References

[1] A. Agarwal, R. Simoni, J. Hennessy and M. Horowitz. "An evaluation of directory schemes for cache coherence". *In Proc. of the 15th Annual Int. Sym. on Computer Architecture*, p 280-289, June 1988

[2] J.A.C. Bogaerts, R. Divia, H. Muller and J.F. Renardy, *SCI-based Data Acquisition Architectures*

[3] J.W. Bothner and T.I. Hulaas, *Various Interconnects for SCI-based Systems*

[4] L. Censier and P. Feutrier. "A new solution to coherence problems in multicache systems". *IEEE Trans. on Computers*, c-27(12):112-118 December 1978.

[5] A. Gupta, W-D. Weber and T. Mowry, *Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Systems*

[6] S. Gjessing, D.B. Gustavson, James R Goodman, David V James, Ernst H Kristiansen. "The SCI Cache Coherence Protocol". In *Scalable Shared Memory Multiprocessors*, M. Dubois and S. Thakkar, eds., Kluwer academic Publishers, Norwell, Mass. 1992

[7] The IEEE. "IEEE P1596 Standard Specification". IEEE Publication, 1989.

[8] A.G. Greenberg and I.Mitrani. "Analysis of Snooping Caches". *Proc. of Performance 87, 12th Int. Symp. on Computer Performance*, Brussels, December 1987.

[9] E. Ametistova and I. Mitrani. "Modelling and Evaluation of Cache Coherence Protocols in Multiprocesssor Systems". *Proc. of U.K. Performance Engineering Workshop*, Loughborough 1993.

[10] P.G. Harrison and N.M. Patel. *Performance Modelling of Communication Networks and Computer Architectures*. Addison-Wesley. 1993

[11] J. Hennessey and D. Patterson. "Computer Architecture: a Quantitative Approach", Morganb Kaufmann, 1990.

[12] D.V. James. "SCI Cache Coherence". In *Cache and Interconnect Architectures in multiprocessors*, M. Dubois and S. Thakkar, eds., Kluwer academic Publishers, Norwell, Mass. 1990

[13] L. Kleinrock. *Queueing Systems Volume II: Computer Applications*. John Wiley & Sons. 1976

[14] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy. "The Directory-based Cache Coherence Protocol for the DASH Multiprocessor". sym on Computer Architecture. *In Proc. of the 17th Annual Int. Sym. on Computer Architecture*. June 1990

[15] S.L. Scott, James R. Goodman and M.K. Vernon. "Performance of the SCI ring". *In Proc. of the 19th Annual Int. Sym. on Computer Architecture*. May 1992.

[16] B. Spiers and H. Wiggers. "Performance Analysis of SCI Cache Coherency Protocol". IEEE P1956 doc124. 1989.

[17] T.M. Cole (ICL Ltd., Bracknell, Berkshire, U.K.). Private communication, 1994.

tween shared and private data, nor between different classes of shared data, e.g. locks and read-only shared structures. In practice, the access patterns to these are far from uniform in practice [17] and we are currently investigating enhancements to the workload model to take account of private data and the various shared-memory regions which are observed in real applciations. These refinements involve significant alterations to the line states and message transmission profiles.

Thus far we have not had the opportunity to validate the model, although the preliminary graphs produced to date do behave as might be expected. In the absence of a real SCI system against which to perform this validation we are now undertaking a detailed simulation exercise in which SPLASH programs are executed on multi-threaded implementations of the DLX simulator [11], suitably modified to interface to an SCI simulator at the back end. This "execution-driven" simulation should provide a reliable model against which to validate the analytical model, although like most simulations of its kind is likely to suffer very long execution times.

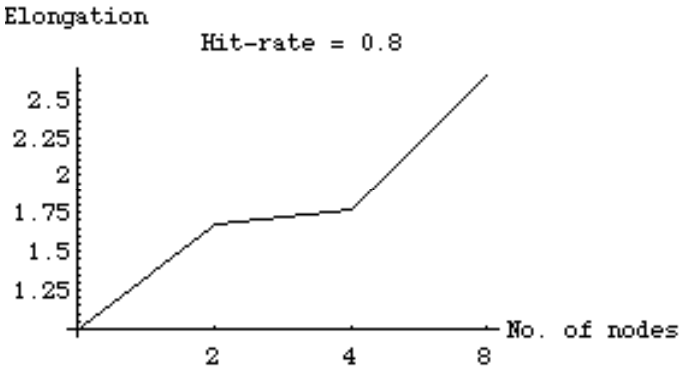


Figure 10: Elongation as a function of K ($\beta = 0.8$)

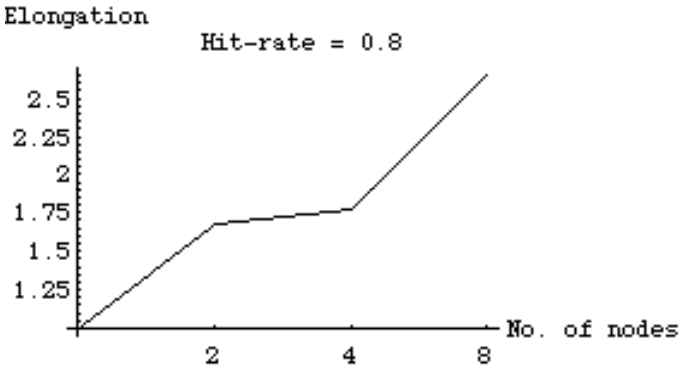


Figure 11: Elongation as a function of K ($\beta = 0.8$)

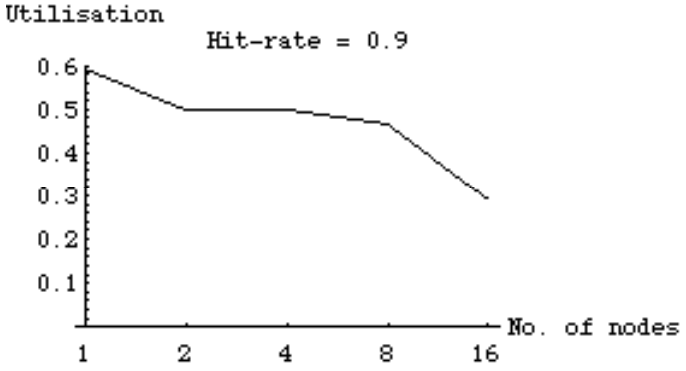


Figure 8: Utilisation as a function of K ($\beta = 0.9$)

These preliminary results show that for the parameterisation given, the eolongation is very sensitive to the cache hit rate. At $\beta = 0.9$, for example, the elongation is slight up to eight processors, but thereafter jumps dramatically as the number is increased to 16. There is a corresponding drop in the processor utilisation at the same point on Figure 8 and is suggestive of a bottleneck appearing in the SCI ring. For lower hit rates a similar jump is observed, but at lower processor numbers. This we would expect since the lower the hit rate the more SCI traffic is generated per memory request.

