Modelling and Validation of Shared Memory Coherency Protocols

Abstract

We present an analytical model of a cache coherent shared-memory multiprocessor and compare the results obtained with those from an execution-driven simulation of the same system. Our objective is to evaluate the accuracy of analytical models of this type of system, and in particular to identify the principal sources of error in the modelling of the coherency protocol. The analytical model first derives equilibrium cache line state probabilities which are then used to determine the expected long term message traffic generated by each coherency operation. These traffic rates in turn form the inputs to a queueing model of the processing nodes. Performance measurements such as processor and bus utilisations, mean queue lengths and read/write latency then follow. Validation of the model using synthetic workloads that exercise the whole of a portion of distributed memory of known size shows excellent agreement with respect to simulation. We also consider a "real" benchmark, taken from the Stanford SPLASH suite, which has interesting implications for the parameterisation of our model. The models still validate well but we speculate some sources of discrepancy due to limitations in both the analysis and the simulation, suggesting how these may be overcome.

1 Introduction

A topic of considerable interest in parallel computing in recent years has been that of shared-memory computer design and, in particular, the analysis of various caching strategies for maintaining a global coherent address space.

The simplest shared-memory machines are traditionally bus-based and benefit from the broadcasting properties of the bus which both serialises non-local accesses to shared memory and enables all memory traffic to be monitored simultaneously by all nodes. It is well known, however, that buses saturate quickly as the number of processors increases. More recent research has been concerned with scalable systems where the bus is replaced by a more general interconnection network. In these systems coherency has to be maintained by complex message passing protocols which replace the relatively straightforward "snooping" schemes used in a shared bus architecture [2]. A read or write request from a processor may now initiate a significant number of coherency messages which must be individually routed to other nodes. The interesting issue now is the extent to which this additional overhead elongates the optimal memory access time for a given protocol and hardware implementation.

At the design stage performance questions such as this have to be answered by appealing to a performance model of the proposed system. From the engineering point of view it is important to be able to predict how a particular architectural design will perform *before* the system is built. A change to the design of a processing node, for example, may affect the flow of coherency traffic and create an unexpected bottleneck. Equally, for a particular type of workload it is important to be able to explore the relative performance of different protocols or individual protocol parameters. Experiments with other hardware parameters such as cache line size are also important at the design stage.

One approach is to develop a detailed simulation of the proposed design and to drive the simulation either using address traces (synthetic or program-generated) or directly from an executing program [3]. Whilst this may yield high accuracy for particular workload/architecture specifications it is a labour-intensive task requiring often large and detailed simulation codes to be written, tested and then executed. Development times are often long and detailed simulation runs typically incur long execution times since the response to each read/write by each processor is modelled in software. The inherent complexity of the simulation also makes it prone to logical errors which in turn can lead to unreliable results.

An alternative, but complementary, approach is to develop a more abstract mathematical model of the system and to solve that model using a combination of established analytical results and numerical techniques. A number of such models have been developed for shared-memory computers, with varying levels of detail [1, 5, 10] and numerical predictions have been produced for a range of architectures and coherency protocols. However, validation has generally been with respect to stochastic simulations which make similar model assumptions and it is far from clear how effective they are at predicting accurately the behaviour of a real system running real parallel programs.

In this paper we show how a distributed cache shared memory multiprocessor can be modelled using a general-purpose analytical approach which can be adapted to different architectures and coherency protocols. Apart from the model of the coherency protocol itself, concerned with cache line states and bus and network traffic, the queueing model of the nodes introduces interesting problems in its own right. We use some numerical predictions from the model to validate it against an existing execution-driven simulation of the reference system [9]. The main validation presented here is with respect to a specially constructed benchmark program; this is a parallel C program which can be configured to use memory in a controlled way for experimental purposes. This enables us to construct a well-understood workload in order that direct quantitative comparisons between model behaviour and simulation behaviour can be undertaken. The validation exercise is inspired by a recent paper (*reference omitted*) which presents a very detailed model of the protocol [4] but with no accompanying validation. This model was developed in conjunction with a team of commercial computer architects and the importance of validation in this context cannot be understated.

Finally, we consider a "real" benchmark application, the MP3D particle simulation code from the Stanford SPLASH suite, which has interesting implications for the parameterisation of our model. Not surprisingly, validation becomes more problematic but reasonable agreement is still observed and limitations in both modelling approaches are revealed.

The rest of the paper is organised as follows: Section 2 gives an overview of the architecture used in this study, Section 3 describes the analytical model of the system, Section 4 presents a comparative analysis of the results of both models, and Section 5 gives a summary, conclusions and some pointers to future work.

2 System Architecture

The coherency protocol, which is described below, is similar to that in [12]. The machine consists of K physically identical nodes, each with a processor, a portion of the global memory and a second-level cache which is kept consistent across the machine in accordance with the coherency protocol. The communication network is taken to be contention-free and with low latency so that the communication delay is proportional to the length of the message being sent. Although such a network of any size does not exist as yet, recent work in optical communication technology [8] suggests that networks with similar properties may be feasible in the near future. In any case, the assumption is justified here since it is the model of the protocol which is more of interest than the absolute performance of the chosen architecture. It would, of course, be straightforward to incorporate an established sub-model of, for example, a crossbar network or multi-stage interconnection network but this would significantly cloud the issues whilst allowing nothing new to be learned. The assumed node structure is shown in Figure 1.

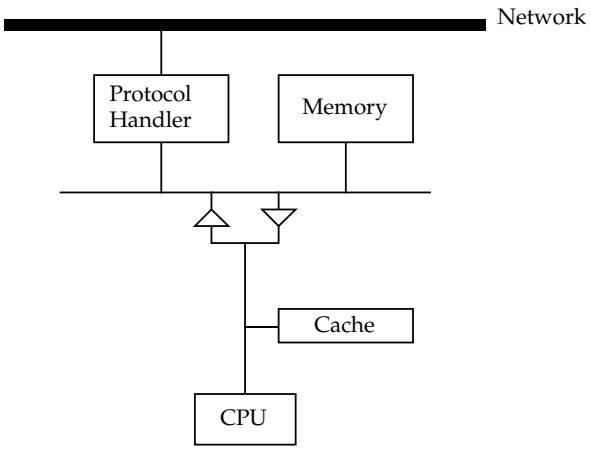

Figure 1: Node organisation

2.1 The Coherency Protocol

Memories and caches are divided into lines each comprising a fixed number of blocks of data, with associated tag information. It is assumed that there are N such shareable lines in total and each cache is assumed to have a capacity of n lines. The memory within each node is large and for reasons of cost is constructed from DRAM. Caches, on the other hand, use static RAM and special-purpose logic to achieve high-speed associative look-up.

The coherency protocol is invalidation based so that a write to a line may only proceed after all other cached copies of the line have been removed. This is achieved by a singly linked *sharing list*. The address of a given line uniquely determines the *home* node of that line; in the absence of any sharers, the line exists solely in the DRAM associated with the home node. A cache may contain a copy of a line from the local DRAM but this copy does *not* form part of the sharing list of the line.

The home node is fixed, *c.f.* a *cache-only* memory architecture (COMA) where, in effect, all memory accesses are associative, *e.g.* [7]. A read miss to the line from a processor in another node will cause a copy of the line to be

forwarded to the requesting processor where it is cached in the second-level cache; at the same time a link is made to this node via an additional pointer field in the home copy of the line. Subsequent readers will be added to the sharing list in a similar manner with new sharing list entries added at the head. Figure 2 shows the situation when nodes j and k each have a cached copy of a line whose home is at node i. Note that the processor at the home node of a line may cause a copy of the line to be read into its second-level cache so that two copies may potentially exist at the same node.

