Tree Arbiter with Nearest-Neighbour Scheduling

Isi Mitrani and Alex Yakovlev*
Dept. of Computing Science, University of Newcastle upon Tyne, NE1 7RU

Abstract

A tree arbiter designed to minimize the average delay between consecutive allocations of a contentious resource is described and evaluated. The idea is to divide the competing users into nested clusters corresponding to sub-trees of varying size, and to keep the resource within the smallest cluster currently containing requests. This design reduces the number of steps in the arbitration procedure and hence increases the overall throughput of requests. The performance of the new arbiter is analysed and is compared to that of an algorithm which maintains first-in-first-out order among requests. The trade-offs between the nearest-neighbour and FIFO designs are examined numerically over wide ranges of parameter values. It is shown that when the system becomes large and/or heavily loaded, the benefits of the the nearest-neighbour arbiter become greater.

Keywords: arbitration, asynchronous systems, conflict resolution, mutual exclusion, performance analysis, resource allocation, tree arbiter.

1 Introduction

Parallel and distributed systems use arbiters to resolve mutual exclusion between independent users accessing shared resources. For example, a number of processors may be competing for a bus, sending requests to it asynchronously and independently of each other. Whenever new requests arrive, or old ones are completed, the arbiter has to decide which of the waiting processors should be allowed to use the bus. Other examples of such systems involve multi-port memories and packet routers [2, 9].

The ability to use the resource is usually signalled by the possesssion of a "privilege token". A typical user behaviour pattern is to request the token, wait until it is granted, use the resource, release the token, perform tasks which do not require the resource (this last activity will be referred to as *thinking*), request the token again, etc. In those terms, the arbiter is a hardware device which receives token requests and token release signals as inputs, and produces token grants to individual users as outputs. This paper is concerned with some trade-offs involved in the construction of that device: different design alternatives and their implications for the performance of the system.

The design of a fully parallel arbiter, like that of a cross-bar switch, tends to be too complex, too expensive and too difficult to scale up. A more common solution is therefore to put together a number of simple components which combine parallel and sequential actions. The usual basic building blocks are 2-way arbiter cells which form the nodes of a binary tree. Each cell passes token request and release signals from descendents to parent (if any), and token grants from parent to descendents; the latter are either users or other cells. Thus a single cell (a binary tree of height 1) can arbitrate between 2 users; a tree of height 2 containing three cells can arbitrate between 4 users; in general, an arbiter for $n = 2^k$ users would consist of 2^{k-1} cells forming a complete binary tree of height k. Because arbitration tasks are performed in sequence by cells at different levels of the tree, this type of arbiter is also described as *cascaded* (see figure 1).

If a user can occupy itself with other tasks while waiting for the resource to become available, it would be sensible to design an arbiter which delivers 'negative acknowledgements' when the resource is busy (see [4]). Here we are concerned with systems where the opposite is the case: once a user requests the resource, it cannot do anything useful until that request is granted; the interval between requesting and receiving the token represents idle waiting time. In these circumstances, reducing the arbitration time leads to a more efficient system and is therefore a worthwhile objective to pursue. Two recent designs of low latency arbiters where the request is propagated from one cascade stage to another in parallel with

*This work was partially supported by EPSRC grant GR/K70175.

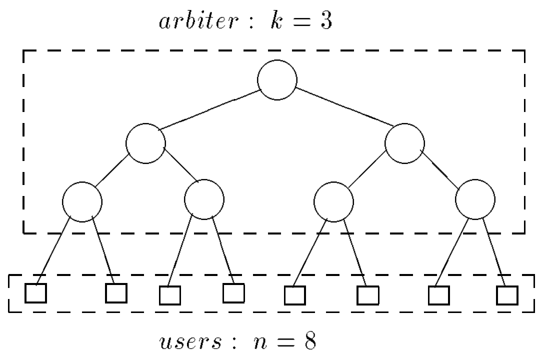

Figure 1: A cascaded tree arbiter with 7 cells and 8 users

mutual exclusion resolution, and where the release of request-grant handshakes in different stages is also done in parallel, are described in [11, 5].

Every action performed by a node in the tree, whether it involves exclusion resolution or passing information to parent or descendent, causes a delay. Hence, a reduction in the arbitration time can be achieved either by reducing each such delay, or by reducing the number of nodes that participate in the arbitration process. An important design consideration in this regard is the token scheduling strategy. Normally, requests for the resource are served in first-in-first-out (FIFO) order. This is implemented by queueing the incoming requests at the root of the tree, where the token is returned after each release. Hence, every arbitration involves passing the token from the root to a leaf node and back; for a tree of height k, this implies a total of $2k$ node delays.

We propose a different scheduling policy. When the token is released, it is of course at one of the leaves of the tree; the user to whom it is allocated next is at another leaf: the latter is chosen so as to minimize the token travel time. Thus, if the sibling of the releasing user is among those waiting, it receives the token after a single node delay. Otherwise, the token returns to the parent of that node and, if any of the users descending from that parent are waiting, one of them receives the token after a total of 3 node delays; etc. This will be referred to as the *nearest-neighbour* scheduling policy.

When comparing the performance of the FIFO and nearest-neighbour policies, the following factors should be taken into account: on one hand, the cells comprising the FIFO arbiter are simpler, and their delay times are shorter; on the other, the number of cells and hence the number of delays involved in a nearest-neighbour arbitration is, on the average, smaller. Modelling and evaluating those trade-offs under different conditions is a major contribution of this paper.

The performance measure used as a criterion for comparison is the long-term average throughput of requests. No premium is placed on preserving their order of arrival.

It should be noted that the arbiter model presented here for performance analysis is relatively high-level and approximate (see section 3). It does not take into account the actual physical delays of a circuit implementation. Such high-level analysis is however very valuable because it reflects both strong and weak points of the nearest-neighbour policy. For example, it clearly demonstrates that the gain in performance that one can get out of this policy compared to the standard FIFO one is normally only 20-30% and certainly not twice or more times.

The definition of the tree arbiter and its behavioural description using Petri nets are given in section 2. The logic circuit implementation of the nearest-neighbour policy is not covered here; one version of such an implementation was presented in [10],pp. 190-193 (called non-resetting arbiter there). Section 3 describes the assumptions, analysis and solution of the FIFO and nearest-neighbour models. Several numerical and simulation results for a number of system configurations are presented in section 4. The conclusion outlines some perspectives for future work.