A complete set of graphs for a realistic machine based upon the Dolphin chip set timings will be produced for the final version of the paper.

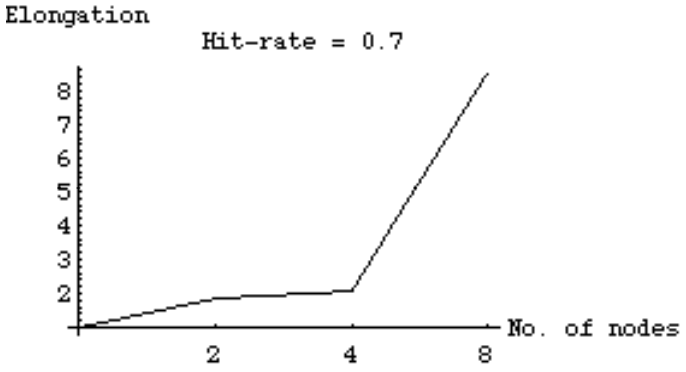


Figure 9: Elongation as a function of K ($\beta = 0.7$)

5 Summary

In this paper we have developed a new analytical model of the IEEE Standard Coherent Interface running on a unidirectional ring architecture. We have essentially combined two existing modelling approaches: one for bus-based shared-memory systems in which the processor utilisation is derived indirectly from the equilibrium cache line state probabilities, and the other for analysing slotted rings using an underlying M/G/1 queueing model.

We have presened some preliminary numerical results for the processor utilisation and SCI overheads, for a particular parameterisation, as the number of processors is increased.

The current model assumes a uniform memory access model and makes no distinction be-

4 Results and Conclusions

We now present some preliminary numerical results for a contrived system in which the cache access time is assumed to be 100ns and the memory access time 480ns. The mean processor think time is assumed to be equivalent to two cache memory cycles, i.e. 200ns. Messages that do not contain data are assumed to be 16 bytes long, messages that do contain data are taken to be 64 bytes long, in keeping with our simple SCI model. We take the link speed to be 1Gb/sec. The average service times at a node is thus $16/10^9$ sec for short messages (this defines t_{short}) and $64/10^9$ sec for long messages.

We shall just assume that the service time for short messages is exponentially distributed so that its second moment is given by t^2_{short}.

Figures 6– 8 show the variation in system power (i.e. processor utilisation) for various cache hit rates as the number of processors is increased.

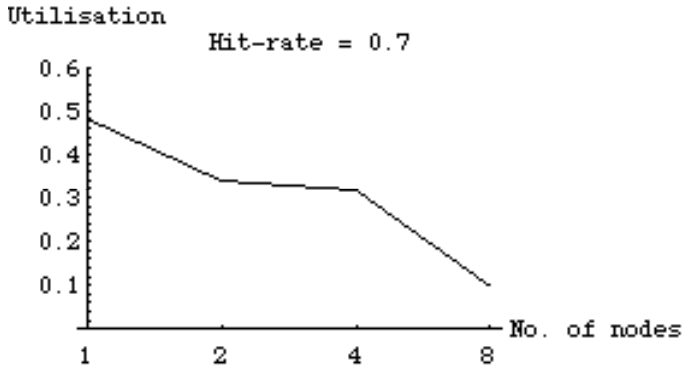


Figure 6: Utilisation as a function of K ($\beta = 0.7$)

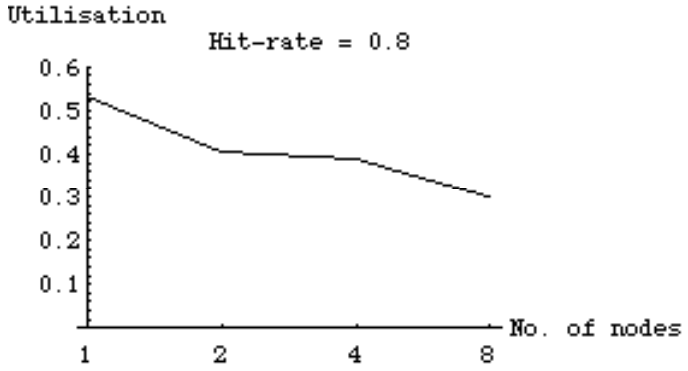


Figure 7: Utilisation as a function of K ($\beta = 0.8$)

Figures 9– 11 illustrate the variation in *elongation* as the number of processors is increased. The elongation is a measure of the overhead on memory access time that is imposed by SCI. It is calculated as the ratio of the actual memory service time to that of an "ideal" system in which all cache misses are serviced in exactly one memory access time. That is:

$$elongation = \frac{T}{\beta\, W_{cache} + (1-\beta)\, W_{mem}}$$

where Q_t and Q_r are respectively the queueing times of messages in the transmit buffer and ring buffer. These are determined by subtracting an average service time from the mean waiting time for each:-

$$\begin{aligned} Q_t &= W_t - (p_s + M(1-p_s))\, t_{short} \\ Q_r &= W_r - (p'_s + M(1-p'_s))\, t_{short} \end{aligned}$$

In order to derive the mean time to service a memory request, T we note that some requests can be service immediately within the local cache of the requesting processor whilst others will require SCI actions to take place. In the latter case, we must consider not only the message traffic generated by each action, together with the message transmission times, but also the number of cache and memory accesses required to service it. The cache and memory access counts are given in the tables C and M defined earlier for each action. Defining t_{cache} and t_{mem} to be the cache and memory access times respectively, we obtain:

$$\begin{aligned} T &= \sum_s (\sum_a q_s \delta_{s,a}\, (S_{s,a}\, T_s + L_{s,a}\, T_l + C_{s,a}\, t_{cache} + M_{s,a}\, t_{mem}\,) \\ &\quad + (1 - \sum_s \delta_{s,a})\, t_{cache}) \end{aligned}$$

From this we can finally determine the processor utilisation, or "system power":

$$\pi_0 = \sum_a \frac{\frac{1}{\tau}}{\frac{1}{\tau} + T}$$

This concludes our model.

(short) echo packet. This echo packet is circulated through the remaining $K/2$ buffers back to the transmitter. Hence:

$$\begin{aligned} p'_s &= \frac{(K-1)\ \lambda_s + (K/2)\ \lambda_l}{(K-1)\ \lambda_s + (K/2)\ \lambda_l + ((K-2)/2)\ \lambda_l} \\ p'_l &= \frac{((K-2)/2)\ \lambda_l}{(K-1)\ \lambda_s + (K/2)\ \lambda_l + ((K-2)/2)\ \lambda_l} \end{aligned}$$

If we define $X \in \{1, M\}$ to be the number of short message equivalents in each message in the aggregate stream emanating from each node then the n^{th} moment of X is given by:

$$M_n = p_s + M^n p_l$$

Similarly for the number of short message equivalents in each each message from the ring we obtain the n^{th} moment as:

$$M'_n = p'_s + M^n p'_l$$

We use these to derive the waiting times below.