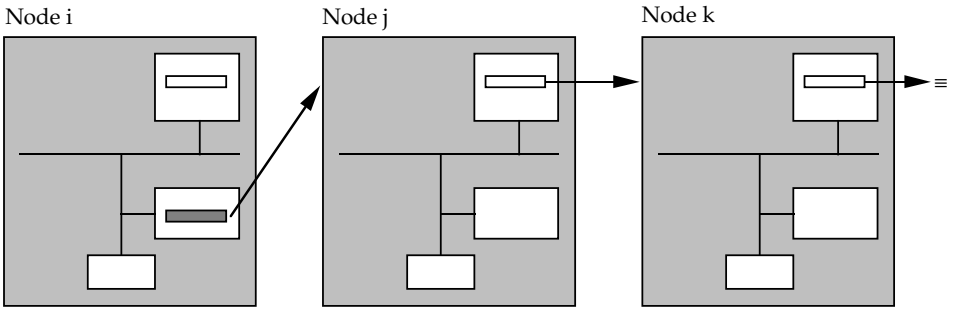

Figure 2: Sharing lists

In order to write to a line all other copies (and the home copy) must be invalidated before the write may proceed. This entails sending an invalidate request to the home node which marks the home copy as being invalid and then forwards the request down the sharing list. With the exception of the writing node all sharing list entries will be invalidated by setting a bit in the associated line in the second-level cache. The home node is locked for the duration of this operation. The last entry on the sharing list will respond to the invalidate request by sending a completion message to the requesting node. In the event of the write being a miss this message will also carry a copy of the line. A final message is sent by the requester to the home node which releases the home lock.

When a write is complete the locally cached copy is marked as being *dirty*, *i.e.* inconsistent with the original home copy. So long as the line remains in the dirty state the processor may write to it repeatedly without incurring coherency traffic.

If another processor tries to read a line that has been written by a remote processor, the read request sent to the home node is forwarded to the first entry of the sharing list. This will subsequently supply a (valid) copy of the line to the reader and the home node, and all sharing list links will be updated. At this point both copies of the line are marked as clean.

If a second-level cache line forming part of a sharing list is displaced following a miss on another location which maps to the same line, the cached copy has to be "unhooked" from the list. This is achieved by sending a message to the home node which passes it down the sharing list until it reaches the entry preceding the unhooking node. The identifier of this node is then passed to the unhooker which decouples itself from the list by replying with the identifier of its successor. This is used to update the list pointers in the obvious way. We assume that the pointers associated with the second-level cache lines are stored in the network controller so that pointer maintenance and traversal can be performed without generating internal bus traffic.

Note that a line may be copied to the cache at the home node. In this situation the line state is maintained as for any

other cached copy except that it does not explicitly appear in the sharing list for that line. Status information at the home memory indicates whether a copy of the line is held in the local cache.

The second-level cache line states are therefore as follows:

HXC Home Exclusive Clean—The line is cached at the same node as the home copy, and is the only cached copy.

HXD Home Exclusive Dirty—As above, except that the line has been written and so the copy in memory is out of date.

HS Home Shared—The line is cached at the home node and there is at least one other copy cached at another node. The cached copies are consistent with the memory copy.

CX Client Exclusive—The home of the cached line is on another node. This is the only valid cached copy.

CHD Client Head Dirty—The home of the cached line is on another node. This copy is the first on the sharing list, and the home memory copy is out of date.

CHC Client Head Clean—As above, except that the home memory copy is coherent.

CS Client Shared—The home of the line is on another node, and other copies exist.

INV Invalid—The line contains no usable information.

A fuller set of line states could be defined, for example distinguishing clean and dirty shared states. However, the coherency protocol can be specified satisfactorily with this parsimonious set which we therefore adopt for brevity and efficiency.

2.2 Coherency Operations

Read/write requests from the processor may result in certain operations being applied to the sharing lists as listed below. Note that two message types are distinguished: short messages which contain only control information (*e.g.* for managing updates to a sharing list) and long messages which contain both control information and a line of data. Long messages are typically about an order of magnitude longer than short messages and an important objective of the protocol is to avoid sending long messages whenever possible.

Creation (CR) A read or write miss on a line in state INV not cached by client nodes. If a client, the processor sends 1 short message to the home node and receives 1 long message from it containing a copy of the line. The line transits to state HXC, HXD, CHC or CX depending on whether or not the home copy is on the same node as the requesting processor and whether the request was a read or write.

Addition (AD) A read miss on a line in state INV but cached by client processors. The processor sends 1 short message to the home node, which supplies the data if it can, or else forwards the request to the sharing list head. The requesting processor receives one long message containing the line and the sharing list pointers are updated accordingly. The line transits to state HS or CS depending on the location of the home node.

Reduction (RE) A write hit on a (non-invalid) line. If a client, the processor sends one short message to the home requesting invalidation of the chain. In any case, the home sends a short invalidation message down the sharing list, causing all copies to be invalidated. If the requesting node is not the last entry in the sharing list, it reads and invalidates the line in its cache, and sends it to the requesting node. The line transits to state HXD or CX depending on whether the processor is at the home node. Note that write to lines which are HXC, HXD and CX require no messages are sent – only local updates to the memory and/or cache are required.

Deletion-Creation (DC) A read or write miss on an uncached line whose address maps to a line not in the invalid state in the cache. This must first be displaced from the cache which involves a deletion: an "unhook" message is sent to the home node which is forwarded down the sharing list. On average this will traverse half the length of the sharing list before arriving at the unhooking node—a short message is associated with each hop. A new list is created off the line being read or written as in CR above.

Deletion-Addition (DA) A read miss on an already cached line whose address maps to a line not in the invalid state in the cache. This must first be displaced from the cache as above. The node is added to the new sharing list as in AD above.

Deletion-Reduction (DR) A write miss on an already cached line whose address maps to a line not in the invalid state in the cache. This is first displaced and the list associated with the line being written is invalidated. The reduction is similar to RE above. The line transits to state HXD or CX.

Invalid-Reduction (IR) A write miss on a line in state INV cached by other processors. Reduction proceeds as per DR above and the line transits to state HXD or CX.

Read Hit (RH) A read hit on the second-level cache. The line state does not change although bus traffic is generated as a result.

3 The Analytical Model

A processor alternates *think periods* and periods when it waits for a memory access to be serviced. Let the think period have a mean of $1/\tau_0$. After a think period, the processor generates a memory request and this request may or may not invoke a transaction to a remote node. During the time a request is processed, the processor is idle. We aim to determine the probability, π, that a processor is busy doing useful work. We will write $\tau = \pi\tau_0$ which is the net rate at which a processor generates read/write requests to the memory system.

The second-level cache is taken to have hit/miss rates of β_{rh} and β_{rm} for reads and correspondingly β_{wh} and β_{wm} for writes. For convenience we write $r = \beta_{rh} + \beta_{rm}$ and $w = \beta_{wh} + \beta_{wm}$. These are parameters of the workload since they express in some way the degree of locality in the application.