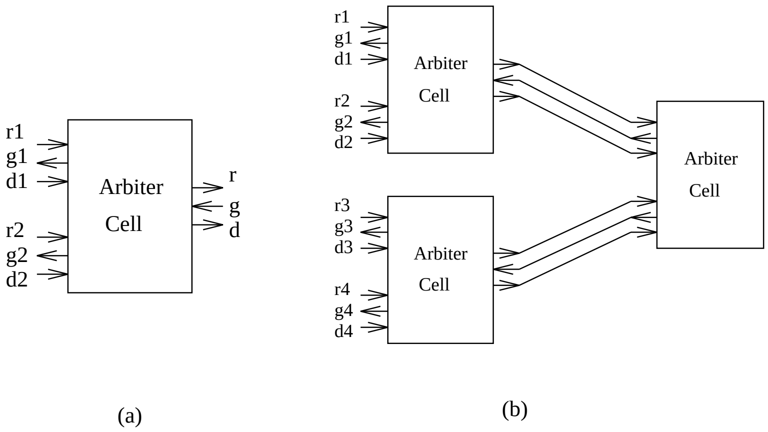

Figure 2: Cascaded arbiter

2 Definition of the arbiter

An asynchronous arbiter is defined as a circuit or a subsystem that dynamically allocates a single shared resource to the user components in a system which is free from common clock. Each user, when it requires the resource, issues an asynchronous *request* and waits until the arbiter produces a *grant* in the form of a *privilege token*. The user who wins the current arbitration session uses the resource and after finishing its task releases the privelege token by sending a *release* signal to the arbiter. This creates the opportunity for another (or even the same) user to acquire the resource.

An arbitration session starts with the receipt of a *release* signal or, if there are no requests present, with the arrival of a *request* signal. It ends, after some finite delay, with the privilege token being allocated to exactly one of the users with pending requests. Requests which arrive while the resource is busy do not trigger arbitration sessions.

The duration of an arbitration session is called the *reaction time* of the arbiter. To minimize the average reaction time and at the same time to avoid an overly complex circuitry, multi-way arbitration is organised by building a cascade of basic cells to form a (balanced) tree. Sometimes such a tree is a natural form of the overall system's topology and so the arbitration structure simply conforms to it.

Thanks to the existence of a well-tested implementation of a standard *2-way mutual exclusion cell* (see, for example, [9]), the multi-way asynchronous arbiter becomes a robust and realistic possibility, despite a non-zero probability of a metastable state in its behaviour [3]. The latter happens when two requests arrive in the same cell very close to each other and the inertiality of the device, which is built on the basis of an SR-flip-flop, may put it to a state in which neither of its two outputs is in its stable HIGH state. This state is however normally not much longer than a simple latch delay, and should not produce any undesirable effects in an asynchronous environment. An interested reader may refer to [3] for more information on the metastability phenomenon. In this paper, we assume that metastability is statistically infrequent and, while being an important issue from the safety point of view, plays a minor role in terms of its effect on the average system's performance.

Each 2-way mutual exclusion cell arbitrates between two users. It propagates *request* and *release* signals from lower to higher levels of the structure, while the grants are generated in the opposite direction. Figure 2 (a) shows one such cell, a 2-way arbiter, with its three *request-grant-release* interfaces $(r1, g1, d1)$, $(r1, g2, d2)$ and (r, g, d), where $(r1, g1, d1)$ and $(r2, g2, d2)$ stand for the links with the descendent cells, generating competing requests at $r1$ and $r2$, and the (r, g, d) triple is the link with the parent cell. An example of a 4-way arbiter, shown in figure 2 (b), illustrates the regular way in which a cascaded multi-way arbiter can be composed from the basic cells.

2.1 FIFO and and nearest-neighbour arbitrarion

A convenient language to capture the dynamic asynchronous behaviour of the arbiter is offered by Petri nets [7]. A Petri net is a graph with two types of nodes, places (circles in the graph) to represent partial states of the system, and transitions (bars in the graph) to represent actions or events. The Petri net model of the 3-cell, 4-user arbiter from figure 2 (b) is shown in figure 3.

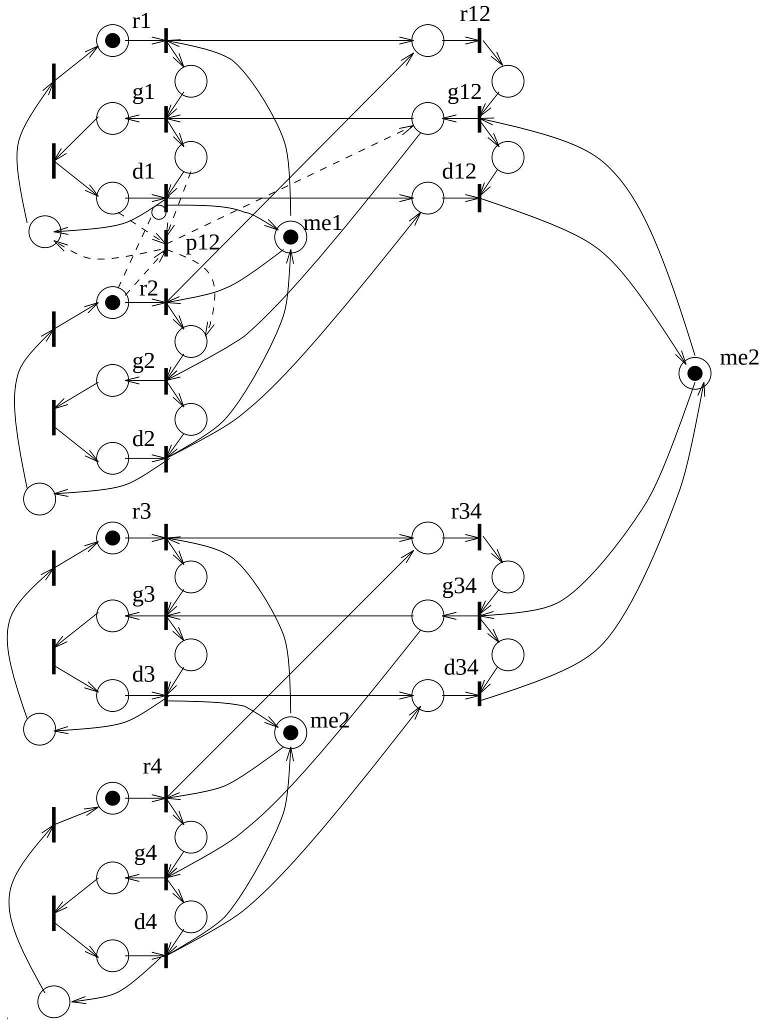

Figure 3: Petri net model for cascaded arbiter (the nearest-neighbour version is shown with dashed arcs)

Consider first the Petri net without the dashed arcs and "bubbles" on the net places. Its transidions are labeled ri or rij (accepting a request), gi or gij (granting the privilege token) and di or dij (receiving a *release* signal). All transitions are timed and possibly stochastic, i.e. an interval of time, which may be random, elapses between the enabling of the transition and its firing (the latter is atomic and instantaneous). Mutual exclusion is accomplished by the places labeled mei or me: when a token is consumed from $me1$ or $me2$ by the firing of one of a pair of ri transitions, the other one cannot fire. Similarly, when the privilege token is consumed from me by the firing of one of the gij transitions, the other one cannot fire.