In order now to use Cobham's formulae we shall make the assumption that the stream of messages arriving at the ring buffer are *Poisson*. They are not in general, but the assumption is common and often found to be reasonable, especially in representing superposed traffic.

If t_{short} is the time taken for a link to service the transmission of a short message thenthe mean message service time is $M_1\, t_{short}$ and we define:

$$\begin{aligned} \rho_t &= \lambda_t M_1 t_{short} \\ &= \lambda_t(\lambda_s/\lambda_t + M\,\lambda_l/\lambda_t)t_{short} \\ &= t_{short}(\lambda_s + M\ \lambda_l) \end{aligned}$$

So we define

$$\begin{aligned} \rho_r &= \lambda_r M'_1 t_{short} \\ &= t_{short}((K-1)\ \lambda_s + (K/2)\ \lambda_l + M((K-2)/2)\ \lambda_l) \end{aligned}$$

We then obtain:

$$\begin{aligned} W_t &= \frac{W_0}{(1-\rho_r)(1-\rho_r-\rho_t)} \\ W_r &= \frac{W_0}{1-\rho_r} \end{aligned}$$

where

$$\begin{aligned} W_0 &= \frac{1}{2}(t^2_{short}\ M_2\ \lambda_t + t^2_{short}\ M'_2\ \lambda_r) \\ &= \frac{1}{2}t^2_{short}((\lambda_s + \lambda_l\ M^2) + ((K-1)\lambda_s + (K/2)\ \lambda_l \\ &\quad +(K-2)/2)\ \lambda_l M^2) \end{aligned}$$

From this we can now determine the mean transmission time of short and long messages to be:

$$\begin{aligned} T_s &= K t_{short} + Q_t + (K-1)Q_r \\ T_l &= (K/2)(1+M)t_{short} + Q_t + (K-1)Q_r \end{aligned}$$

memory accesses respectively, indexed by the line state and the action initiated. Thus, for example, $C_{5,DA} = 4$ and $L_{12,AD} = 0$.

The rate of generation of short and long messages to the ring, λ_s and λ_l say, are then given by:

$$\lambda_s = \tau \sum_{s,a} q_s \ \delta_{s,a} \ S_{s,a}$$

$$\lambda_l = \tau \sum_{s,a} q_s \ \delta_{s,a} \ L_{s,a}$$

3.10 Mean transmission time for a message

We now determine the mean transmission time of a message around the ring. For simplicity we make the assumption that all echo messages have the same length as a short message. Thus a short message issued by a transmitting node will perform one full circuit of the ring (i.e. through $K-1$ ring buffers). The receiving node will extract the incoming packet and, at the same time, pass it on as an echo packet. A long message will be converted to a short echo message once it has reached the receiving node.

It is important to remember that messages originating from the ring have priority over messages in the transmit queue originating from the node. In order to model these priorities we will use Cobham's formulae [10] to determine the mean waiting times of messages in the transmit queue and ring buffer, W_t and W_r respectively. From these we can obtain the mean transmission time of short and long message around the ring, T_s and T_l respectively.

To use Cobham's formulae we need to determine the rate at which messages are generated by the transmit queue and ring buffer of each node. We will denote these by λ_t and λ_r respectively. The total rate at which messages are generated by a node is thus:

$$\lambda_t = \lambda_s + \lambda_l$$

To simplify the discussion we will now assume that the length of a long message is a fixed multiple M of the length of a short message. The various transmission times and utilisations can then be expressed in terms of the transmission time of a short message, which we denote by t_{short}.

The proportion of short and long messages within the aggregated message stream from each node, p_s and p_l respectively, can be written in terms of the corresponding arrival rates of long and short messages:

$$\begin{aligned} p_s &= \frac{\lambda_s}{\lambda_s + \lambda_l} \\ p_l &= \frac{\lambda_l}{\lambda_s + \lambda_l} \end{aligned}$$

To determine the corresponding proportions, p'_s and p'_l, for messages emanating from the ring we must take into account the fact that long messages are converted to (short) echo messages once they have reached their destination. Thus, a short message circulates through $K-1$ ring buffers and each long message through an average of $(K-2)/2$ ring buffers before being converted to a

We will denote this table by δ so that, for example, $\delta_{5,DC} = \overline{\eta}\overline{\beta}$.

3.9 Message Streams

Each action makes the processor send and receive a certain number of long or short messages through the ring. In order to calculate the time these messages take to be transmitted we need to calculate the rate at which they are produced. Note that some messages may be destined for the local memory block of the processor and so will create no ring traffic.

The following table shows, for each action, how many short and long messages are sent by the action, and how many cache and memory accesses are involved in maintaining the sharing list(s) concerned. We will assume that the message assembly times are negligible in comparison to the ring transmission and cache/memory access times. Those messages which involve communication with memory are marked with a *; it is these messages which may (with probability $\frac{1}{K}$) by-pass the ring. The term $x*$ can therefore be read $x(K-1)/K$.

Reductions are a special case since the reducing processor sends and receives as many short messages as there are members in the sharing list behind its cached block. We estimate this by using the mean length of a non-singleton sharing list calculated above. Hence, once it has taken the head of the list, the processor generates for the deletion itself $\Delta = 2(1 + q_5/q_6)$ short messages.

	State	*Short* (S)	*Long* (L)	Cache (C)	Memory (M)
CR	7	1*	1*	1	1
AD	7	2* + 1	1	2	1
DC	12	2*	1*	1	2
	34	2*+1	1*	2	2
	5	1*+2	1*	3	1
	6	1*+1	1*	2	1
DA	12	3*+1	1	2	2
	34	3*+2	1	3	2
	5	2*+3	1	4	1
	6	2*+2	1	3	1
DR	12	3*+1+Δ	1	2+Δ	2
	34	3*+2+Δ	1	3+Δ	2
	5	2*+3+Δ	1	4+Δ	1
	6	2*+2+Δ	1	3+Δ	1
TR	5	2*+3+Δ	0	3+Δ	1
	6	2*+2+Δ	0	2+Δ	1
HR	34	Δ	0	Δ	0
IR	7	2*+1+Δ	1	2+Δ	1

Thus, for example, an AD action issues one short message to memory requesting the head of the sharing list (involving a memory access), receives one short message containing the head address, sends one further short message to the head of the list requesting a copy of the line (involving a cache access) and receives a long message containing the line (this must be written to the processor's cache and therefore involves one further cache access). An access in this sense refers to a general read-modify-write cycle of the cache or memory.

