An Analytical Model of the SCI Coherence Protocol

Eric Rotenberg

June 19, 1995

Abstract

This paper develops an analytical model for the SCI coherence protocol based on approximate mean-value analysis techniques. The model is tested against a detailed simulator running on the Wisconsin Wind Tunnel [7] using four SPLASH benchmarks [6]. The simulator is instrumented to obtain model input parameters per application. For two of the four benchmarks, the model produces response times which are within 6% of the response times obtained from simulation. For the other two benchmarks, response times differ by 20% and 30%; a hypothesis for the discrepancy is presented.

1 Introduction

The Scalable Coherent Interface (SCI) [1] has emerged as an important hardware-based approach to the cache coherence problem in large-scale shared memory multiprocessors. Since SCI is an IEEE standard, it may become the hardware coherence protocol of choice in industry. Consequently, being able to predict the performance of SCI implementations will become increasingly important.

This paper develops an analytical model for the SCI protocol based on approximate mean-value analysis techniques. The model derived here is one possible implementation of SCI – because SCI is a *standard*, it encompasses many different design options. However, the derivation yields a general approach which can be applied to other options.

In Section 2, a brief overview of the basic SCI coherence protocol is given. The AMVA analytical model is developed in Section 3. Parameterization and validation of the model is described in Section 4. Section 5 shows various interesting statistics gathered for the benchmarks. Finally, conclusions are drawn in Section 6.

2 The SCI Protocol

This section briefly reviews the basic operations specified in the SCI coherence protocol. Understanding of cache coherence schemes is assumed. The reader is referred to the actual specification [1] for a more comprehensive description of the SCI protocol.

SCI is a directory-based invalidate coherency protocol. It differs from other directory-based protocols [4][5] in that the state of a cache block is distributed among the sharers of the block. Distribution of the state is achieved through the use of doubly-linked lists, as shown in Figure 1. Each memory module maintains the state of its blocks in a directory. In the base protocol, a directory entry can be in one of two states: *home* or *gone*. *Home* says the block is not cached by any processor, and therefore memory can satisfy requests for the block immediately. If the state is *gone*, the block is cached by one or more processors and may have been modified. In this case, the directory entry contains a pointer to the first processor in the sharing list for this block. On a request to the block, memory responds with the pointer instead of the data; the requesting processor then uses this pointer to forward its request to the head of the list. During the transactions with memory and the head of the list, the requesting processor prepends itself to the head of the sharing list, becoming the new head. In general, a coherent transaction consists of three phases:

- *Memory Read*: when a processor misses in its cache, it first sends a request for the data to the *home* directory – the memory module responsible for the given block. If the state of the block is *home*, memory responds with the data, changes the state to *gone*, and sets the value of its pointer equal to the node id of the requesting processor. If the state was originally *gone*, memory cannot satisfy the request directly. Instead, it responds with a pointer to the head of the list. Memory updates its pointer to reflect the new head, just as before.

- *Cache Read*: if memory responds with a pointer instead of data, the processor must forward its request to the head of the list – this is a cache-to-cache transfer. Upon receiving the cache read, the head of the list responds with the possibly modified data and changes its backward pointer to the node id of the requester. After this phase, the requester has prepended itself to the head of the list and has obtained the cache block.
- *Cleanup*: if the cache miss which generated the coherent transaction is a store, the processor cannot proceed with the store until it has invalidated all other cached copies (SCI is an invalidate, or single-writer protocol). The new head issues an invalidate request to the next processor in the list; this processor acknowledges with a pointer to the next processor after it. The head uses this new pointer to issue the next invalidate, and the cleanup process continues in this serial fashion until a null pointer is returned by the tail processor.