The initial marking of the model shown in figure 3 corresponds to the state where all the users have submitted requests (all transitions labelled $ri, i = 1, ..., 4$, are enabled), and the privilege token is at me. To trace a possible sequence of transition firings, assume that transition $r1$ fires first. This firing consumes the token from $me1$, and the request propagates to the next stage by enabling transition $r12$. Now if this transition fires before its rival $r34$, the privilege token is taken from me and begins its way through the stages, causing transitions $g12$ and $g1$ to fire in turn. At this point, according to the behaviour of user 1, shown by dotted arcs, the privilege token stays with user 1 for the firing time of the corresponding unlabeled transition (this represents the service time of the resource). When that transition fires, it enables transition $d1$ (user 1 releases the resource) and also another unlabeled transition whose firing time represents the 'think time' of user 1. The firing of transition $d1$ leads to the backward propagation of the privilege token towards place me.

Note that the overall Petri net model is essentially built from fragments corresponding to two-way arbiter cells. This design can easily be extended recursively to arbiters for $n = 8, 16, \ldots$ users.

Analysis of this Petri net model shows the following important behavioural detail. Whenever a user releases the resource (i.e., one of the transitions di fires), the privilege token is returned to the root of the tree (place me). Further, if another user submits a request, it will have to travel through the net, winning arbitration in all the intermediate stages, until it reaches the root. Therefore, if all transitions ri have constant and equal firing times, and similarly for transitions rij, the arbiter serves incoming requests in FIFO order. In practice of course that is not strictly true, due to small delay variations. Nevertheless, violations of the FIFO order are sufficiently rare to justify describing this net as a FIFO arbiter.

Now consider the extended Petri net in Figure 3, including the dashed arcs and the transition labeled $p12$. Note that this net also has a so-called inhibitor arc betwen a place and a transition (the one with a bubble end). The semantics of such an arc is that a marked place prevents the transition from firing, while an unmarked one enables it. In the net shown in Figure 3, this implies that when user 1 releases the privilege token, transition $d1$ can fire only if user 2 has not submitted a request; if it has, then the alternative transition, $p12$, is enabled and the privilege token goes via transition $g2$ to user 2. This mechanism, suitably extended to all cells, implements the nearest-neighbour scheduling policy (for the corresponding asynchronous circuit implementation, see [10]).

3 Performance analysis

For the purposes of performance evaluation, we model the arbiter at a higher level of abstraction than that of the Petri net or the asynchronous circuit. Rather, we consider it as a binary tree of the type illustrated in figure 1, with the users at the leaves and the privilege token moving from node to node, experiencing delays at every step. As well as the height of the tree, k, or the number of users, $n = 2^k$, the model is characterized by the following quantities:

- User think times (intervals between releasing the resource and requiring it again) are i.i.d. random variables distributed exponentially with mean $1/\tau$.

- Resource service times (intervals between giving the privilege token to a user and having it released) are i.i.d. random variables distributed exponentially with mean $1/\mu$.

- Delays of passing the privilege token from node to node are i.i.d. random variables distributed exponentially with mean $1/\nu$ (these are in fact the actual delays of the logic circuit for a two-way arbiter cell, plus the interconnection between adjacent stages; for the sake of simplicity we model them by a single random variable).

The exponential distribution assumptions are made principally for analytical convenience. They can usually be generalized at the price of increasing the complexity of the solution.

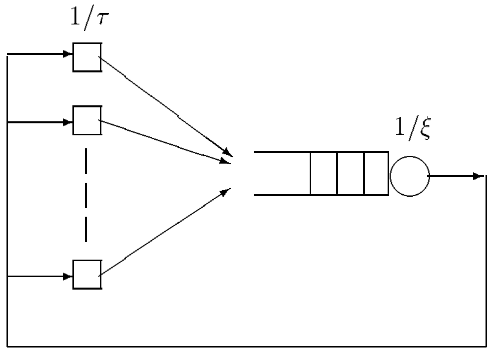

Figure 4: A FIFO queue with n users and a single server

3.1 FIFO arbiter

In the FIFO case, the system can be modelled as a finite-source, single-server queue; see figure 4.

The service rate for this model, ξ, is obtained by noting that, since every arbitration involves passing the privilege token from the root to a leaf and back, the average time during which a request occupies the server is

$$\frac{1}{\xi} = \frac{1}{\mu} + \frac{2k}{\nu} . \tag{1}$$

A simple approximate solution is derived by assuming that the service times are distributed exponentially. That solution is in fact insensitive to the distribution of the user think times. The steady-state probability, p_i, that there are i requests at the arbiter, is given by (see [6]):

$$p_i = \frac{n!}{(n-i)!}(\frac{\tau}{\xi})^i p_0 \quad ; \quad i = 0, 1, \ldots, n , \tag{2}$$

where

$$p_0 = \left[\sum_{i=0}^{n} \frac{n!}{(n-i)!}(\frac{\tau}{\xi})^i \right]^{-1} . \tag{3}$$

An exact solution which treats the service time as a sum of exponentially distributed random variables also exists, but has higher computational complexity.

The throughput of the FIFO arbiter, T, defined as the average number of requests that are completed per unit time, is equal to:

$$T = (1 - p_0)\xi . \tag{4}$$

The average response time of a request, W, i.e. the interval between submitting it to the arbiter and having it granted, is determined by applying Little's theorem [6]:

$$W = \frac{n}{T} - \frac{1}{\tau} . \tag{5}$$

3.2 Nearest-neighbour arbiter

In principle, it would be possible to construct a Markov chain model of the nearest-neighbour arbiter and solve it exactly. However, that is an impractical approach, due to the size of state space that would be required. To describe the state of a tree arbiter of height k, one would have to specify which of the 2^k users have submitted requests, at which node is the token and, if not at a leaf, in which direction it is moving. This leads to a state space whose size grows with k as $2^{k+1}2^{2^k}$.

We shall provide an approximate analysis, based on the notion of a *local busy period*. Consider sub-trees of varying height, together with their sets of users, as illustrated in figure 5.

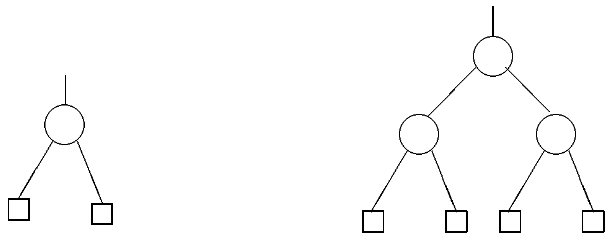

Figure 5: (a) sub-tree of height 1; (b) sub-tree of height 2

Remember that, according to the nearest-neighbour scheduling policy, once the privilege token has entered a sub-tree of height j, for any $j < k$, it remains within it, serving existing and newly arriving requests, until there are no requests present. When that sub-tree is empty, the token goes to the parent of its root, thus finding itself in a subtree of height $j+1$.

Denote by $B_i(j)$ the average interval between the token entering a sub-tree of height j, with i requests present, and leaving it empty. Both entry and exit occur via the root of the sub-tree. We shall make the approximating assumption that the initial i requests *are packed as closely as possible*. For example, if a busy period for a sub-tree of height 2 (see firure 5 (b)), starts with 2 requests present, the assumption is that they come from two sibling users. That assumption simplifies the analysis considerably, without affecting the behaviour of the system unduly. It is possible to solve the model without it, at the price of introducing a large number of different types of busy periods; however the small accuracy gain is not worth the extra effort.