This table implicitly defines the non-zero entries of four new tables which we shall call S, L, C and M representing the number of short messages, long messages, cache accesses and

- 1,2 to 6: A location in state 1,2 transits to state 6 if another cache adds to the list, which means a remote read miss on the location. This may be done by any of the K-1 other processors, again since there is only one cached copy of the block in total. The probability of this is $(K-1)\alpha(1-\beta)/KN$. In the similar transition 3,4 to 5, one must take care not to count processors that are already in the list, which explains the factor: $K - L_e$.

- 5 to 1,2: A location in state 5 may transit to state 1,2 if the local processor needs the location for another memory block, i.e. after a read miss on a home block or a write on a block mapping to the same cache address. This happens with probability $(\alpha(1-\beta)(1-\eta)+(1-\alpha))/n$.

- 5 to 6: A location in state 5 may transit to state 6 if the tail has to delete itself from the list and the Mid was just before the tail. This happens only if the tail issues incurs a (read or write) miss on the location.

- to 7: A location turns to Invalid if a remote processor which does not already have a copy of the block issues a write miss to the same location or if any other processor in the list performs a write hit. The former happens with probability $(K-L)(1-\beta)(1-\alpha)/KN$ and the latter with probability $(L-1)\beta(1-\alpha)/n$ where L is the mean length of the sharing list. The value of L is different depending whether the location is in the middle of a sharing list or at the head/tail.

- from 7: An invalid location becomes a (valid) sharing list head as soon as the processor issues a read or write (miss) to a block mapping to that cache address.

3.7 Balance equations

The balance equations can be derived from the transition rates in the usual way—see for example [10]. These equations are not linear, but define a *fixed point* on the vector $[q_{12}, q_{34}, q_5, q_6, q_7]$ and so can be solved iteratively.

3.8 Probabilities of each action

With the q_j now known, or at least the steady-state probabilities of the necessary aggregate states, we can calculate the probability that a processor emerging from a think state generates each action (we assume that the equilibrum probability distribution of the states is the same at such instants).

For the actions including a DELetion, one has to distinguish the cases according to the state of the location, because the same kind and number of messages is not sent according to the cases as will be shown in the next section. The probability of each action is given in the following table (note that the notation $\overline{x}$ denotes $1-x$):

	Actions							
State	CR	AD	DC	DA	DR	TR	HR	IR
1,2	0	0	$\overline{\eta}\overline{\beta}$	$\alpha\eta\overline{\beta}$	$\overline{\alpha}\eta$	0	0	0
3,4	0	0	$\overline{\eta}\overline{\beta}$	$\alpha\eta\overline{\beta}$	$\overline{\alpha}\eta$	0	$\overline{\alpha}\beta$	0
5	0	0	$\overline{\eta}\overline{\beta}$	$\alpha\eta\overline{\beta}$	$\overline{\alpha}\eta$	$\overline{\alpha}\beta$	0	0
6	0	0	$\overline{\eta}\overline{\beta}$	$\alpha\eta\overline{\beta}$	$\overline{\alpha}\eta$	$\overline{\alpha}\beta$	0	0
7	$\overline{\eta}$	$\alpha\eta$	0	0	0	0	0	$\overline{\alpha}\eta$

These can not be determined exactly but we can approximate them by making assumptions about the lengths of sharing lists.

The mean length of non-singleton sharing lists is given by $2 + q_5/q_6$, so the length of the 'mid-zone', that is the zone between the head and the tail is q_5/q_6. This can range from 0 to infinity, but we shall assume here that it is either 0 or 1. In this case, everything happens as if there were at most three members in a list. In this case, a location in mid position is exactly between the head and the tail, and we make the approximation that $q_5 << q_6$ and hence

$$p_a \simeq 1 \text{ and } p_b \simeq 1 \tag{1}$$

The ratio q_5/q_6 then approximates the proportion of non-singleton sharing lists with three members. So

$$p_c \simeq 1 - \frac{q_5}{q_6} \tag{2}$$

Using these results we can write down the transition rates as follows—in every case, we have divided by the (default) factor $\pi_0\tau$ which is the rate at which processors leave the think state:

$1,2 \rightarrow 3,4$	$\frac{\alpha\eta(1-\beta)}{n}$
$1,2 \rightarrow 6$	$\frac{(K-1)\alpha(1-\beta)}{KN}$
$1,2 \rightarrow 7$	$\frac{(K-1)(1-\alpha)(1-\beta)}{KN}$
$3,4 \rightarrow 1,2$	$5 \rightarrow 1,2 + \frac{p_c(1-\beta)}{n}$
$3,4 \rightarrow 5$	$\frac{(K-L_e)\alpha(1-\beta)}{KN}$
$3,4 \rightarrow 7$	$(1-\alpha)\left[\frac{(K-L_e)(1-\beta)}{KN} + \frac{(L_e-1)\beta}{n}\right]$
$5 \rightarrow 1,2$	$\frac{[\alpha(1-\eta)(1-\beta)+(1-\alpha)]}{n}$
$5 \rightarrow 3,4$	$\frac{(1-\beta)[\alpha\eta+(1-\beta)p_a]}{n}$
$5 \rightarrow 6$	$\frac{p_b(1-\beta)}{n}$
$5 \rightarrow 7$	$(1-\alpha)\left[\frac{(K-L_m)(1-\beta)}{KN} + \frac{(L_m-1)\beta}{n}\right]$
$6 \rightarrow 1,2$	$3,4 \rightarrow 1,2$
$6 \rightarrow 3,4$	$1,2 \rightarrow 3,4$
$6 \rightarrow 7$	$3,4 \rightarrow 7$
$7 \rightarrow 1,2$	$\frac{1-\alpha\eta}{n}$
$7 \rightarrow 3,4$	$\frac{\alpha\eta}{n}$

The variable L_e denotes the mean length of a sharing list when there is at least one end element (i.e. a head or a tail) and L_m is the mean length of a sharing list when there is at least one middle element. Given our assumptions about sharing lists, we have $L_e = 2 + \frac{q_5}{q_6}$ and $L_m = 3$.

Some of the transitions warrant further explanation:

$q_{34} = q_3 + q_4$.