State Transitions

Ultimately we need to determine the message traffic that is generated by each processor in servicing memory requests; this will include messages required by the cache coherency protocol. Assuming that all the processors behave in the same way, we begin by deriving the equilibrium cache line state probabilities. To do this we assume that the evolution of the state of a given line in a cache (we shall refer to this as the *observed* cache line) follows a Markov process, independent of the states of other lines. This process is irreducible, aperiodic and has a finite state space. It thus has a steady-state.

We define the following: P_u – the probability that a line is uncached remotely so that the home copy is the only copy in the machine, *i.e.* the probability that no sharing list exists for that line; P_{loc} – the probability that an address maps to the local memory of a given node (a workload parameter, equal to $1/K$ for uniform memory access); P_{hv} – the probability that the home copy of a given line in DRAM is up to date; P_{hd} – the probability that a locally cached copy of a line in DRAM is dirty (i.e. the DRAM copy is out of date and the cached copy is the only valid copy anywhere in the system); P_1 – the probability that a sharing list has exactly one element; P_{second} – the probability that a sharing list element is second in a list of length greater than 1.

For convenience we will also write the complementary probabilities $P_c = 1 - P_u, P_{rem} = 1 - P_{loc}$ and $P_{hi} = 1 - P_{hv}$.

Using s to denotes the set of states HXC,...,INV the generators of the above Markov process are given by the transition rates below; note that we have divided each by the factor τ which is the rate at which a processor issues memory requests. The symbol Δ denotes the mean length of a sharing list at equilibrium and results in an approximate rate.

$$
\begin{aligned}
s \rightarrow HXC, s \neq HS, INV & \quad \frac{\beta_{rm} P_u P_{loc}}{n} \\
HS \rightarrow HXC & \quad \frac{\beta_{rm} P_u P_{loc}}{n} + \frac{(\beta_{rm} + \beta_{wm}) P_1}{n} \\
INV \rightarrow HXC & \quad \frac{r P_u P_{loc}}{n} \\
s \rightarrow HXD, s \neq HXC, HS, INV & \quad \frac{\beta_{wm} P_{loc}}{n} \\
HXC, HS \rightarrow HXD & \quad \frac{\beta_{wh}}{n} + \frac{\beta_{wm} P_{loc}}{n} \\
INV \rightarrow HXD & \quad \frac{w P_{loc}}{n} \\
s \rightarrow HS, s \neq HXC, HXD, INV & \quad \frac{\beta_{rm} P_{loc} P_c}{n} \\
HXC, HXD \rightarrow HS & \quad \frac{(K-1) r}{N} + \frac{\beta_{rm} P_{loc} P_c}{n} \\
INV \rightarrow HS & \quad \frac{r P_{loc} P_c}{n} \\
HXC, HXD, HS \rightarrow CX & \quad \frac{\beta_{wm} P_{rem}}{n} \\
CHC, CHD, CS \rightarrow CX & \quad \frac{\beta_{wm} P_{rem}}{n} + \frac{\beta_{wh}}{n} \\
INV \rightarrow CX & \quad \frac{w P_{rem}}{n}
\end{aligned}
$$

$$
\begin{aligned}
s \rightarrow CHC, s \neq CHD, CX, CS, INV & \quad \frac{\beta_{rm} P_{rem} P_{hv}}{n} \\
CHD, CX \rightarrow CHC & \quad \frac{\beta_{rm} P_{rem} P_{hv}}{n} + \frac{r}{N} \\
CS \rightarrow CHC & \quad \frac{\beta_{rm} P_{rem} P_{hv}}{n} + \frac{(\beta_{rm} + \beta_{wm}) P_{second}}{n} \\
INV \rightarrow CHC & \quad \frac{r P_{rem} P_{hv}}{n} \\
s \rightarrow CHD, s \neq INV & \quad \frac{\beta_{rm} P_{rem} P_{hi}}{n} \\
INV \rightarrow CHD & \quad \frac{r P_{rem} P_{hi}}{n} \\
CX \rightarrow CS & \quad \frac{(K-2) r}{N} \\
CHC, CHD \rightarrow CS & \quad \frac{r(K - \Delta - 1)}{N} \\
s \rightarrow INV & \quad \frac{(K-1) w}{N}
\end{aligned}
$$

For example, the transition $s \rightarrow HXC, s \neq HS, INV$ occurs when there is a local read miss (β_{rm}) on a line which is currently uncached (P_u) and located in the local memory of the requesting processor (P_{loc}). The factor $1/n$ is the probability that the read request maps to the observed cache line. Note that remote operations can also induce state transitions locally. For example, the transition $HS \rightarrow HXC$ can occur if a remote processor performs a miss ($\beta_{rm} + \beta_{wm}$) on a cache line which currently holds a copy of the observed line. The transition occurs when the processor is the only other one holding a copy in the machine (P_1). The other term ($\frac{\beta_{rm} P_u P_{loc}}{n}$) covers the general case of a transition into state HXC of a read miss on a locally held line. Finally, note that the transition $s \rightarrow INV$ corresponds to invalidation—any remote write operation to a line cached locally will cause the line to be invalidated. The factor $(K-1)$ here is the number of remote processors which can issue such a write.

The balance equations can be derived from the above transition rates in the usual way—see for example [6] and are solved iteratively. We will write $q_j, HXC \leq j \leq INV$ to denote the equilibrium probability that a cache line is in state j.

3.1 Sharing

The quantities Δ, P_1 and P_{second} above require the distribution of the number of sharers of a memory line to be known. This is produced from a separate Markov model of line sharing, taken from the point of view of the memory.

We again assume that the evolution of the number of sharers of a memory line follows a Markov process, independent of the states of other memory lines. The model can be solved using standard techniques since the transition rates are expressed solely in terms of known model parameters.

The Markov process state transition diagram for the number of sharers is shown in Figure 3. Note that the state 0 covers both the case where there is no cached copy of the line and the case where the only cached copy is at the home node. This state can be entered from state 1 as the result of a displacement at the (only) remote node with

a copy of the line, and from any other state as the result of a write to the line from the home node. The former occurs with probability P_{miss}/n and the latter with probability $\beta_w P_{loc}/(N/K)$. The state 1 can be reached from state 0 as the result of a remote read to the line (probability $(K-1)\beta_r P_{rem}/(N/K)$), from state 2 as the result of a displacement (probability $2P_{miss}/n$), and also from any other state as a result of a write to the line from any non-home node (probability $(K-1)\beta_w P_{rem}/(N(K-1)/K)$). The transition probabilities for the general case are shown in the diagram.