Let also $m_i(j)$ be the average number of requests served during a busy period for a sub-tree of height j, starting with i requests present. If the quantities $B_i(j)$ and $m_i(j)$ are known for $j = k$, one can find the system throughput as follows: A busy period for the full arbiter tree starts with the arrival of one request into an empty system, and ends with the system being empty again; its average duration is $B_1(k)$. The average length of the subsequent idle period, I_k, is equal to $1/(2^k\tau)$ (that is the interval until the first of the 2^k users submits a request). During the busy–idle cycle, an average of $m_1(k)$ requests are served. Hence, the throughput of the nearest-neighbour arbiter is given by

$$T = \frac{m_1(k)}{B_1(k) + I_k} \ . \tag{6}$$

The expression (5) for the average response time of a request still applies.

The averages $B_i(j)$ and $m_i(j)$ can be determined recursively for all $j = 1, 2, \ldots, k$ and $i = 1, 2, \ldots, 2^j$. Consider first a sub-tree of height 1 (figure 5 (a)). A busy period may start with 2 requests or with 1 request present. In the former case, the token goes to one of the users, remains there for a service time and returns to the root, after which the sub-tree behaves as if a busy period starts with 1 request present. Thus we have

$$B_2(1) = \frac{1}{\mu} + \frac{2}{\nu} + B_1(1) \ . \tag{7}$$

In a busy period which starts with 1 request, the token goes to that user, remains there for a service time and returns to the root. If in the meantime the second user does not submit a request, the busy period ends; otherwise it continues as if a new period starts with 1 request. The probability that a thinking user does not submit a request during a service time and two node delays is equal to $\mu\nu^2/(\mu+\tau)(\nu+\tau)^2$. Hence,

$$B_1(1) = \frac{1}{\mu} + \frac{2}{\nu} + \left[1 - \frac{\mu\nu^2}{(\mu+\tau)(\nu+\tau)^2}\right] B_1(1) \ . \tag{8}$$

Similar equations allow us to determine the numbers of requests served during those busy periods:

$$m_2(1) = 1 + m_1(1) \ \ ; \ \ m_1(1) = 1 + \left[1 - \frac{\mu\nu^2}{(\mu+\tau)(\nu+\tau)^2}\right] m_1(1) \ . \tag{9}$$

Suppose now that the average busy periods and numbers of requests completed during them have been obtained for all sub-trees of height strictly less than j. Consider a sub-tree of height j, with i requests present and the privilege token at the root. According to the 'closely packed' assumption, as many as possible of the i requests are in one of the descendent sub-trees, say the left one, and the rest are in the other. That is, $\ell(i) = \min(i, 2^{j-1})$ requests are initially in the left descendent sub-tree and $r(i) = i - \ell(i)$ requests are in the right one. (Note that from the performance standpoint, the 'closely packed' assumption presents an optimistic situation since the token stays maximally within the left subtree and may not even need to switch to the right subtree afterwards. That move would obviously contribute with an extra delay to the busy period. In a more general case the distribution of requests may not be 'closely packed' in one subtree. It is therefore possible to put a pessimistic bound, $\ell(i) = \lceil \frac{i}{2} \rceil$, assuming that the requests are equally divided betwen the left and right subtrees. Obviously, it remains that $r(i) = i - \ell(i)$.) The token goes to the left descendent, remains in that sub-tree until it is empty, then returns to the root. If, during that time, s of the thinking users in the right sub-tree submit requests, the continuation is equivalent to a busy period with $r(i) + s$ initial requests. Denoting the probability of the latter occurrence by $\alpha_{i,s}$, we can write a recurrence equation for $B_i(j)$:

$$B_i(j) = B_{\ell(i)}(j-1) + \frac{2}{\nu} + \sum_{s=0}^{2^{j-1}-r(i)} \alpha_{i,s} B_{r(i)+s}(j) \ ; \ i = 1, 2, \ldots, 2^j \tag{10}$$

(if $r(i) = 0$, the term with $s = 0$ in the right-hand side is 0 by definition).

This is not a linear recurrence, because the probabilities $\alpha_{i,s}$ depend on $B_{\ell(i)}(j-1)$. To estimate those probabilities, we make another approximation, namely we treat the time that the token spends in the left descendent sub-tree, together with the two propagation delays of getting there and back, as being distributed exponentially. The parameter of that distribution, η, is obtained from

$$\frac{1}{\eta} = B_{\ell(i)}(j-1) + \frac{2}{\nu} \, . \tag{11}$$

The probability that a given user in think state will submit a request during an exponentially distributed interval with parameter η, is equal to $\tau/(\eta + \tau)$. This implies that the probabilities $\alpha_{i,s}$ are Binomial:

$$\alpha_{i,s} = \left(\begin{array}{c} 2^{j-1} - r(i) \\ s \end{array} \right) \left(\frac{\tau}{\eta + \tau}\right)^s \left(1 - \frac{\tau}{\eta + \tau}\right)^{2^{j-1} - r(i) - s} \, , \tag{12}$$

for $s = 0, 1, \ldots, 2^{j-1} - r(i)$.

The recurrence equations for the number of completed requests during a busy period are:

$$m_i(j) = m_{\ell(i)}(j-1) + \sum_{s=0}^{2^{j-1}-r(i)} \alpha_{i,s} m_{r(i)+s}(j) \ ; \ i = 1, 2, \ldots, 2^j \, , \tag{13}$$

with the same definitions for $\ell(i)$, $r(i)$ and $\alpha_{i,s}$.

The throughput and average response time for a nearest-neighbour tree arbiter with 2^k users can now be computed by means of (6) and (5), together with the above recurrences.

4 Numerical experiments

In this section we present the experimental results obtained for the analytical and simulation-based models of the nearest-neighbour and FIFO arbiters. The analytical results have been obtained by iterative solution of the linear equation systems (using the NAG software) that are generated by the recurrences shown in the previous section. A Fortran-77 program accessing the NAG subroutines accepts as its input the value of the size of the tree n and set of values for the three main statistical parameters of the system: request arrival (think) rate τ, privilege release (service) rate μ and interstage propagation rate ν. These results have been checked against those from simulation, and appeared to be within acceptable range of accuracy (less than 10%).