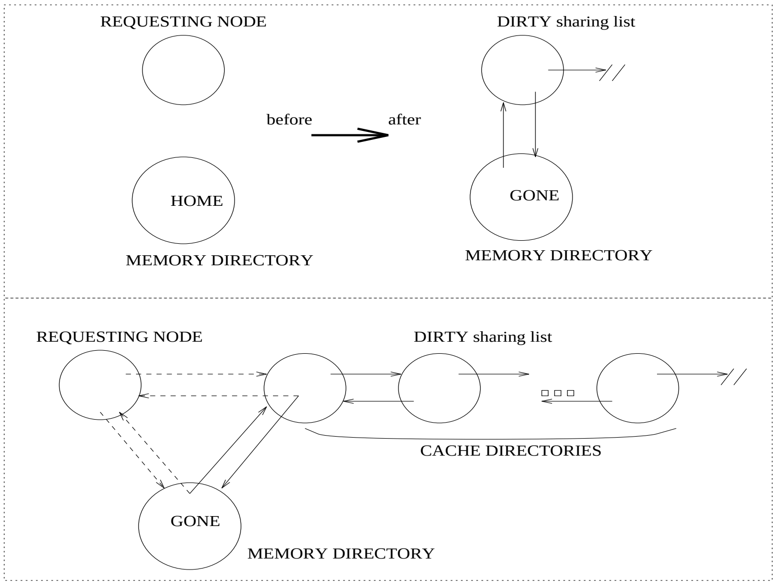

Figure 1: This figure shows the doubly-linked list structure used to distribute the state of a cache block in SCI. The top diagram illustrates the creation of a dirty list – data is returned from memory. The bottom diagram illustrates prepending to an existing dirty list – the processor must forward its request to the head of the list. If the request was generated by a store, a cleanup must be performed after prepending: serial invalidation of each node in the list.

The SCI standard specifies two other sets of options beyond the base protocol, each having higher performance and additional complexity. These protocols specify additional states for a sharing list. Whereas the base protocol only has one kind of sharing list – *dirty* – the more complex options introduce *fresh* and *clean* sharing lists. Having these states results in higher performance as follows:

- *fresh and gone*: if memory can distinguish between "cached and modified " and "cached and unmodified", there is opportunity for memory to respond with data even if it is cached elsewhere. The *fresh* state indicates that there are other copies in the system, but memory still has an up-to-date copy; *gone* indicates the block is cached and modified.
- *clean lists*: it may be desirable to distinguish between two types of reads – "normal read-only" and "read with intent to modify later". A *read-fresh* leaves the memory in the state *fresh*, whereas *read-clean* leaves memory in the state *gone*. The latter read allows a subsequent write to the block to proceed immediately, just as with the *dirty* state. Further, if a block must be replaced in the cache

and it is in state *clean*, the replacement does not require any data to be written to memory. This is in contrast with the *dirty* state, where a full writeback is required even if the data is not actually modified.

The advanced SCI options also specify features such as QOLB [8]. pairwise sharing, and relaxed consistency. Of these options, only relaxed consistency is considered in this paper.

3 Analytical Model

This section describes the analytical model of the SCI protocol. The modeling technique used is approximate mean value analysis (AMVA) [9]. AMVA is chosen for several reasons. First, it has proven successful in modeling other cache coherence schemes [11] and memory systems [10]. Second, it is fairly straightforward to apply – the derived equations are intuitive. Further, it is also fairly simple to heuristically add support for non-product form features in the system being modeled. Lastly, the solution complexity of AMVA is reasonable.

3.1 Overview of the Model

Figure 2 shows the view of the SCI system as seen by the model. The hardware consists of multiple nodes, each node having five resources:

- Processor Cache
- Memory (includes Directory)
- Memory Bus
- Network Interface - Outgoing queue
- Network Interface - Incoming queue

The cache, memory, and network interface are connected together through a common bus. The shared memory multiprocessor is formed by connecting many of these nodes together through an arbitrary interconnection network. Network contention is not modeled – the network looks like an infinite server with a fixed latency per message type. The tradeoff here is between accuracy and modeling complexity [3]. As described in [2], latencies used in all of the experiments were validated against a detailed network simulator for each benchmark application.

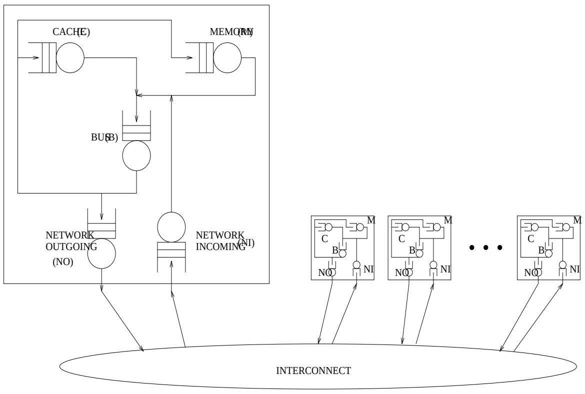

Figure 2: This figure shows the view of the hardware as seen by the model.