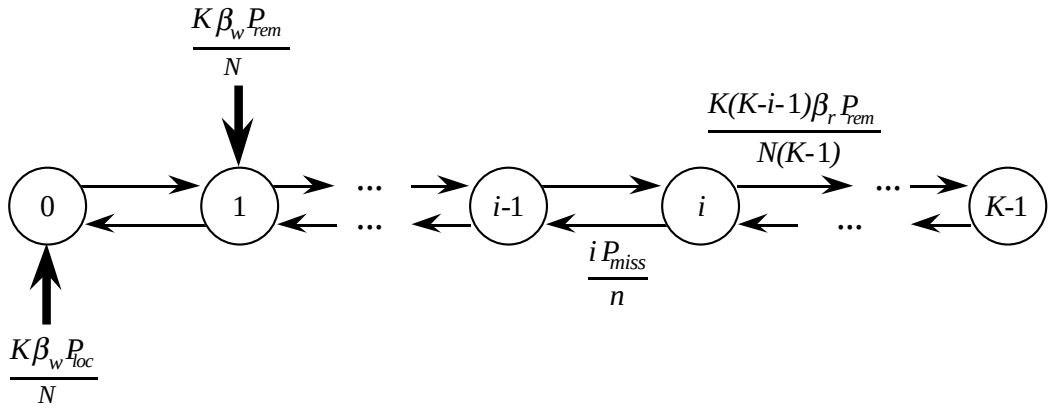

Figure 3: Markov model of the number of ALITE line sharers

The model is solved to obtain the mean length of the sharing list, Δ, and the equilibrium probability P_i that the sharing list is of length $i, 0 \leq i \leq K-1$. We can now express the probabilities defied earlier:

$$\begin{aligned} P_u &= P_0 \\ P_{hv} &= P_u + \frac{P_c q_{CHC}}{q_{CHC} + q_{CHD} + q_{CX}} \\ P_{hd} &= \frac{Knq_{HXD}/N}{1 - Knq_{HS}/N} \\ P_{second} &= \frac{1}{1 - P_0 - P_1} \sum_{i=2}^{K-1} \frac{P_i}{i} \end{aligned}$$

List Operation Probabilities

We next calculate the probability that a processor emerging from a think state invokes a given list operation (the probability distribution of the state at such instants is the same since the memory access stream is assumed Poisson). These are summarised in Table 1. We will denote this table by δ so that, for example, $\delta_{HXD,DC} = (\beta_{rm} + \beta_{wm})P_u$.

Message and Cache/Memory Traffic

Each coherency operation results in a certain number of bus transactions, short messages and long messages. The type of bus transaction varies significantly in each case. Some transactions require just one of the two buses, others require

	Operations						
State	CR	AD	RE	DC	DA	DR	IR
HXC—CS	0	0	β_{wh}	$(\beta_{rm} + \beta_{wm})P_u$	$\beta_{rm}P_c$	$\beta_{wm}P_c$	0
INV	P_u	$\beta_m P_c$	0	0	0	0	$\beta_m P_c$

Table 1: The δ table

both. Once a bus has been claimed the bus holding time also varies depending on the operation being performed.

In order to capture this variability in the model we define a number of bus transaction classes and specify the expected number of each class for each operation/state pair in the form of a table T. We also define similar tables S and L representing the number of short and long messages for a given operation/state pair.

For reasons which will become apparent, we divide the classes into four groups. Transactions in the first group are initiated by the processor and require just a cache bus transaction; those in the second require just the memory bus; the third and fourth groups contain those that require both buses, distinguished by which buses they have to *queue* for. Only transactions initiated from the memory bus can claim both buses; if a cache bus transaction requires the memory bus as well, the cache bus is released prior to queueing for and subsequently claiming the memory bus.

We label the classes C_1, $C_2, \ldots$ and specify the service times for each bus of class i in clock cycles in Table 2. Note that each class belongs to exactly one group. We write $h_{b,i}$ for the holding time (in seconds) of bus b for a class i transaction, $b \in \{c, m\}$. Thus $h_{b,i} = y_{b,i} t_{clock}$ where t_{clock} is the clock cycle time and $y_{b,i}$ is an integer number of cycles. These times correspond to the constants used in the execution-driven simulation to determine the various bus delays.

The entries of T are specified by Table 3. The first two columns specify the operation and state. The fourth column details the bus transaction classes that are invoked in order to complete the given operation in the given state, together with the number of short and long messages sent. Since each operation/state pair may produce a number of transaction sequences depending on whether a particular line is local/remote, dirty in the home cache or valid in the home memory, the corresponding probability is listed separately with each row.

Note that reductions are a special case since the reducing processor sends and receives as many short messages as there are members in the associated sharing list. We estimate this by using the mean length of a sharing list, Δ.

Note also that the various deletion operations (DC, DA, DR) comprise an initial "unhook" phase followed by either a separate creation (CR), addition (AD) or invalidation (IR) phase. After the unhook the cache line is essentially left in a temporary invalid (INV) state.

By substituting the class descriptions from Table 2 in place of the C_i in Table 3, a descriptive breakdown of each operation/state pair is produced. For example, the bus and network traffic involved by performing operation AD in state INV depends on the status and location of the new line which is to be read. There are four cases, each occurring with an associated probability. For example, if the line being read is on the same node and is valid with respect to

Class i	*Description*	*Cache bus cycles* $y_{c,i}$	*Memory bus cycles* $y_{m,i}$
Group 1	(*Cache bus only*)		
C_1	Cache read hit	26	0
C_2	Cache write hit	8	0
C_3	Cache miss	8	0
Group 2	(*Memory bus only*)		
C_4	Acquire memory bus	0	3
C_5	Initiate request message for CPU	0	8
C_6	Read or write memory	0	47
Group 3	(*Queue for and utilise both buses*)		
C_7	Cache lookup (maybe invalidate), read from memory	8	55
C_9	Read line from cache (maybe invalidate)	32	35
C_{10}	Invalidate cache	8	11
C_{14}	Read from cache, copy back to local memory	52	55
Group 4	(*Queue for memory bus, utilise both buses*)		
C_8	Insert in cache, restart CPU	26	29
C_{12}	Copy into or between cache and memory, restart CPU	44	47
C_{13}	Restart CPU	2	5
C_{15}	Copy fro message to cache and memory, restart CPU	26	49
C_{16}	Read line from cache (CPU is stalled)	30	33
C_{17}	Invalidate cache (CPU is stalled)	8	11

Table 2: Service classes

Operation	*State*	*Probability*	*Transaction classes invoked*
CR	INV	P_{loc}	$C_3 + C_{12}$
		$P_{rem}P_{hd}$	$C_3 + C_5 + C_{14} + C_8 + S + L$
		$P_{rem}(1 - P_{hd})$	$C_3 + C_5 + C_7 + C_8 + S + L$
AD	INV	$P_{loc}P_{hv}$	$C_3 + C_{12}$
		$P_{loc}P_{hi}$	$C_3 + C_5 + C_9 + C_{15} + S + L$
		$P_{rem}P_{hv}$	$C_3 + C_5 + C_6 + C_8 + S + L$
		$P_{rem}P_{hi}$	$C_3 + C_5 + C_9 + C_8 + C_4 + 2S + L$
RE	HXC	1	$C_3 + C_{12}$
	HXD	1	C_2
	HS	1	$C_3 + C_5 + (\Delta - 1)C_{10} + C_9 + C_8 + \Delta S + L$
	CX	1	C_2
	CHC	1	$C_3 + C_5 + \Delta C_{10} + C_{13} + C_4 + (\Delta + 2)S$
	CHD	1	$C_3 + C_5 + 2C_4 + (\Delta - 1)C_{10} + C_{13} + (\Delta + 2)S$
	CS	P_{hv}	$C_3 + C_5 + \Delta C_{10} + C_9 + C_8 + C_4 + (\Delta + 2)S + L$
		P_{hi}	$C_3 + C_5 + (\Delta - 1)C_{10} + C_9 + C_8 + 2C_4 + (\Delta + 2)S + L$
IR	INV	P_{loc}	$C_3 + C_5 + (\Delta - 1)C_{10} + C_9 + C_8 + \Delta S + L$
		$P_{rem}P_{hv}$	$C_3 + C_5 + \Delta C_{10} + C_7 + C_8 + C_4 + (\Delta + 2)S + L$
		$P_{rem}P_{hi}$	$C_3 + C_5 + (\Delta - 1)C_{10} + C_9 + C_8 + 2C_4 + (\Delta + 2)S + L$
Deletion	HS,HXC		0
	HXD		30
	CHC,CHD,CX		$C_5 + C_4 + C_{16} + C_6 + 2S + L$
	CS		$C_5 + (\Theta + 1)C_4 + C_9 + C_6 + (\Theta + 2)S + L$
RH	not INV	1	C_1

Table 3: Bus and message transactions

other sharers ($P_{loc}P_{hv}$) then the operation can be completed without communication: the new line can be read into the cache from memory. This corresponds to two transaction classes (C_3 and C_{12}) for which there are associated bus delays in Table 2.