Note the following arrangements in the experiments:

- In all experiments, we used normalised value $\mu = 1$ for convenience;

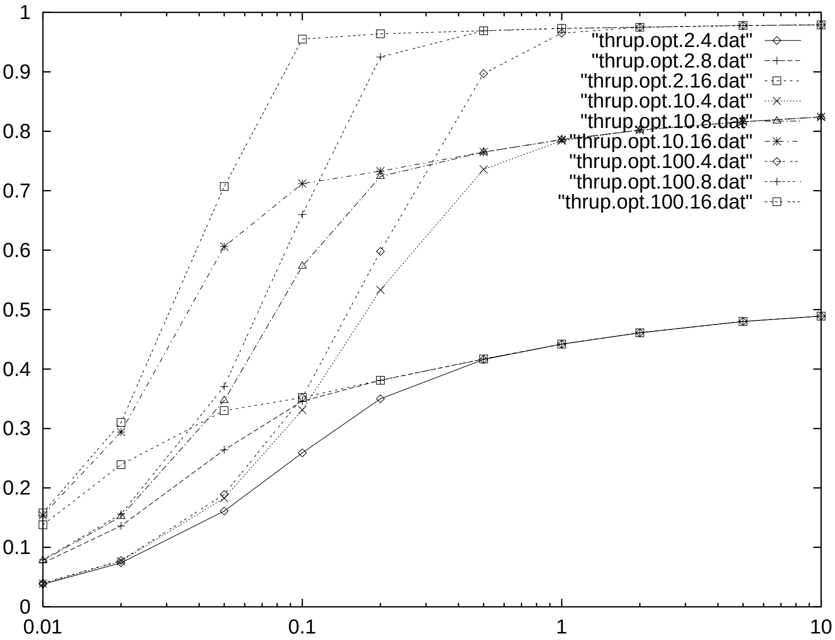

Figure 6: Nearest-neighbour arbiter throughput versus user think rate ($n = 4, 8, 16$ and $\nu = 2, 10, 100$).

- When running computations for the FIFO model we had to "adjust" the value of its interstage propagation delay to allow for the fact that the logic implementation of a nearest-neighbour arbiter cell is more complex (and hence creates greater delay) that of a FIFI arbiter. We therefore applied an appropriate set of scaling factors, e.g., $\nu_F = 1.2\nu$, where ν_F is the "adjusted", by 1.2, value for the FIFO case.
- The values of τ and ν have been changed effectively in a logarithmic scale to cover most cases concerned with the *request load* and *internal delays* in the arbiter.

Table 1 shows the data obtained for n-user ($n = 4, 8, 16$) arbiters under the assumption $\nu_F = 1.2\nu$. This data clearly shows that the ratio T/T_F between the throughputs of the nearest-neighbour arbiter and the FIFO one increases above 1 when the request flow reaches some critical point; e.g., for $n = 4$ at $\nu = 10$, such a point is $\tau = 0.2$. At the same time, as the speed of the arbiter circuit decreases (ν becomes smaller) the effect of nearest-neighbour is more apparent. E.g., for $n = 8$ and $\tau = 0.2$ the ratio T/T_F increases from 1.03 at $\nu = 100$ to 1.33 at $\nu = 2$. Then, under higher request flow, for $\tau = 1$, the ratio T/T_F goes from 1.02 at $\nu = 100$ to 1.55 at $\nu = 2$.

The above-mentioned gain becomes more impressive as the size of the arbiter grows from $n = 4$ to $n = 8$ and then to $n = 16$. E.g., for $\tau = 1$ and $\nu = 10$ the ratio T/T_F is 1.05 at $n = 4$, 1.17 for $n = 8$ and for $n = 16$. This is explained by the fact that in a deeper cascaded structure, at a high request rate, the overhead of the privilege travelling up and down the whole tree is apparent. At the same time it is interesting to note that that the throughput ratios T/T_F for $n = 8$ and $n = 16$ differ (in favour of $n = 16$) only under relatively low think rates τ (and greater propagation delays $1/\nu$), where the effect of saving on travelling between the stages in the nearest-neighbour arbiter is more apparent. Otherwise, for the values of $\tau > 0.2$, the size of the arbiter does not seem to affect the ratio.

The plottings of the throughput against think rate τ for the three cases $n = 4, 8, 16$, at $\nu = 2$ (three bottom plots), $\nu = 10$ (three middle plots) and $\nu = 100$ (three top plots), are illustrated in Figure 6.

The relationship between the throughput ratio T/T_F and τ, again for $n = 4, 8, 16$ at $\nu = 2, 10, 100$ and $\nu_F = 1.2\nu$ is plotted in Figure 7. This plottings show that at low propagation delays $\nu = 100$, for all three values of n, we do not gain anything form the nearest-neighbour technique. However, as the propagation delay increases, even at the level of $\nu = 10$, the nearest-neighbour arbiter begins to save on the server movement at reasonable think rates, e.g. between 0.1 and 0.5.

The plottings of the throughput ratio T/T_F against the interstage propagation delay $\frac{1}{\nu}$ ($\nu_F = 1.2\nu$) for $n = 4, 8, 16$ at $\tau = 0.1, 0.2$ and $\tau = 1, 5$ are shown in Figure 8 and Figure 9, respectively.

We have also obtained data illustrating the effect of the variable scaling factor s in $\nu_F = s\nu$ for