In order to derive the transition rates between states we will need the following quantities (K is the number of nodes, N is the number of blocks in each node memory and n is the number of blocks in each node cache):-

Mean number of sharing lists. This is the same as the mean number of sharing list heads, hence is given by:

$$(q_1 + q_2 + q_3 + q_4)Kn$$

Mean length of a sharing list. Similarly, this is given by:

$$\frac{q_1 + q_2 + q_3 + q_4 + q_5 + q_6}{q_1 + q_2 + q_3 + q_4}$$

Probability for a memory block to be cached. There are as many *cached* memory blocks as there are sharing lists in the caches. The number of memory blocks is K N, so assuming a uniform distribution of cache lines over the memory blocks, the probability is the ratio of the mean number of sharing lists over the total number of blocks, that is:

$$(q_1 + q_2 + q_3 + q_4)\frac{n}{N}$$

Probability for an address to be cached remotely (η). That is the probability for a block to be cached by the remote processors, so (again assuming uniformity):

$$\eta = (q_1 + q_2 + q_3 + q_4)\frac{n}{N}\frac{K-1}{K}$$

Probability for a cached location to be clean (ϵ). This probability is approximated by the proportion of clean lists among all the sharing lists.

$$\epsilon = \frac{q_1 + q_3}{q_1 + q_2 + q_3 + q_4}$$

Mean length of a non-singleton sharing list. For that, we just express the ratio of the average number of locations involved in non-singleton lists to the average number of such lists. Since a list has as many tails as heads, that is $q_6 = q_3 + q_4$, we obtain:

$$\frac{q_3 + q_4 + q_5 + q_6}{q_3 + q_4} = 2 + \frac{q_5}{q_6}$$

This follows directly from $2 + Pr\{\text{list has a middle} \mid \text{list has a tail}\}$.

Other parameters need to be known in order to write the transition rates; we denote these probabilities p_a, p_b and p_c:

- p_a is the probability for a location in state 5 (Mid) to be just after the head of the sharing list
- p_b is the probability for a location in state 5 to be just before the tail
- p_c is the probability for a head to be followed by just a tail

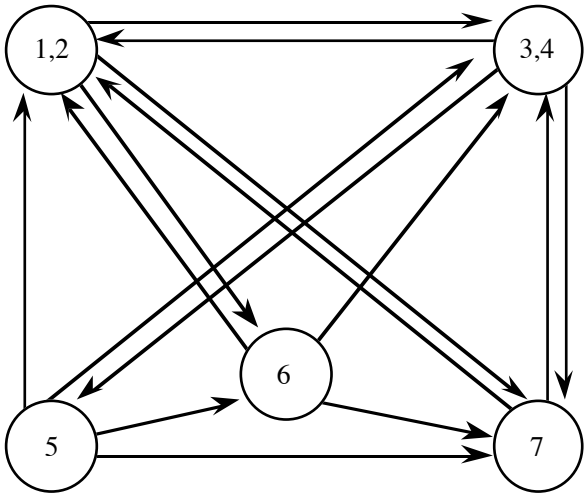


Figure 5: State Transitions

3,4 → 5	Remote read miss on the cached location
3,4 → 7	Remote write hit from another node in the sharing list, or a write miss to the associated memory location from any other node
5 → 1,2	Local read miss on a home block or a local write (hit or miss) to any location mapping to the same cache address
5 → 3,4	Local read miss on a cached block or remote delete of the head of the list when the cached block is second to the head
5 → 6	Remote delete of the tail of the list when the cached block is immediately before the tail
5 → 7	As per transition 3,4 → 7
6 → 1,2	As per transition 3,4 → 1,2
6 → 3,4	As per transition 1,2 → 3,4
6 → 7	As per transition 3,4 → 7
7 → 1,2	Local read miss on a home block or a write miss
7 → 3,4	As per transition 1,2 → 3,4

Note that in order to simplify the model we have combined states 1 and 2, and similarly states 3 and 4, into single states (1,2 and 3,4 respectively). This precludes certain transitions from the model, for example the write-back of a cache block in state 2 after displacement. We intend to assess the effect of this simplification on the accuracy of the model (if any) as part of the validation exercise—see below. In respect of this simplification we will write $q_{12} = q_1 + q_2$ and

list), and then creates a new sharing list. As before it sends 1 short and receives 1 long. The location transits to state 1 or 2.

Deletion-Addition (DA) A read miss on an already *cached* block mapping to a location in state 1–6. The processor deletes the old entry from the sharing list (involving 1 or 2 short messages as above) and adds itself to the head of the new one (3 short messages and 1 long one), the location transits to state 3 or 4.

Deletion-Reduction (DR) A write miss on a *cached* block mapping to a location in state 1–6. The processor deletes its entry from the sharing list, therefore sending 1 or 2 short messages, takes the head of the needed list sending 2 short receiving 1 long and 1 short, and then sends (and receives) short messages to reduce the list to a single sharing list sending and receiving one short message for each list entry.

Take-Reduction (TR) A write hit on a block mapping to a location in state 5 or 6. The processor deletes the old entry from the sharing list (involving 1 or 2 short messages as above), then takes the head of the same list sending 2 short messages and receiving one. It then reduces the list to a singleton as above.

Head-Reduction (HR) A write hit on a location in the processor's cache but in state 3,4. The processor sends (and receives) short messages to reduce to a singleton sharing-list as above.

Invalid-Reduction (IR) A write miss on a *cached* block mapping to a location in state 7. The processor adds itself to the list and reduces it. The location transits to state 1 or 2.

3.5 The analytical model

We need to evaluate the traffic that is generated on the ring as a consequence of of the coherency protocol. Assuming that all the processors behave in the same way, we shall derive the equilibrium cache location state probabilities. To do this we assume that the evolution of the state of a given location in a cache follows a Markov process, independent of the states of other locations. This process is irreducible, aperiodic and has a finite state space and thus has a steady-state.

Let q_j be the steady state probability that a given cache location is in state j for $j = 1, ..., 7$. It is also the average proportion of cache locations in state j.

According to the possible actions generated by a processor, we derive the instantaneous transition rates from one state to another. A transition may be created by an action of either the local processor or a remote processor. The q_j will then be obtained by solving the resulting system of balance equations.

3.6 Transition rates

Figure 5 shows the possible transitions between states. The state transitions occur as a result of the following actions:

1,2 → 3,4	Local read (miss) on an already cached block
1,2 → 6	Remote read miss on the cached location
1,2 → 7	Remote write miss on the cached location
3,4 → 1,2	As per transition 5 → 1,2 or remote deletion of the tail of the list when there are only two elements

3.3 Sharing-list updates

The following operations on sharing lists will be assumed:

Creation of a list On a cache miss, the requesting processor sends a request to the memory that contains the addressed block. If the block is in state *home* the memory returns a copy of the data and a singleton sharing list is formed.

Addition to a list On a cache miss, if there is already a sharing list on the addressed block, the memory returns a pointer to the list head and updates its sharing list pointer to the new head. The processor then sends a request to the old head; this later returns a copy of the data and updates its reverse pointer to the new head.