If, however, the new line's home node is elsewhere and if the home copy is invalid, due to a client write operation, then more work must be done. A short message is sent to the home node of the new line; when this is received the home node forwards a short request message to the current sharing list head which has an up-to-date copy of the line. This node claims both buses, fetches a copy of the line from its cache and returns it as a long message, targeted to the initiator of the AD operation. When this long message is received both buses at the initiator are claimed and the line is transferred to the second-level cache. This in turn restarts the processor. In this latter case two short messages and one long message are sent — their transmission does not affect the bus queueing times but does add a delay to the overall read/write response time.

The (sum of the) coefficient(s) of C_i for operation p in state s, together with the associated probability determine $T_{p,s,i}$. If the coefficient is F and the associated probability is r then $T_{p,s,i} = rF$. Similarly the coefficients of S and L (*i.e.* the number of short and long messages respectively) in Table 3 determine $S_{p,s}$ and $L_{p,s}$ respectively.

3.2 Modelling the Nodes

The node architecture is modelled as a queueing network with a server representing each bus. The bus delays depend on the type of transaction and so the transaction classes in Table 2 become service classes in an M/G/1 queueing model. Pointer traversal and pointer maintenance is handled by the network controller; this is pipelined and the associated delays are therefore assumed to be subsumed by the message transmission times.

We need to distinguish the class groupings in Table 2 and we write G_i to denote the set of classes associated with Group i in the table. The set of transactions that require respectively the cache bus and memory bus are defined to be:

$$\begin{aligned} Bus_c &= G_1 \cup G_3 \cup G_4 \\ Bus_m &= G_2 \cup G_3 \cup G_4 \end{aligned}$$

The average arrival rate of transactions of class i is given by:

$$\lambda_i = \tau \sum_{p,s} q_s \delta_{p,s} T_{p,s,i}$$

so that the arrival rate of transactions to the cache and memory buses are respectively:

$$\begin{aligned} \lambda_c &= \sum_{i \in Bus_c} \lambda_i \\ \lambda_m &= \sum_{i \in Bus_m} \lambda_i \end{aligned}$$

The queueing network is complicated by the fact that internal requests from the processor and external requests from the network may require either one bus, or both buses, to complete a transaction. Group 3 transactions hold both buses at the same time, leading to a form of simultaneous resource possession in the queueing network and hence blocking-before-service. We develop an approximate solution to this problem by augmenting the service time at the memory bus with the *waiting time* at the cache bus, for transactions in group 3.

The n^{th} moment of the service time at bus $b \in \{c, m\}$ is given by:

$$m_{n,b} = \frac{\sum_{i \in Bus_b} \lambda_i h_{b,i}^2}{\lambda_b}$$

The queueing time at the cache bus, Q_c, is obtained from the Pollaczek-Kintchine formula, suitably adapted to account for the multiple classes:

$$Q_c = \frac{\sum_{i \in Bus_c} h_{c,i}^2 \lambda_i}{2(1 - \sum_{i \in Bus_c} h_{c,i} \lambda_i)})$$

The mean queueing time at the memory bus is complicated by the fact that bus transactions in class 3 require both buses to be held simultaneously. The M/G/1 model requires the second moment of service time and so we have to calculate the second moment of the waiting time at the cache bus. Since the service time at the cache bus is constant for the transaction class in question, the required second moment is given by the second moment of the *queueing time* at the cache bus, Q_{2c}. To simplify the notation that follows, the memory bus holding times listed for group 3 classes in Table 2 include their cache bus holding times as well; a sum of two constants. For consistency, the same applies to group 4.

Now, by analyzing the generating function of the steady-state queue length distribution it can be shown that the n^{th} moment of the waiting time at bus $b \in \{c, m\}$ can be written

$$W_{n-1,b} = m_{n-1,b} + \frac{\lambda_b \sum_{i=2}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) m_{i,b} W_{n-i,b}}{n(1 - \lambda_b m_{1,b})}$$

(see *reference omitted*) from which the n^{th} moment of the queueing time follows:

$$Q_{n,b} = W_{n,b} - \sum_{i=1}^{n-1} \left(\begin{array}{c} n \\ i \end{array} \right) m_{i,b} Q_{n-i,b} - m_{n,b}$$

Thus,

$$Q_{2c} = W_{2,c} - m_{2,c} - 2Q_1 m_{1,c}$$

The mean queueing time at the memory bus may now again be obtained from the Pollaczek-Kintchine formula:

$$Q_m = \frac{\sum_{i \in G_2} h_{m,i}^2 \lambda_i + \sum_{i \in G_3} (h_{m,i}^2 + Q_{2c} + 2h_{m,i} Q_c) \lambda_i}{2(1 - (\sum_{i \in G_2} h_{m,i} \lambda_i + \sum_{i \in G_3} (h_{m,i} + Q_c) \lambda_i))}$$

The response time of the memory system, R, may now be now obtained from the various bus transaction times and the short and long message transmission times t_{short} and t_{long}. Defining $\Omega = \{CR, ..., RH\}$ to be the set of coherency

operations and $\Theta = \{CXC, ..., INV\}$ to be the set of sharing list operations, we obtain:

$$R = \sum_{p \in \Omega} \sum_{s \in \Theta} q_s \, \delta_{s,p} \left(\left(\sum_{i=1}^{C} T_{s,p,i} \, w_i \right) + (S_{s,p} \, t_{short} + L_{s,p} \, t_{long}) \right)$$

where w_i is the waiting time for class i transactions:

$$\begin{aligned} w_i &= h_{c,i} + Q_c && i \in G_1 \\ &= h_{m,i} + Q_m && i \in G_2 \cup G_4 \\ &= h_{m,i} + Q_c + Q_m && i \in G_3 \end{aligned}$$

With the memory response time R known, the processor utilisation, or "system power" is then easily found:

$$\pi = \frac{\frac{1}{\tau}}{\frac{1}{\tau} + R} = \frac{1}{1 + \tau R}$$

Note, however, that in calculating the arrival rates for each bus transaction class we assumed the existence of π by virtue of the factor $\tau = \pi \tau_0$ in the defining summation. We have thus introduced a new fixed-point problem for determining π and again appeal to an iterative solution method. Essentially, this is how we are solving a closed system in which no more than K nodes can be waiting for a memory access, the rest "thinking". We note in passing that some care has to be taken in updating the approximation to π in order to ensure convergence. The execution time for the whole model is around 5 seconds using Mathematica 2.2 on a PowerMacintosh 7100/66 computer. This can be compared with the execution times for the simulator which, for realistic application benchmarks, can run into many hours, or even days. The benefits of the analytical model in this respect are self-evident.

