operations with carry chains of length up to sixteen take nearly zero time, but carry chains of lengths twenty-five or longer take the full 32 time units. With this change, the average processing time decreases by 8%, but the average cycle time for processors in the ring *increases* by 3.6%. As this shows, there can be design changes that lower performance by decreasing the average delay of the components.

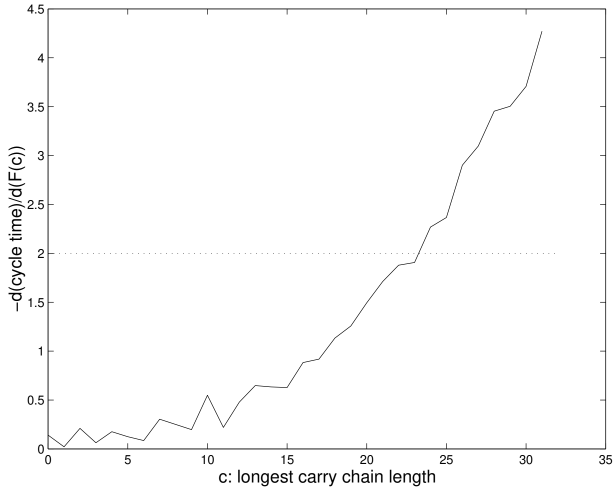

Figure 9: Sensitivity of performance to the distribution of the longest carry chain length

5 Two-Dimensional Processor Meshes

This section extends the ideas of the previous sections to networks with more complicated interconnection. In particular, we consider the two-dimensional mesh described in [11]. Figure 10 shows this mesh. Each processor communicates with its north, south, east, and west neighbours. To avoid boundary conditions, we embedded the mesh on a torus. When a processor

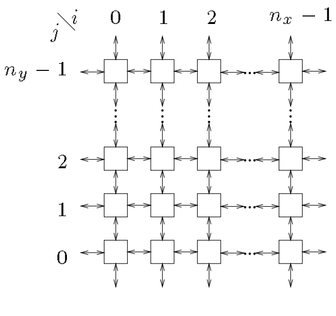

Figure 10: A four-connected mesh

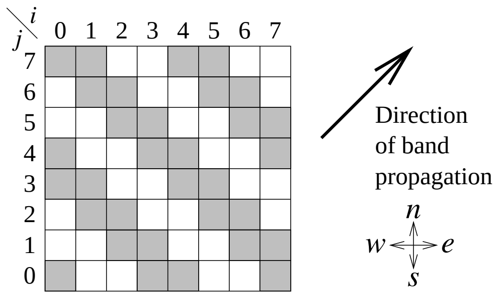

Figure 11: Bands on the four-connected mesh

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. This protocol is delay-insensitive.

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 11 shows, these contiguous regions form bands around the torus that correspond to the segments of one-dimensional pipelines. In [11] we showed that the highest throughput is achieved when these bands are on average two processors wide, the situation shown in figure 11.

The task graph for the two-dimension mesh looks like a stack of checker boards. On even numbered layers, there is a vertex for each black square, and on odd numbered layers, there is a vertex for each red square. There are four directed edges from each vertex to its four nearest neighbors on the next layer down. This structure gives rise to a three-dimensional, oriented, site-model percolation network. Monte Carlo simulations indicate that the critical probability for this percolation network is roughly 0.345. This means that if just over a third of the operations are slow, the overall throughput of the mesh is the same as if all operations were slow.

Figure 12 shows the average cycle time for the two-dimensional mesh as a function of the fraction of the processors that are slow. As in figure 3, a fast operation takes one time unit and a slow operation takes two. Qualitatively, the curve has the same overall shape as that for the ring. Quantitatively, the degra-

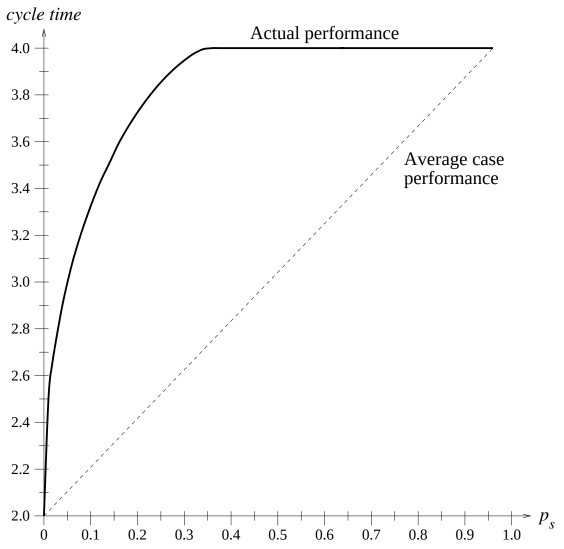

Figure 12: Cycle Time vs. p_s for a two-dimensional mesh

dation of cycle time when just a few slow operations are added is quite severe. For example, if roughly 80% of the operations are fast, the overall cycle time is only 14% of the way from the all-slow performance to the all-fast performance.