τ	ν	T	T_F	T/T_F	T	T_F	T/T_F	T	T_F	T/T_F
		$n=4$			$n=8$			$n=16$		
0.01	100	0.040	0.040	1.00	0.079	0.079	1.00	0.158	0.158	1.00
0.01	50	0.040	0.040	1.00	0.079	0.079	1.00	0.157	0.158	0.99
0.01	20	0.039	0.040	0.99	0.079	0.079	1.00	0.156	0.158	0.99
0.01	10	0.039	0.040	0.99	0.078	0.079	0.99	0.154	0.157	0.98
0.01	5	0.039	0.039	1.00	0.077	0.078	0.99	0.149	0.156	0.96
0.01	2	0.038	0.039	0.99	0.074	0.076	0.97	0.138	0.150	0.92
0.02	100	0.078	0.078	1.00	0.156	0.156	1.00	0.310	0.311	1.00
0.02	50	0.078	0.078	1.00	0.156	0.156	1.00	0.308	0.310	0.99
0.02	20	0.078	0.078	1.00	0.154	0.155	0.99	0.302	0.308	0.98
0.02	10	0.077	0.078	0.99	0.152	0.154	0.99	0.294	0.304	0.97
0.02	5	0.076	0.077	0.99	0.148	0.152	0.97	0.278	0.295	0.94
0.02	2	0.074	0.075	0.99	0.136	0.143	0.95	0.239	0.286	0.92
0.05	100	0.189	0.189	1.00	0.371	0.371	1.00	0.707	0.699	1.01
0.05	50	0.188	0.188	1.00	0.368	0.368	1.00	0.695	0.690	1.01
0.05	20	0.186	0.187	0.99	0.360	0.364	0.99	0.660	0.660	1.00
0.05	10	0.183	0.187	0.99	0.347	0.354	0.98	0.606	0.603	1.00
0.05	5	0.177	0.181	0.99	0.323	0.331	0.98	0.512	0.488	1.05
0.05	2	0.161	0.168	0.96	0.264	0.255	1.03	0.330	0.286	1.15
0.1	100	0.352	0.352	1.00	0.660	0.652	1.01	0.955	0.937	1.02
0.1	50	0.349	0.350	1.00	0.649	0.642	1.01	0.919	0.909	1.01
0.1	20	0.342	0.345	0.99	0.619	0.612	1.01	0.828	0.797	1.04
0.1	10	0.331	0.336	0.99	0.573	0.560	1.02	0.712	0.666	1.07
0.1	5	0.311	0.318	0.98	0.496	0.465	1.07	0.560	0.5	1.12
0.1	2	0.259	0.268	0.96	0.346	0.284	1.22	0.352	0.286	1.23
0.2	100	0.598	0.595	1.01	0.925	0.896	1.03	0.964	0.952	1.01
0.2	50	0.59	0.588	1.00	0.897	0.864	1.04	0.930	0.909	1.02
0.2	20	0.568	0.568	1.00	0.823	0.776	1.06	0.843	0.8	1.05
0.2	10	0.533	0.535	1.00	0.724	0.658	1.10	0.733	0.667	1.10
0.2	5	0.473	0.476	1.00	0.587	0.498	1.18	0.588	0.5	1.18
0.2	2	0.350	0.344	1.02	0.381	0.286	1.33	0.381	0.286	1.33
0.5	100	0.897	0.882	1.02	0.969	0.952	1.02	0.966	0.952	1.02
0.5	50	0.876	0.860	1.02	0.941	0.909	1.04	0.941	0.909	1.04
0.5	20	0.817	0.8	1.02	0.865	0.8	1.08	0.865	0.8	1.08
0.5	10	0.736	0.714	1.03	0.765	0.667	1.15	0.765	0.667	1.15
0.5	5	0.614	0.584	1.05	0.626	0.5	1.25	0.626	0.5	1.25
0.5	2	0.416	0.373	1.12	0.417	0.286	1.46	0.417	0.286	1.46
1	100	0.965	0.954	1.01	0.973	0.952	1.02	0.973	0.952	1.02
1	50	0.941	0.926	1.02	0.947	0.909	1.04	0.947	0.909	1.04
1	20	0.874	0.849	1.03	0.878	0.8	1.10	0.878	0.8	1.10
1	10	0.784	0.745	1.05	0.786	0.667	1.17	0.786	0.667	1.17
1	5	0.652	0.598	1.09	0.653	0.5	1.31	0.653	0.5	1.31
1	2	0.442	0.375	1.18	0.442	0.286	1.55	0.442	0.286	1.55
2	100	0.975	0.966	1.01	0.975	0.952	1.02	0.975	0.952	1.02
2	50	0.952	0.936	1.02	0.952	0.909	1.05	0.952	0.909	1.05
2	20	0.889	0.856	1.04	0.889	0.8	1.11	0.889	0.8	1.11
2	10	0.802	0.750	1.07	0.802	0.667	1.20	0.802	0.667	1.20
2	5	0.673	0.6	1.12	0.673	0.5	1.35	0.673	0.5	1.35
2	2	0.461	0.375	1.23	0.461	0.286	1.61	0.461	0.286	1.61
5	100	0.978	0.968	1.01	0.978	0.952	1.03	0.978	0.952	1.03
5	50	0.956	0.937	1.02	0.956	0.909	1.05	0.956	0.909	1.05
5	20	0.898	0.857	1.05	0.898	0.8	1.12	0.898	0.8	1.12
5	10	0.816	0.75	1.09	0.816	0.667	1.22	0.816	0.667	1.22
5	5	0.692	0.6	1.15	0.692	0.5	1.38	0.692	0.5	1.38
5	2	0.480	0.375	1.28	0.480	0.286	1.68	0.480	0.286	1.68

Table 1: Throughputs T, T_F and their ratio (T/T_F) for variable n, τ and ν ($\nu_F = 1.2\nu$).

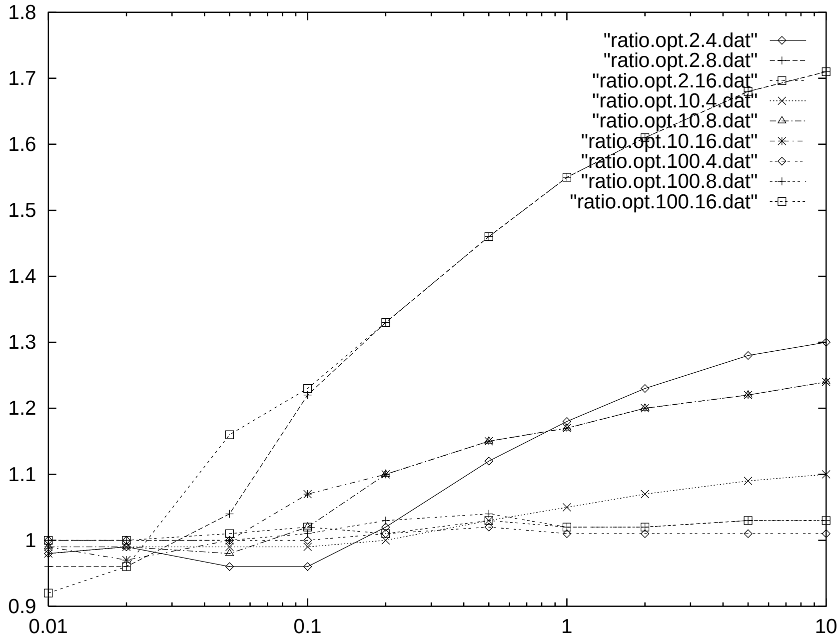

Figure 7: Throughput ratio T/T_F against think rate τ ($n = 4, 8, 16$ and $\nu = 2, 10, 100$, $\nu_F = 1.2\nu$).

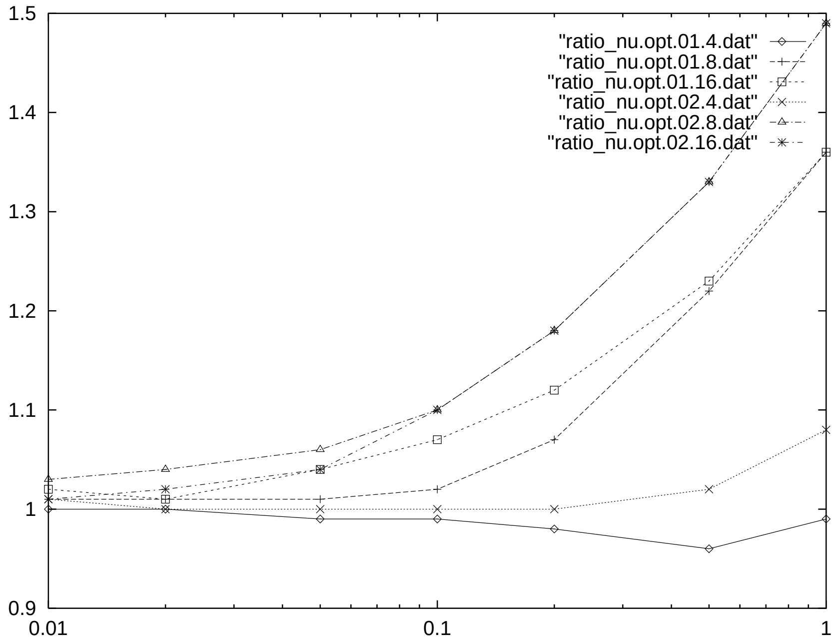

Figure 8: Throughput ratio T/T_F against interstage propagation delay $\frac{1}{\nu}$ ($n = 4, 8, 16$ and $\tau = 0.1, 0.2$).