Deletion from a sharing list When a processor needs a cache location to store a different memory location, it will delete itself from the sharing list of which it is a part. It therefore sends a message to its predecessor with a pointer to its successor (this may be the main memory if the location is at the head of the list), and a message to its successor (if there is one) with a pointer to its predecessor, in order that they can link around the deleting block. If the location is at the tail of a list containing two or more elements, (inclulding a singleton list) only one message need be sent. If the location represents a singleton list and the contents are dirty, the cached block additionally must be written back to memory.

Reduction to a single-entry sharing list The SCI cache coherence is *invalidation based*; so, in order to write to a cached block the location must first form a singleton list by invalidating all other copies in the sharing list. Invalidation can proceed only from the head of a list, so if the location is elsewhere in a list when the write is attempted it must delete and then re-add itself before issuing the invalidations. Invalidation works by forwarding purge requests (these set the addressed locations to *invalid*); each purge request is followed by a response message which contains the address of the successor block in the sharing list.

3.4 Actions generated by a processor

In order to develop an analytical model of the system, we need to break down these operations into specific processor actions. The number and type of messages sent by a processor is determined by the action incurred.

A given processor emerges from the think state at the rate $\pi_0\tau$, and according to the state of the cache location it is accessing, can generate a number of actions. In the following, a short message is a message containing a pointer, a long one contains a copy of a complete cache/memory block.

Creation (CR) A read or write miss on a location in state 7 not cached by other processors. The processor sends 1 short message to memory and receives 1 long answer from memory. The location transits to state 1 or 2.

Addition (AD) A read miss on a location in state 7 but cached by other processors. The processor sends 1 short message to memory, receives one short message from memory, sends 1 short to the head of the list, receives 1 long from the cache. The location transits to state 3 or 4.

Deletion-Creation (DC) A read or write miss on a *home* block mapping to a location previously in state 1–6. The processor deletes the old entry from the sharing list, therefore sending 1 or 2 short messages [1] (depending on its location in, and length of, the sharing

[1] If the location was in state 2 then a *writeback* of the line will also be performed at this point involving one additional long message.

Currently the only analytical model of SCI that we are aware of is described in [15] for a ring architecture. This paper essentially represents each node on the ring by an M/G/1 queue and the bulk of the paper is concerned with setting up the model parameters. Although flow control is represented, cache coherency traffic is not. A service time of the M/G/1 queue corresponds to a round trip of the whole ring.

Although the SCI cache coherency protocol is directory-based and not snoopy-based, the method used in [8] can still be applied since each cache line undergoes similar transitions, although the states are somewhat different as they must reflect the changes in the structure of a sharing list as entries are added, deleted and displaced.

The approach we have taken in the model described below combines the two approaches by using the method in [8] to obtain the equilibrium line state probabilities and using these to determine the message arrival rates for an M/G/1 ring model as used in [15].

3 A New Model for SCI

3.1 The ring

A processor in an SCI system alternates *think periods* and periods when it waits for a transaction to be made through the ring. Let the think period have a mean of $1/\ \tau$. After a think period, the processor generates a request and this request may or may not provoke a transaction across the ring. During the time a request is processed, the processor is idle. We will aim to determine the probability, π_0, that a processor is busy doing useful work.

We will assume that a request from a processor is a read access with probability α and a write with probability $1-\alpha$. Furthermore we will take the probability of a cache hit to a valid line to be β, and thus a miss to be $1-\beta$. All blocks are assumed to be referenced equiprobably.

3.2 Possible States

A memory block may be in two possible states

1 Home The block is not cached by any processor

2 Cached The block is cached by one or more processors

A cache location may be in seven possible states

1 Only Clean The location contains a *clean* copy of a memory block (i.e. modified, different to the home memory block) and is alone in the sharing-list.

2 Only Dirty The location contains a *dirty* copy of a memory block and is again alone in the sharing list.

3 Head Clean As 1 but the location is at the head of a list containing at least two members.

4 Head Dirty As 3 but the cached copy is dirty. All members of the list hold the same dirty copy, but this is different to the home memory block.

5 Mid Valid The location is in the middle of the list (i.e. at neither the head nor tail); the cached copy may be clean or dirty.

6 Tail Valid As 5 but the location is at the tail of the sharing list; the cached copy may be clean or dirty.

7 Invalid The location contains no usable information.

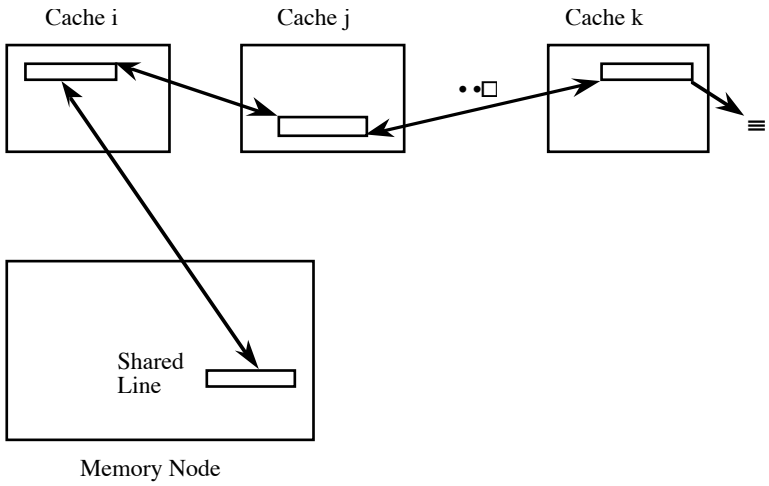


Figure 4: Sharing List Structure

the new node address at the head of the list. If the list was non-empty to start, the address of this new node must be sent to the node currently at the head of the list since it will need to update its backward pointer to reflect the update.

Line Deletion

Deletion of an element of the sharing list occurs when a line is purged from a cache. To achieve this the node sends messages to both its successor (if any) and predecessor in the sharing list informing them of the deleted node's former predecessor and successor respectively. This information is sufficient for the surrounding nodes to join up around the deleted node, although some care has to be taken when two or more nodes are being deleted simultaneously.

List Reduction

When a node wishes to write to a line, it must first invalidate all copies of the line which involves reducing the sharing list associated with the line to one element (the address of the writing node). In the simplest rendition of the protocol the writing node first makes itself the head of the sharing list and then issues purge requests to its first successor. This causes the successor node to remove itself from the list (and invalidate the corresponding cache line) and return the address of its successor in a response message. The writing node then issues a purge request to that node and the process continues until the list is reduced to a single element. The write may then proceed.

A performance model of SCI's coherency protocol must therefore take into account the nature and operation of the sharing list associated with each line since it determines the amount of traffic produced to support the protocol.