The system is closed and has as many classes as there are nodes in the multiprocessor. That is, each processor generates its own class of customers. A customer is in the think state as long as the processor hits in the cache – as soon as a cache miss occurs, a coherency request is generated. A processor can have any number of outstanding requests up to some maximum, the maximum being an input parameter of the model. Other input parameters include the probabilities of different request types, routing probabilities, and deteriminstic service times. The primary output of the model is mean response time for a cache miss (per processor).

An interesting aspect of this analytical model is that it allows specifying a different set of input parameters for each processor. That is, each processor has its own think time, maximum number of outstanding requests, probabilities of request types, and routing probabilities. If all processors behave uniformly or if only averages are known, then the same set of parameters can be used across all processors. However, detailed knowledge of sharing patterns in specific applications can be be modeled accurately using per processor statistics.

3.2 Classes, Request Types, and Subactions

As mentioned previously, customers from a given processor form a class of customers – this allows distinguishing customers originating from different processors. It is useful to make further distinctions even within a given class:

- *Request Types*: primary request type is defined by two components – whether the customer is a read or write, and the state of the block at the time of the access. For example, the types $RNSL$, $RDSL$, $RCSL$, and $RFSL$ respectively refer to "read (R) to a block for which no sharing list (NSL) exists", "read (R) to a dirty sharing list (DSL)", "read (R) to a fresh sharing list (FSL)", and "read (R) to a clean sharing list (CSL)". The primary type of a customer partly determines which resources will be visited. Distinguishing by primary type is merely a convenience in writing out the response time expressions (divide-and-conquer strategy). There are also secondary request types which are described in the next section – secondary requests are asynchronous requests spawned by primary request types.

- *Subactions*: an SCI coherency transaction consists of multiple low-level messages. For example, a $WDSL$ customer type will generate a memory read request, a memory read response without data, a cache read, a cache read response with data, and any number of invalidate messages. It is necessary to distinguish between these different *subactions* because they may have different service times at resources in the system.

Request types and subactions will be revisited in Section 3.5, where notation is described in detail.

3.3 Asynchronous Tasks

Certain operations in the SCI protocol are asynchronous, or *non-blocking*. A non-blocking operation is one which is performed without stalling the processor. In other words, the operation is performed asynchronously and in parallel with respect to the processor. Here are some examples of asynchronous tasks:

- With an aggressive write buffer design, all stores can be treated as non-blocking even if the stores miss in the cache (blocking only occurs when the buffer is full).

- Another option for store misses is to block until the processor becomes the head of the list. The cleanup phase, i.e. invalidations of other readers in the list, can then proceed assynchronously.

- On a read miss to a fresh sharing list, the overall coherency transaction is performed in two steps: fetching the data from memory and prepending to the list. The prepending phase does not have to block the processor since it occurs after the data has arrived from memory.

- If a processor performs a write and it is in the middle of a dirty sharing list, the processor must remove itself from the list, access memory, prepend to the list as head, and issue invalidates. The process of leaving the list (notifying neighbors) can proceed in parallel with all other operations.

- A cache miss may cause the writeback of a dirty cache block (due to replacement). Writebacks can be done in the background.

As far as the analytical model is concerned, asynchronous tasks do not appear directly in the response time expressions since they operate in parallel with primary customers. The analytical method used to model asynchronous tasks is the same as that developed in [12]. An asynchronous task is modeled as an open class of *secondary* customers. The arrival rate of this open class is set equal to the throughput of the primary customer type which spawns the secondary customers. Knowing the throughput and service times of the secondary tasks allows utilization of resources to be computed. The service times of all closed classes are inflated to reflect the utilization of resources by open classes.

3.4 SCI Options Modeled

The SCI coherence specification is flexible in that it provides sets of options instead of just one protocol. The analytical model developed in this paper uses the base protocol, i.e. *only dirty lists are supported.* However, the model notation is general enough that it is straightforward to add support for fresh and clean sharing lists. The decision to model only dirty sharing lists stems mainly from the fact that the SCI simulator on the Wind Tunnel also does not support fresh or clean lists.

In addition to the base protocol, the *asynchronous cleanups* option is also modeled. The motivation is that for reasonable performance, stores should proceed as soon as the cache block is fetched and should not have to wait for serial invalidates (this is also referred to as relaxed consistency, since strict ordering of stores is no longer guaranteed).