4 Comparative Analysis

In this section we undertake validation of the above analytical model against results obtained from an existing low-level event-driven simulator of the same system [9]. The simulator is capable of running real parallel programs, and collects statistics such as cache hit rates, bus utilisation, processor utilisation, etc. The simulator models the same target architecture, and has been calibrated identically, using the constants defined in Table 2, and the coherency protocol used adheres to the actions of Table 3.

Validation obviously requires the analytical model and the simulator to use identical workloads. We have adopted two synthetic programs for this purpose: first `synth`, a synthetic memory reference generator program is used, in which each processor repeatedly performs local operations for a random period, before making a memory reference to a location chosen randomly from a fixed sized region according to a uniform probability distribution. Thus we take $P_{loc} = 1/K$ in the analytical model. The program is unrealistic in that no locality is modelled, but it does closely reflect the workload parameterisation of the analytical model. Moreover, the size of the program's memory is clearly the size of the given fixed region. It is therefore reasonable to expect a close correlation between the two sets of results.

Run number	Number of CPUs	Cache size (lines)	Memory size (lines)
1	4	256	2048
2	4	256	4096
3	4	256	8192
4	8	256	2048
5	8	256	4096
6	8	256	8192
7	16	256	2048
8	16	256	4096
9	16	256	8192

Table 4: Simulation parameters for `synth` and `synth-l`

The second program, `synth-l`, is a modified version of the first, in which it is more likely that a processor references memory for which it is the home node. Specifically we chose $P_{loc} = 3/k$. In this way, we use P_{loc} to reflect the locality which exists in real programs: programmers structure programs so that most references are to locally allocated data.

4.1 Methodology

The comparative analysis is done as follows: the simulator is first used to execute a given benchmark program. This produces a set of workload parameters for the benchmark which are used to parameterise the analytical model. These consist of the mean think period τ_0, the various hit rates for reads and writes, $\beta_{rh}, \beta_{rm}, \beta_{wh}, \beta_{wm}$ and the memory usage N, together with the machine configuration parameters K and n.

We first compare the equilibrium line state probabilities predicted and then consider the key performance metrics, namely the bus and processor utilisations. From these mean bus queueing times and memory latency follow immediately: see the formulae at the end of Section 3.2. We remark that the execution time of the analytical model is of the order of five seconds using Mathematica on a Macintosh Power PC. By way of contrast the *simplest* simulation runs here take of the order of three hours: this increases dramatically as the problem size is scaled.

4.2 Simulation Parameters

The simulation parameters for `synth` and `synth-l` are shown in Table 4. For example, the first benchmark run assumes a 4 node system with a total of 2048 lines of memory. The caches at each node are 256 lines in size, and therefore cache replacements will occur. Although the cache and memory sizes used here are small, their ratio largely determines the output of simulations. More realistic cache sizes require longer simulation times to ensure that results are not dominated by startup effects.

Run number	β_{rm}	β_{rh}	β_{wm}	β_{wh}	Think time (cycles)
1	0.814	0.054	0.130	0.002	395.3
2	0.844	0.027	0.127	0.002	383.9
3	0.860	0.014	0.126	0	378.7
4	0.819	0.050	0.130	0.001	390.8
5	0.846	0.025	0.128	0.001	380.7
6	0.860	0.013	0.126	0.001	375.6
7	0.827	0.042	0.130	0.001	388.1
8	0.848	0.023	0.128	0.001	378.9
9	0.861	0.013	0.126	0	374.1

Table 5: Miss rates and think times for `synth`

Run number	β_{rm}	β_{rh}	β_{wm}	β_{wh}	Think time (cycles)
1	0.722	0.135	0.137	0.006	452.0
2	0.804	0.062	0.131	0.003	416.2
3	0.839	0.032	0.128	0.001	402.6
4	0.797	0.067	0.134	0.002	424.5
5	0.832	0.038	0.129	0.001	438.7
6	0.852	0.020	0.127	0.001	398.0
7	0.816	0.051	0.132	0.001	415.0
8	0.841	0.029	0.129	0.001	402.4
9	0.857	0.016	0.127	0	395.8

Table 6: Miss rates and think times for `synth-l`

4.3 Simulated Miss Rates and Think Times

The miss rates and think times determined by the simulator characterise, albeit in a somewhat simplistic manner, the workload. These results, together with the simulation parameters (number of nodes, cache and memory size) form the parameters to the analytical model. The results are shown in Table 5 for `synth` and Table 6 for `synth-l`.

4.4 Equilibrium Cache Line State Probabilities

These are the results of the first phase of the analytical model, which in turn form the input to the queueing model of the nodes (phase 2). It is vital, therefore, that close agreement is found between the corresponding results from the simulation and the analytical model. For each run of the simulator, the time spent in each cache line state is recorded, allowing state probabilities to be determined. Results are summarised in Tables 7 and 8. Two lines for each run are shown: the upper line are simulation results, whereas the lower comes from the analytical model. Since the simulator is execution-driven, hence relating to specific programs, confidence bands were not appropriate in its output analysis.

Note that the simulator does not differentiate between certain pairs of states, and a single probability for such pairs

Run number	INV	HXC	HXD	HS	CS	CX
1	0.045	0.145	0.025	0.069	0.642	0.074
	0.047	0.149	0.025	0.064	0.640	0.075
2	0.023	0.180	0.028	0.037	0.650	0.083
	0.023	0.180	0.028	0.036	0.650	0.083
3	0.011	0.201	0.029	0.020	0.652	0.087
	0.012	0.199	0.029	0.019	0.654	0.087
4	0.097	0.044	0.009	0.060	0.728	0.062
	0.103	0.047	0.009	0.057	0.721	0.063
5	0.051	0.070	0.011	0.038	0.750	0.079
	0.053	0.071	0.011	0.037	0.748	0.080
6	0.026	0.089	0.013	0.022	0.759	0.091
	0.027	0.088	0.013	0.021	0.759	0.092
7	0.188	0.009	0.003	0.039	0.718	0.043
	0.198	0.010	0.003	0.037	0.708	0.044
8	0.104	0.022	0.004	0.030	0.778	0.062
	0.107	0.022	0.004	0.030	0.773	0.064
9	0.055	0.035	0.005	0.020	0.804	0.081
	0.056	0.035	0.005	0.020	0.803	0.081

Table 7: Comparison of equilibrium line state probabilities for `synth`

Run number	INV	HXC	HXD	HS	CS	CX
1	0.027	0.347	0.076	0.059	0.429	0.062
	0.051	0.290	0.055	0.113	0.432	0.059
2	0.015	0.444	0.077	0.035	0.378	0.051
	0.024	0.440	0.070	0.069	0.349	0.048
3	0.007	0.528	0.083	0.018	0.321	0.045
	0.012	0.509	0.076	0.037	0.321	0.045
4	0.097	0.052	0.011	0.062	0.714	0.064
	0.106	0.052	0.011	0.063	0.703	0.064
5	0.049	0.110	0.019	0.048	0.699	0.075
	0.054	0.104	0.011	0.052	0.696	0.075
6	0.025	0.125	0.019	0.025	0.718	0.088
	0.027	0.122	0.018	0.029	0.716	0.088
7	0.189	0.010	0.003	0.043	0.712	0.043
	0.200	0.011	0.004	0.042	0.700	0.043
8	0.103	0.024	0.005	0.033	0.772	0.062
	0.108	0.025	0.005	0.034	0.775	0.063
9	0.054	0.049	0.008	0.026	0.783	0.080
	0.056	0.048	0.008	0.026	0.782	0.080

Table 8: Comparison of equilibrium line state probabilities for `synth-l`

Run number	cache bus utilisation	memory bus utilisation
1	0.118 0.116	0.249 0.255
2	0.119 0.120	0.257 0.270
3	0.119 0.120	0.260 0.280
4	0.116 0.115	0.257 0.272
5	0.118 0.120	0.272 0.290
6	0.120 0.122	0.280 0.296
7	0.114 0.117	0.253 0.273
8	0.117 0.119	0.274 0.290
9	0.119 0.120	0.286 0.300

Table 9: Bus Utilisations for `synth`

is produced. For example the states CHC and CHD are not explicitly maintained, and form part of CS. These states were introduced in the model in order to estimate P_{hv}.