2.5 Existing Models for SCI

Existing work on shared-memory multiprocessors has led to fairly detailed performance models of bus-based multiprocessors which are capable of modelling the cache coherency traffic on the bus—see for example [8,9] which works from equilibrium cache line state probabilities that are derived from an analysis of the line state transitions.

to give the stripper time to synchronise with the clock signal in the incoming packet. This is called *elasticity* and effectively means that each packet contains one additional symbol. This is not included in the analytical mode, however.

2.4 Cache Coherency

Each node in an SCI system contains a portion of the shared memory and this is divided into a number of lines. A line defines the unit of storage associated with one cache location. Each line has associated with it a sharing list indicating which caches contain copies of that line. This is necessary in a distributed environment to replace the "snoopy" capabilities of a shared-bus architecture: coherency must now be maintained by explicit communication among line sharers rather than by having each such sharer listen to the bus to maintain the line status. This is conceptually similar to directory-based protocols [1,4] such as the Stanford DASH multiprocessor [14,5]. In SCI the sharing list for each memory block corresponds to the sharer directory in a directory-based protocol.

The coherency algorithm is supported by hardware in SCI in the form of a Cache Memory Controller which manages all updates to the memory component at a node. Figure 3 shows the architecture of a typical node in which both the node memory and top-level cache are housed in the same storage unit. The cache in this diagram contains copies of all lines cached at this node; this includes at least all those lines cached in lower-level caches attached to the system bus and possibly (probably) many more. The sharing list associated with each cached line (see below) is housed at this level indirectly.

Note that each node may contain several processors—the system bus subsystem is effectively a standard shared-memory multiprocessor, which may support conventional snoopy coherency protocols if required. This is not part of the SCI standard.

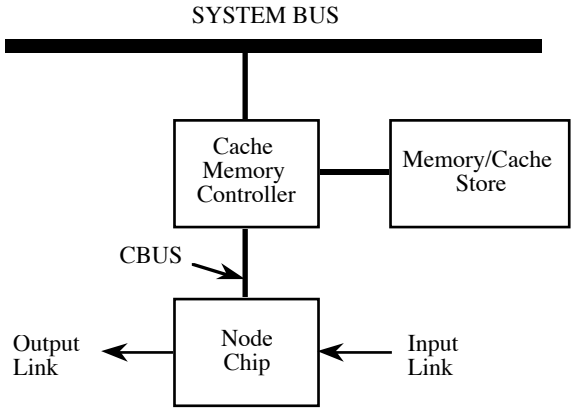


Figure 3: Node Architecture

Each line in main memory contains 64 bytes of data and a 16-bit reference to the *sharing list* for that line; this reference is the address of the first node in the sharing list which contains a copy of that line. The sharing list is doubly linked and thus requires each cache line to contain two additional 16-bit fields to store the forward and backward pointers associated with that line—see Figure 4. A sharing list has three basic operations associated with it:

List Insertion

When a read miss occurs at a node, a request is sent to the required memory line for a copy. Before issuing the copy the sharing list must be updated and this may involve communication since any existing list items will be located in different nodes. The list is updated by inserting

On the receive side, data which is targetted for the node in question is routed automatically toward the local processor via a standard bus called the CBUS. Data which is not for the local processor is routed automatically to the transmitter.

The transmitter sends data out of the node, taking data either from the receiver unit or from the processor. If the output link is busy the data is buffered automatically prior to being sent out.

A synchronisation unit is contained in the receive unit for synchronising the incoming data to an internal clock. The node chip architecture is illustrated in Figure 2(b).

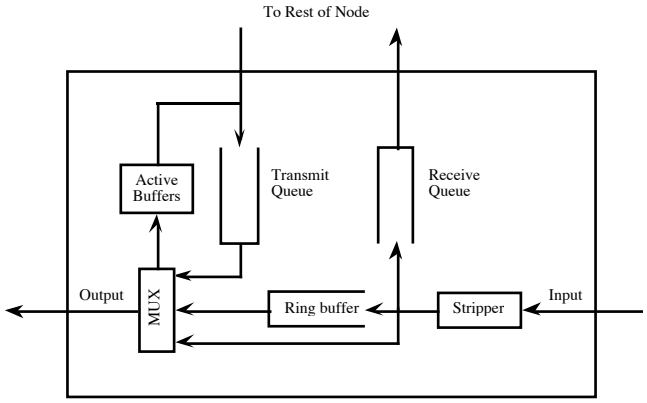


Figure 2: Node Chip Organisation

2.3 Ring Operation

Data is transferred undirectionally around the ring in packets which are either *send* packets or *echo* packets. A send packet consists of a 16-byte header and a data component of between 0 and 256 bytes. The header comprises control information, a 16-bit cyclic redundancy check and a 64-bit address, of which 16 bits identify the node and the remaining 48 bits the address within the node. An echo packet is 8 bytes long.

When a source node wishes to communicate with a destination node, a send packet is created with the appropriate header information. This is placed into the transmit queue and is sent out on the output link when there is nothing to transmit from either the input link or ring buffer. The send packet is *passed* through all intermediate nodes towards its destination. At each such node, the data is routed directly to the output link if the output link is currently idle; otherwise it will be placed into the ring buffer and sent out as soon as the currently transmitting packet has been sent. Note that link data has priority over data in the transmit queue so that in theory the ring buffer need contain no more than one packet slot.

When the data arrives at its destination it is stripped and placed into the receive queue for attention, provided there is spare capacity. The last 8 bytes of the send packet are replaced with an echo packet which continues round the ring toward the source; this will record whether or not the packet was successfully received at the destination. When it arrives back at the source (recall that data travels in one direction only) it is matched against a saved send packet in either the active buffer or transmit queue. The echo will cause the send copy to be either discarded or retransmitted depending on the echo status recorded in the packet. Note that echo packets do not store *response* information from the destination, e.g. the contents of a memory location requested from the source. Any response data required by the source will be packaged into a separate send packet targetted to the source, and this will in turn produce an echo packet for the destination.

When a node is not transmitting packet data, it emits *idle* symbols which comprise one 16-bit transmission each. An idle symbol is also sent between consecutive packet transmissions

multiprocessors but aims to avoid the performance limitations of a single communication component (the bus) by the use of a distributed interconnection network for linking the processing nodes and global shared memory. The architecture thus presents an interesting combination of a shared-memory programming model executing on what is physically a distributed-memory hardware organisation.

The SCI standard concerns the manner in which the processor(s) at one node interface to those at another. The standard addresses both a hardware communication protocol and a cache coherency protocol which ensures that each processor in the system sees a coherent (i.e. consistent) view of the shared memory.