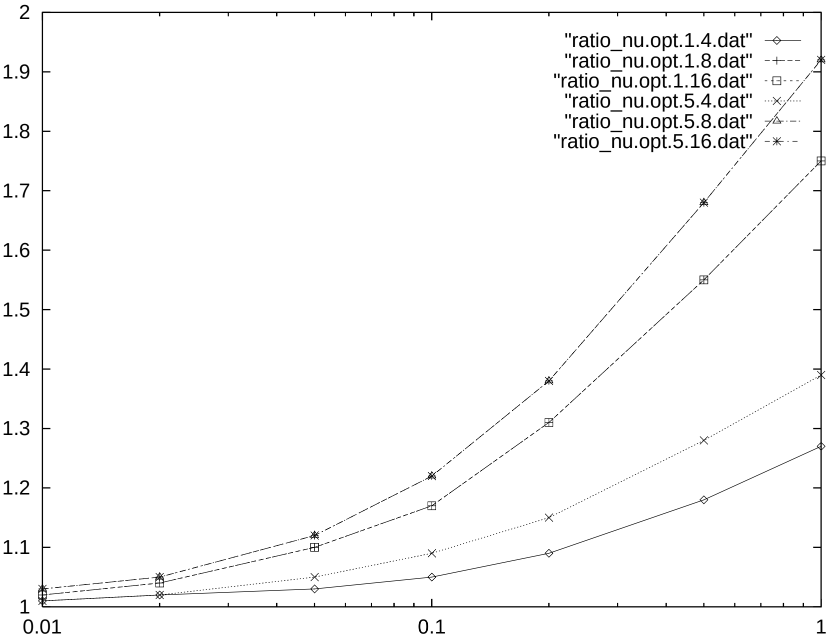

Figure 9: Throughput ratio T/T_F against interstage propagation delay $\frac{1}{\nu}$ ($n = 4, 8, 16$ and $\tau = 1, 5$).

$n = 8$. The data are plotted in Figures 10 (for $\tau = 0.1$), 11 ($\tau = 0.2$) and 12 ($\tau = 1$). The latter for example shows that under a high rate of requests ($\tau = 1$), even a fairly costly implementation of the nearest-neighbour mechanism, with scaling factor s around 2, the nearest-neighbour method can be quite advantageous.

Finally, the plottings in Figure 13 illustrate the effect of distinction between the optimistic ('closely packed' requests) and pessimistic (requests split equally between the left and right subtrees) cases. Note that as the think rate τ grows from 0.5 to 5 the difference between the bounds decreases.

5 Conclusions

We have presented a method for cascaded (tree) asynchronous arbitration which is based on the nearest-neighbour policy of privilege scheduling within a cluster of the system. This method is different from the classical approach (called FIFO here), where the privilege always returns to the top of the tree at the end of each resource acquisition by some user. Our intuitive expectation that the nearest-neighbour arbiter allows the overall increase of performance due to minimisation of the interstage transfers of the privilege has been confirmed by performance analysis. The results of analysis demonstrate that the performance gain due to the nearest-neighbour discipline against the cascaded FIFO method can be achieved for realistic values of the system's parameters, the request (think) rate and the interstate propagation delays. Even though the use of the nearest-neighbour policy negatively affects the system's fairness, one can efficiently exploit its advantages when building a system with two types of tasks. It is possible to construct the arbitration system combining the FIFO and nearest-neighbour arbitration cells [10]. Then, whenever the system reaches its request flow peaks, the nearest-neighbour method allows its specific parts, presumably 'high priority clusters", to retain the privilege until the peak is finished. At the same time, the arbiter can serve its "background clusters" during quieter periods using the FIFO method. The model for performance analysis presented here is relatively high-level and approximate (see section 3), and it does not take into account the real physical delays in potential VLSI implementations. Further work is planned on lower level performance analysis of asynchronous circuit implementations of arbiters consisting of different types of cells, FIFO, low latency, with rejection and nearest-neighbour.

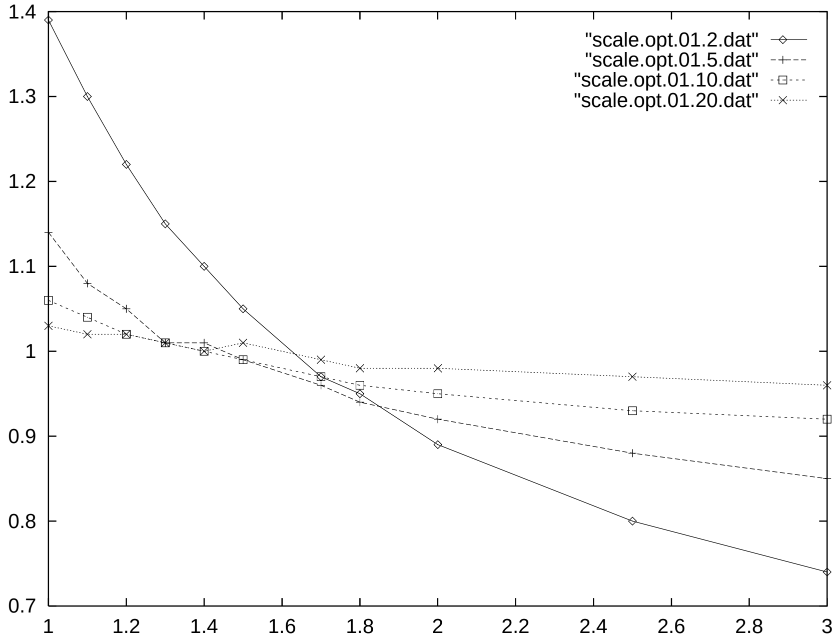

Figure 10: Plot for illustration of the effect of scaling the ν_F (variable factor s in $\nu_F = s\nu$) for $n = 8$ and $\tau = 0.1$ at $\nu = 2, 5, 10, 20$.