It can be seen that the results from the analytical model and the simulator show very strong agreement, especially in the case of uniform memory access. The largest discrepancies occur for rare line states which are less significant and for which the simulation-based probabilities are less reliable anyway. For non-uniform memory access, agreement is slightly less good when there are 4 nodes. Even here, the significant discrepancies are restricted to the invalid state (*e.g.* 3% vs. 5%) and the home-shared state (*e.g.* 6% vs. 11%) which occur with small probabilities.

These results indicate that the cache line state transitions defined for the analytical model in terms of the workload parameters are accurate.

4.5 Bus Utilisation

A key performance metric for architectures of this type is the utilisation of the system buses. If utilisation is too high, requests will spend a relatively large period of time queueing for the buses, thus reducing overall system power. Alternatively, low utilisation implies that the cost of designing and implementing the decoupled bus arrangement may not be offset by an adequately large improvement in system performance. Results for the analytical model and simulator are shown in Table 9. Again, simulation results are shown in the top line of each pair.

Again, good agreement is found, especially for the cache bus where the results are almost coincident. The analytical

Run number	cache bus utilisation	memory bus utilisation
1	0.103	0.177
	0.102	0.181
2	0.114	0.183
	0.113	0.188
3	0.118	0.187
	0.118	0.191
4	0.107	0.233
	0.108	0.250
5	0.111	0.246
	0.114	0.263
6	0.115	0.261
	0.117	0.275
7	0.109	0.240
	0.111	0.257
8	0.113	0.263
	0.114	0.277
9	0.114	0.270
	0.117	0.287

Table 10: Bus Utilisations for `synth-1`

model overestimates memory bus utilisation by 3-8%. These results indicate that the queueing analysis, which forms the central part of the model of the nodes, is accurate. Specifically it implies that:

1. The constants defined for the various operations in Table 2 agree with those of the simulator
2. The assumptions and simplifications adopted in the model are valid.

4.6 Processor Utilisation

Our final results concern processor utilisation, or system power. This is the proportion of the execution time of the program during which processors are active, and is the most important metric. The graph of Figure 4 shows the variation of processor utilisation as a function of the number of nodes, for the three memory sizes.

As expected, the analytical model's predictions are very close to those of the simulator. This is unsurprising in view of the good agreement already obtained for bus utilisations and line state probabilities, together with the fact that service time parameters and frequencies of transaction classes for each operation/line state pair are specified identically.

Processor utilisation decreases as the memory size is increased, since it becomes less likely that the caches will hold the required data. In our terminology, memory size is the size of a program rather than the number of lines provided in the architecture. Hence, increased memory size means a larger program. Utilisation also decreases as the number of processes increases, as expected, since the wider distribution of memory causes increased overheads. However, the decrease is quite slight and so elongation of memory latency (which follows directly from the definition of processor

utilisation) is not excessive. Hence we can be optimistic about the scalability prospects of the architecture and coherency protocol, at least up to 16 processing nodes.

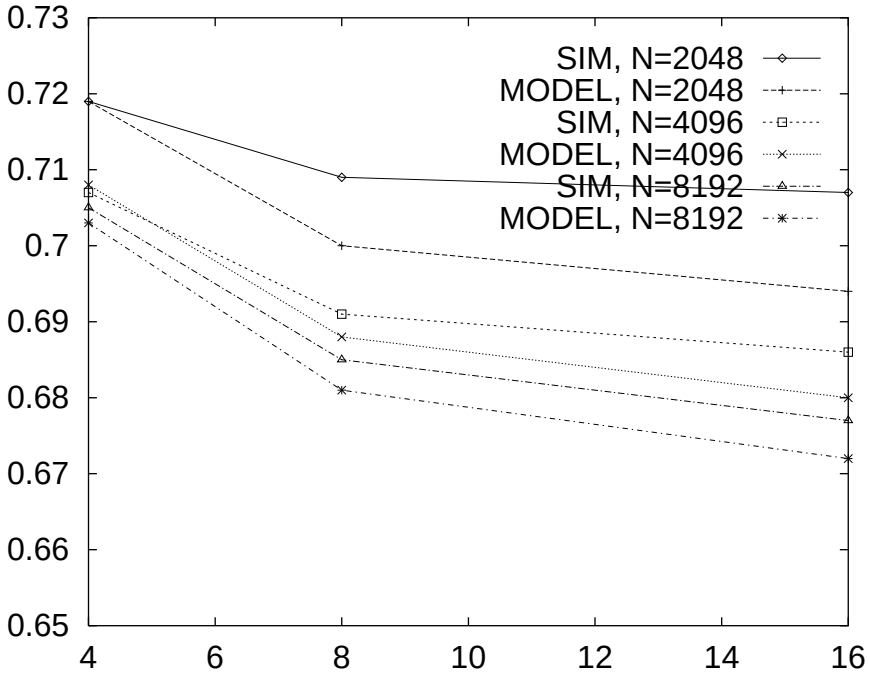

Figure 4: Processor Utilisation of `synth`

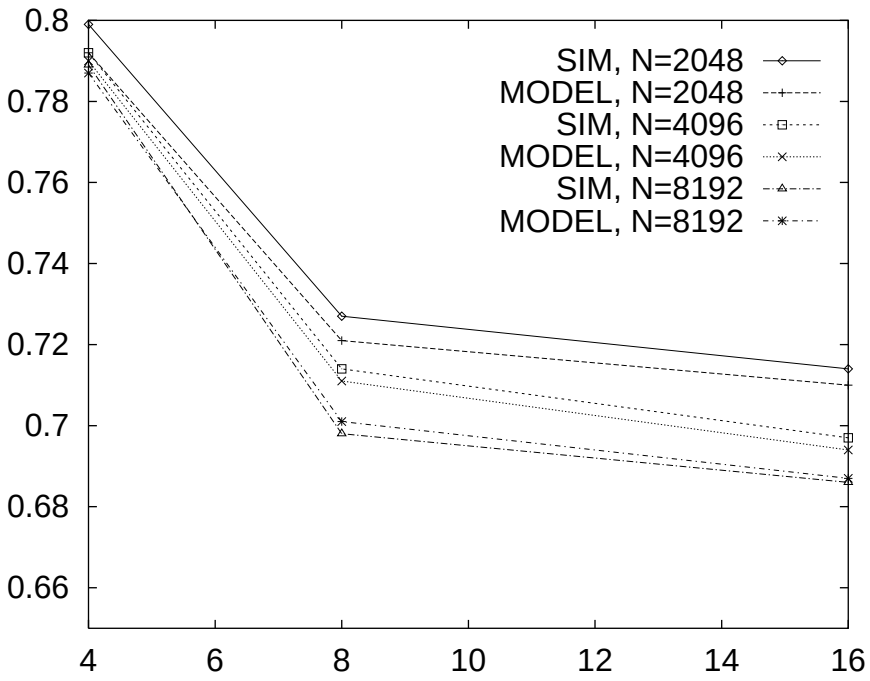

Figure 5: Processor Utilisation of `synth-l`

Qualitatively, a closer match is obtained between the simulation and analytical models for the `synth-l` benchmark. This is because in the analytical model we use the *measured* locality metric, P_{loc} from the simulator, rather than the

INV	HXC	HXD	HS	CS	CX
0.418	0.003	0.068	0.005	0.058	0.448
0.402	0.008	0.126	0.016	0.072	0.376

Table 11: Comparison of equilibrium line state probabilities for MP3D

quantity $3/K$ used to parameterise the benchmark. This removes the distortion in the memory reference characteristics which is caused by the simulator modelling an on-chip first-level cache. This enables the request to be satisfied without stalling the processor. By using the same principle for `synth` an improved correlation between the two models is similarly observed.