This mesh example shows that increasing the degree of connectivity in a self-timed design decreases the critical probability in the corresponding percolation network. This is not surprising. With more dependencies in the task graph, critically slow paths have more opportunities to propagate. Conversely, introducing FIFO buffers between nodes in a mesh or other high-degree interconnect will lower the critical probability. Determining optimal rules for inserting FIFOs is a topic for future research.

6 Conclusions

We have described a connection between the timing behavior of self-timed circuits and percolation networks. In particular, networks of self-timed elements can display critical phenomenon where long chains of slow operations determine the performance of the design. In this case, the overall performance tends to closely match the worst-case performance when every operation takes its maximum amount of time.

We explored the relationship between self-timed rings and two-dimensional, site-model, oriented percolation networks. These networks are characterized by a critical probability: if the probability of an operation being slow is greater than this probability, then worst case behavior results. The critical probability for the two-dimensional, site-model, oriented network does not appear to have been studied in the main literature on percolation networks. We obtained an estimate of 0.72 through Monte Carlo simulation, and a lower bound of 0.6663 by analytical methods.

The asynchronous community has had a longstanding fondness for "average case performance" (e.g. [12, 10, 1, 8]). As we have shown, actual performance often corresponds much more closely to the worst-case performance of the components than to the average-case. In fact, it is possible to "optimize" components in ways that decrease the average-case delay of the component while decreasing system performance. In many cases, performance is largely determined by the slowest 10–20% of the processing time distribution of the components. Thus, "make the common case fast" may not be as important as "make the slow case fast."

The critical behaviors that we've described arise because of handshake protocols that we use: when a component waits for all of it's inputs to be available, the it must wait for the last one. This gives slow events an opportunity to propagate through large expanses of the underlying percolation network. There may be opportunities to overcome these limitations by using protocols that don't wait for all inputs to arrive. Of course, this introduces a need for arbiters to decide when to proceed, and the overhead of arbitration is unacceptable in many fine-grained pipelined applications. There clearly remains much to discover about how to obtain optimal performance in self-timed systems.

Acknowledgements

We are grateful to Nick Pippenger and Claire Kenyon for enlightening conversations.

References

[1] W. chun Chou, P. A. Beerel, et al. Average-case optimized technology mapping of one-hot domino circuits. In *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 80–91, San Diego, California, Apr. 1998. IEEE.

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

[3] E. Gelenbe. *Multiprocessor Performance*. John Wiley and Sons, 1989.

[4] 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.

[5] G. R. Grimmett. Percolation. URL: http://www.statslab.cam.ac.uk/~grg/papers/USopt.ps.

[6] G. R. Grimmett. *Percolation*. Springer, second edition, 1999.

[7] J. Hammersley and D. Welsh. Origins of percolation theory. *Annals of the Israel Physical Society*, 5:47–57, 1983.

[8] K. W. James and K. Y. Yun. Average-case optimized transistor-level mapping of extended burst-mode circuits. In *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 70–79, San Diego, California, Apr. 1998. IEEE.

[9] H. Kesten. The critical probability of bond percolation on the square latice equals $\frac{1}{2}$. *Mathematical Physics*, 74:41–59, 1980.

[10] S. M. Nowick, K. Y. Yun, et al. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 210–223, Eindhoven, The Netherlands, Apr. 1997. IEEE.

[11] P. B. Pang and M. R. Greenstreet. Self-timed meshes are faster than synchronous. In *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 30–39. IEEE, Apr. 1997.

[12] C. L. Seitz. System timing. In *Introduction to VLSI Systems* (Carver Mead and Lynn Conway), chapter 7. Addison Wesley, 1979. See the adder example in section on pages 251–252.

[13] D. Stauffer. Minireview: New results for old percolation. *Physica A*, 242(1–2):1–7, Aug. 1997.

[14] L. Thiele. On the analysis and optimization of self-timed processor arrays. *INTEGRATION*, 12(2):167–187, Dec. 1991.

[15] A. Xie and P. A. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75. IEEE Computer Society Press, Apr. 1997.

[16] A. Xie, S. Kim, and P. A. Beerel. Bounding average time separations of events in stochastic timed Petri nets with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 94–107, Apr. 1999.

[17] R. M. Ziff. Spanning probabilty in 2D percolation. *Physical Review Letters*, 69(18):2670–2673, Nov. 1992.