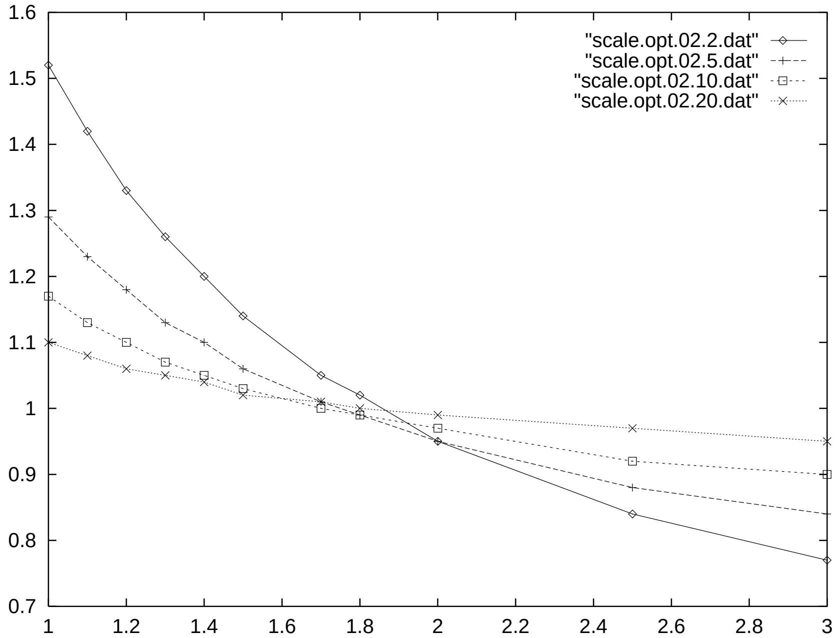

Figure 11: Plot for illustration of the effect of scaling the ν_F (variable factor s in $\nu_F = s\nu$) for $n = 8$ and $\tau = 0.2$ at $\nu = 2, 5, 10, 20$.

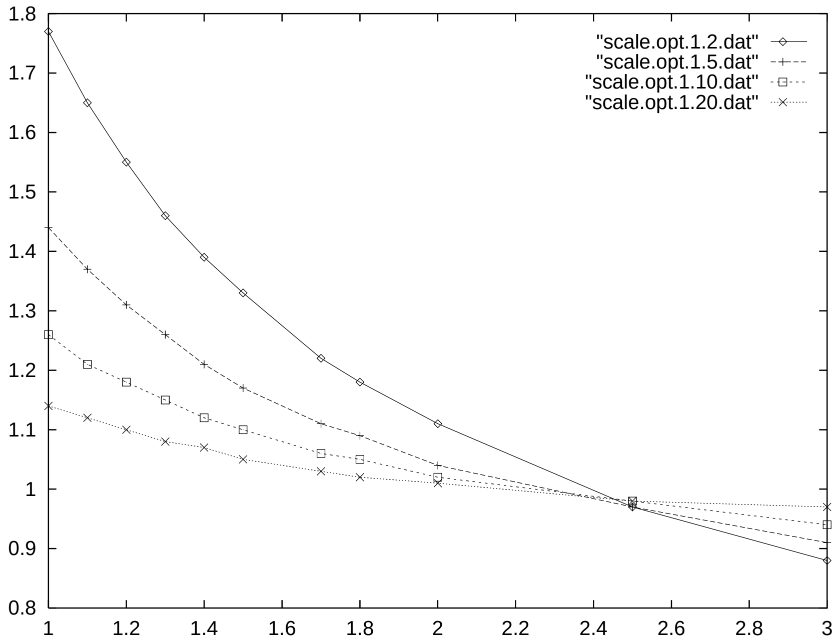

Figure 12: Plot for illustration of the effect of scaling the ν_F (variable factor s in $\nu_F = s\nu$) for $n = 8$ and $\tau = 1$ at $\nu = 2, 5, 10, 20$.

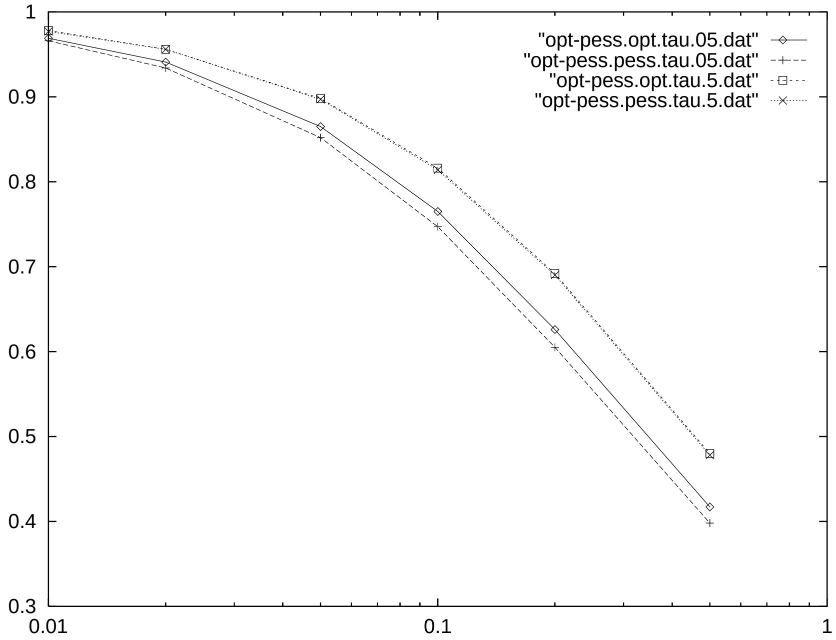

Figure 13: Plot (throughput T versus propagation delay $1/v$) to illustrate the difference between the optimistic and pessimistic models for $\tau = 0.5, 5$.

Acknowledgement

The authors would like that thank Luciano Lavagno for his paticipation in discussing the idea of a nearest-neighbour arbiter and his involvement in implementing the arbiter as a self-timed circuit (using the SIS synthesis tool) [10].

References

[1] B. Berthomieu and M.Diaz (1991), "Modeling and verification of time dependent systems using Timed Petri nets", *IEEE Trans. of Software Engineering*, **17**, 259 –273.

[2] L.N. Bhuyan (1987), "Analysis of interconnection networks with different arbiter designs", *J. Parallel and Distributed Computing*, **4**, 384–403.

[3] T.J. Chaney and C.E. Molnar (1973), "Anomalous behavior of synchronizer and arbiter circuits", *IEEE Trans. on Computers*, **C-22**, 421–422.

[4] B. Coates, A. Davis and K. Stevens (1993), "The Post Office experience: designing a large asynchronous chip", *Integration – the VLSI journal*, **15**, 341–366.

[5] M.B. Josephs and J. Yantchev (1993), "Low latency asynchronous arbiter", Patent application 9308161.0, Oxford University Computing Laboratory.

[6] I. Mitrani (1987), *Modelling of Computer and Communication Systems*, Cambridge University Press.

[7] T. Murata (1989), "Petri nets: Properties, analysis and applications", *Proceedings of IEEE*, **77**, 541–580.

[8] S.M. Nowick and D.L. Dill (1989), "Practicality of state-machine verification of speed-independent circuits", in *Proc. Int. Conf. on CAD (ICCAD'89).*

[9] C.L. Seitz (1980), "Ideas about arbiters", *Lambda*, **1**, First Quarter, 10–14.

[10] A. Yakovlev (1995), "Designing arbiters using Petri nets", in *Proceedings of the 1995 Israel Workshop on Asynchronous VLSI*, Nof Genossar, Israel, VLSI Systems Research Center, Technion, Haifa, Israel, 178–201.

[11] A. Yakovlev, A. Petrov and L. Lavagno (1994), "A Low Latency Asynchronous Arbitration Circuit", *IEEE Trans. on VLSI Systems*, **2**, 372–377.