4.7 A Real Benchmark

Finally we ran the simulator on the MP3D benchmark taken from the Stanford SPLASH suite [11]. Here we anticipated a number of problems. First and foremost, we had no way of estimating the "size" of the program. The simulator accumulates the total amount of memory used, but although this is the required value for the above synthetic programs, a real application runs in phases, each using its "working set" of memory. The required memory size N is this working set size in each phase. It will be much smaller than the total accumulated memory and vary from phase to phase. In fact, a separate instance of the model should be run for each of the phases that can be identified.

However, in the absence of further information, we decided to select a value for N that provided good agreement on the equilibrium probability that a cache line was invalid. The accuracy of the model could then be judged by the closeness of the agreement on the other performance measures, *i.e.* the remaining line state probabilities and bus and processor utilisations.

A reasonably good agreement was obtained for the equilibrium line state probabilities, as shown in Table 11. However, the bus utilisations were over-estimated by the order of 40% in each case. This is almost certainly due to the fact that the MP3D benchmark spends a large proportion of its execution time idle due to barrier synchronisations. A more sophisticated workload model would be required to capture this behaviour, and hence limit the discrepancy between the models. This and other ideas for future work are discussed in Section 5.1.

5 Summary and Conclusions

We have developed a novel analytical model for a distributed cache coherency protocol running on a realistic shared memory multiprocessor. We have also evaluated the accuracy of the model against an execution-driven simulation. The target architecture is quite sophisticated, for example having decoupled buses to reduce contention, and an objective of the coherency protocol is to reduce load on the cache bus. An immediate goal of our research is to measure, and account for, any major inaccuracies in the analytical model under various real workloads, particularly with reference to the model of the coherency protocol and internal bus contention.

Our experiments with a synthetic reference generator program have shown a remarkably close match between the mathematical model and simulation in almost every case. Predictions for processor utilisation and state transition rates correspond closely to those found by a detailed simulation. Although the random reference generator models closely the assumptions made in the analytical model, the comparison is still important since a number of approximations are used in the analysis. These include the assumptions that line state transitions define a Markov process, implying that state holding times are negative exponential random variables, and that arrival processes are Poisson as well as the approximate model of simultaneous resource possession. The validation has demonstrated that the analytical model is robust in terms of accuracy.

Both internal and external communication traffic is represented in considerable detail in the analytical model and this has proven to be one of the major successes. The protocol description, which is essentially an abstraction of the simulator code itself, is broken down into a large table comprising counts of the number of instances of various bus transaction classes, together with the number, and type, of messages which need to be sent, for each type of coherency operation. Because of the close match between the line state probabilities from both models, the number of such transactions and messages actually observed in the simulation are predicted accurately by the model. Some of the discrepancies are exaggerated by the way the simulator handles network traffic. For example, coherency operations which have to queue at the controller in the simulation are assumed to be pipelined in the analytical model, the delay being absorbed into the message transmission time. The mean queue length here is small, however, and this limits the effect. We predict that the differences will become greater in programs which exhibit high degrees of read contention since this will inherently increase the mean queue length at the network buffer. An extension to the queueing model may be required here, although it could be argued that from the architecture point of view the simulator's treatment of this traffic is far from optimal.

5.1 Future Work

The obvious next step is to complete the validation for this model by running it against more realistic parallel programs. The simulator is capable of running programs from the Stanford SPLASH suite [11] and a validation based on selected programs from the suite is is currently in progress. *Author's note: space permitting, these results could be presented in the final version of this paper.*

The analytical model described is deliberately simplistic in nature. The workload model assumes uniform usage of the cache lines, even though for some applications cache usage is heavily "regionalised". Numerous extensions to the workload model are possible in this respect; a regionalised cache model was studied in (*reference omitted*), for example, and the same could be done here. Also a more sophisticated model of the memory addressing pattern will almost certainly be required to model some of the more esoteric benchmark programs. The model does currently allow locality of reference to be captured in a single probability. It would also be interesting to examine the effect on performance of a significant proportion of read-only data. This data would have read probability of unity and its own hit/miss ratio. As a result, sharing lists would be much longer since they would only reduce on a displacement.

The queueing model of the node and the representation of the network traffic are, however, very robust. What has proven particularly tedious, however, is the production of the protocol specification table and bus transaction class

descriptions. A protocol specification and modelling tool which could automatically produce this information, and indeed even the coherency protocol simulation itself, would be of considerable benefit. It is conceivable that formal correctness proofs of a new protocol may also be feasible using such a tool as the starting point.

Finally, the modelling technique proposed here must be demonstrably useful in other protocols. Of particular interest to the authors are various schemes for reducing contention at the cache controller. Predictive performance models of new protocol proposals such as these promise to yield considerable savings in simulation development and execution time during the early stages of a new design.

References

[1] E. Ametistova and I. Mitrani. Modelling and evaluation of cache coherence protocols in multiprocessor systems. In *9th U.K. Performance Engineering Workshop*, 1993.

[2] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[3] H. Davis and S. Goldschmidt. Tango: a multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Stanford University, 1990.

[4] Stein Gjessing, David B. Gustavson, James R. Goodman, David V. James, and Ernst H. Kristiansen. The SCI cache coherence protocol. In Michel Dubois and Shreekant S. Thakkar, editors, *Workshop on Scalable Shared Memory Multiprocessors, Seattle, May*, pages 219–237, Boston, 1992. Kluwer Academic Publishers.

[5] A. G. Greenberg and I. Mitrani. Analysis of snooping caches. In *Performance 87*, 1987.

[6] P. G. Harrison and N. M. Patel. *Performance Modelling of Computer Systems and Communications Networks.* Addison-Wesley, 1993.

[7] Kendall Square Research. *KSR1 Principles of Operations*, 1992.

[8] A. Nowatzyk and P. Prucnal. Are crossbars really dead? The case for optical multiprocessor interconnect systems. In *22nd Annual International Symposium on Computer Architecture*, pages 106–115, 1995.

[9] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An argument for simple COMA. In *First IEEE Symposium on High Performance Computer Architecture, Rayleigh, North Carolina, January 1995*, pages 276–285, 1995.

[10] Steven L. Scott, James R. Goodman, and Mary K. Vernon. Performance of the SCI ring. *19th Annual International Symposium on Computer Architecture, Gold Coast, May, in Computer Architecture News*, 20(2):403–414, May 1992.

[11] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[12] Manu Thapar and Bruce Delagi. Stanford distributed-directory protocol. *IEEE Computer*, 23(6):78–80, June 1990.