The proposed initial implementation of SCI is based around two interface devices and a standard 16-bit data-parallel cabling mechanism for node interconnection. The global topology of an SCI-based system is not defined, but the minimal configuration is that of a "ring" in which two or more nodes are connected in a cyclic arrangement with unidirectional cables linking each node with the next node in the chain; the chain is, of course, closed to form a ring. An eight-node ring is shown in Figure 1.

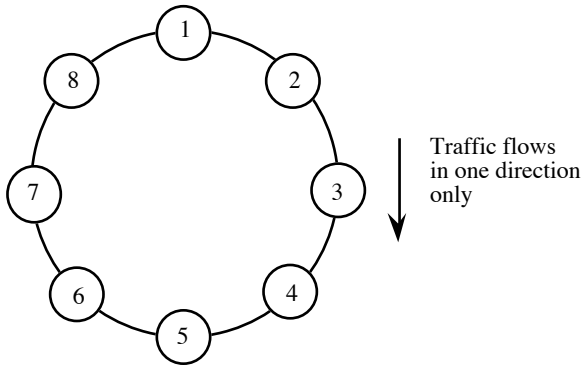


Figure 1: An Eight-node Ring

It should be stressed that the standard does not restrict the global topology to that of a ring: by using additional switching devices more complex interconnection networks can be constructed, for example hypercubes, meshes, bridged rings etc. Some examples of possible configurations are given in [3] although there are many more.

2.1 SCI Physical Interconnect

The standard module and connector is based upon a parallel data stream comprising 16-bits of data, one flag bit and a clock—a total of 18 bits. A number of physical implementations are possible. These range from CMOS prototypes which are capable of transmitting data at approximately 130MBytes per second on shielded-ribbon cable, to high-performance GaAs controllers operating at 1G-bit per second. A bit-serial transport for use with co-ax or fibre optic connections has also been defined. Other transmission mediums are possible, and allowed, however. Each will have its own line driving capabilities and will permit nodes to communicate over distances varying from a few metres to several kilometers in the case of fibre optic connections.

2.2 The Node Chip

In the proposed implementation to be studied each node in a DSMM network using SCI contains two hardware elements for supporting the SCI standard. The node chip is responsible for receiving data from and transmitting data to the network.

We present an analytical model of the IEEE P1596 Standard Coherent Interface, which is a distributed cache coherency protocol for shared memory multiprocessors. We focus upon a simplified version of the protocol running on a unidirectional ring architecture (the "default" architecture for SCI systems) and assume a uniform pattern of memory access. We identify the possible memory and cache line states and corresponding processor actions for a memory access and then derive the equilibrium line state probabilities from a transition rate matrix. These are used to determine the probabilities of a processor performing a particular action, from which the message transmission profile for each processor are derived. These traffic equations are then fed into an M/G/1 model for the ring architecture from which the mean message transmission time, mean memory access time, and processor utilisation follow. Preliminary numerical results are reported for various parameterisations of the system.

1 Introduction

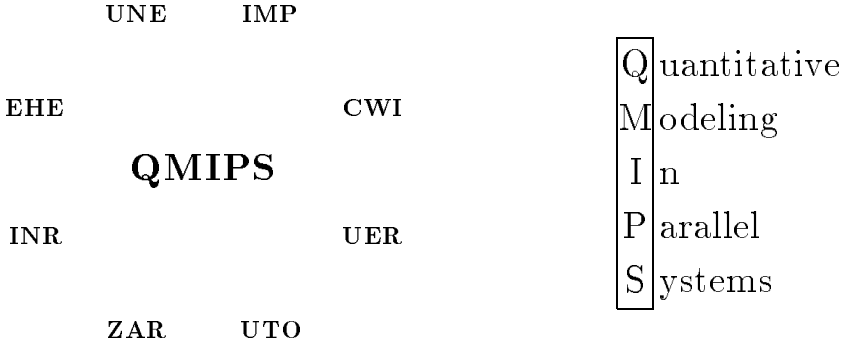
In this paper we present an analytical model of a ring-based shared-memory multiprocessor operating the IEEE P1596 Standard Coherent Interface, or SCI [7,6,16,12]. SCI is an example of an extensible cache coherency protocol. In a bus-based multiprocessor coherency is traditionally maintained by *snooping* on the bus—each attached processor contains an intelligent listening device which monitors global bus traffic and which updates locally cached data and status information in the light of the traffic observed. The problem with such systems is the fixed bandwidth of the bus, which soon saturates when the number of attached processors is increased. Distributed protocols like SCI maintain coherency by an agreed communication protocol between the component nodes. The advantage is that the communication bandwidth can be increased as the number of processors increases (since several memory accesses can proceed in parallel), thereby avoiding the bottleneck associated with bus-based systems.

A number of models of the SCI system have been produced. The most detailed of these is the full chip-level specification (in ANSI C) published as part of the IEEE standard; however running this as a simulation is intractably slow as it essentially simulates the behaviour of each SCI chip on each clock cycle. More abstract stochastic simulations have been produced, for example that of [16] although, to date, the only analytical model which exists is that of [15] which uses an M/G/1 model for analysing slotted ring networks, suitably adapted for SCI. The paper does not address SCI cache line states, however, and so does not accurately model the patterns of message traffic generated by each processing node in a real SCI system. The model described here can be viewed as an extension of this work in which a more sophisticated representation of the SCI caches themselves is built in to the model of each node.

The rest of the paper is organised as follows: Section 2 describes the core principles underlying SCI and defines a simplified version of SCI which we seek to model. This incorporates all the salient features of SCI whilst avoiding low-level details which might render an analytical model intractable. Section 3 describes out new model for SCI. This is essentially derived by combining two modelling approaches, namely that of [8,9] for single-bus systems and [15] for a simple SCI ring. Section 4 presents some preliminary numerical results from the model and Section 5 presents a summary of the paper and proposed future work.

2 Introduction to SCI

The Scalable Coherent Interface, or SCI, is a high performance interface standard for distributed shared-memory multiprocessors (DSMM). In these systems, there is a single global address space which is shared by all processors, but which is physically distributed among the processing nodes. DSMM systems thus offer the same programming model as traditional bus-based shared-memory



A Uniform-memory Access Model of a Distributed Coherent Cache System

A.J. Field, P.G. Harrison, and N. Lehovetski*
Imperial College
180 Queen's Gate
London SW7 2BZ

D W2.T1.10

22 September 1994

©IMP

Distribution Level: All partners

Approved by: Peter G. Harrison

*Summer exchange student from L'ÉcolePolytechnique,1993

The QMIPS Project: The QMIPS project is a collaborative research project supported by the CEC as ESPRIT-BRA project no 7269. It is being carried out by the following organisations: CWI (Amsterdam), EHEI (University of Paris V), Imperial College (London), INRIA (Sophia-Antipolis), University of Erlangen, University of Newcastle, University of Torino and University of Zaragoza.