3.5 Notation

This section introduces the notation used in the model expressions.

3.5.1 Types

t: primary or secondary customer type

- primary
  - $RNSL$ – read to no sharing list
  - $RDSL$ – read to dirty sharing list
  - $WNSL$ – write to no sharing list
  - $WDSLn$ – write to dirty sharing list, writer is a newcomer or in the middle of the list
  - $WDSLh$ – write to dirty sharing list, writer is at the head of the list
- secondary
  - inv – invalidates

3.5.2 Subactions

s: subaction

- $mread64\_rq$: memory read request for a block
- $mread64\_rs$: memory read response with data
- $mread64\_rs\_f$: memory read response with no data (f is for forward)
- $mread00\_rq$: memory read with control information
- $mread00\_rs$: response to $mread00\_rq$
- $cread64\_rq$: cache read request for a block
- $cread64\_rs$: cache read response with data
- $cread00\_rq$: cache read with control information (e.g. invalidate request)
- $cread00\_rs$: response to $cread00\_rq$ (e.g. invalidate acknowledgement)

3.5.3 Resources

r: resource

- C_i: cache resource at node i
- B_i: bus resource at node i
- M_i: memory/directory resource at node i
- NO_i: outgoing network interface resource at node i
- NI_i: incoming network interface resource at node i

3.5.4 Inputs

- P: number of nodes in the system
- N_i: number of customers at node i (maximum number of outstanding requests)
- $\overline{Z}_i$: mean time between miss requests for node i (think time)
- $S_{r,s}$: deterministic service time for subaction s at a resource of type r
- $p_{i,t}$: probability that a primary customer from node i is of type t
- p_{ij}: probability that a primary customer from node i is initially routed to node j (j is the home node)
- $p_{ij=head}$: probability that a primary customer from node i sees node j at the head of the sharing list (j is the head node)
- $\overline{I}$: mean number of invalidates caused by writes to dirty sharing lists
- NL_s: network latency for subaction s

3.5.5 Performance Measures

- $\overline{R}_i$: mean response time for node i customers
- $\overline{R}_{i,t}$: mean response time for a customer from node i of type t
- $\overline{R}_{i,t,s,r}$: mean residence time at resource r for subaction s of a type t customer from node i
- $\overline{Q}_{i,t,s,r}$: mean queue length at a resource r due to subaction s of a type t customer from node i
- $U_{i,t,s,r}$: utilization of a resource r by subaction s of a type t customer from node i
- $\overline{W}_{i,r}$: mean waiting time for a node i customer at resource r
- $\lambda_{i,t}$: arrival rate for secondary tasks of type t which were spawned by node i primary customers
- $U_{r,<asynch>}$: utilization of a resource r by all secondary customers in the system

3.6 Equations

The next three sections derive the equations at the heart of the SCI analytical model. General response time expressions are presented first, followed by equations which model the asynchronous tasks (invalidates). Lastly, the iterative AMVA solution is presented.

3.6.1 Response Time Expressions

The overall round-trip time for a request from node i can be written as the sum:

$$\overline{R}_i = \overline{Z}_i + \overline{R}_{i,RNSL} + \overline{R}_{i,RDSL} + \overline{R}_{i,WNSL} + \overline{R}_{i,WDSLn} + \overline{R}_{i,WDSLh} \tag{1}$$

In other words, mean round-trip time is the sum of the think time of the processor and the response time contributions for each primary customer type, $\overline{R}_{i,t}$. For the derivation of $\overline{R}_{i,t}$, refer to Figure 3.

Figure 3 shows the progression of a customer in the SCI system. Each shaded circle represents a stop at a resource. Each such visit has a residence time which can be expressed as a *wait time - service time* pair, representing the queueing delay experienced by the customer and the service time at the resource:

$$\overline{R}_{i,t,s,r} = (visitcount) * (\overline{W}_{i,r} + S_{r,s}) \tag{2}$$

This equation represents the total residence time at resource r experienced by subaction s of primary customer t from node i. The visit count is a combination of routing probabilities and request type probability.

$\overline{R}_{i,t}$ is computed by summing over $\overline{R}_{i,t,s,r}$ for all resources visited and all subactions experienced in the round-trip:

$$\overline{R}_{i,t} = \sum_{r,s} \overline{R}_{i,t,s,r} \tag{3}$$

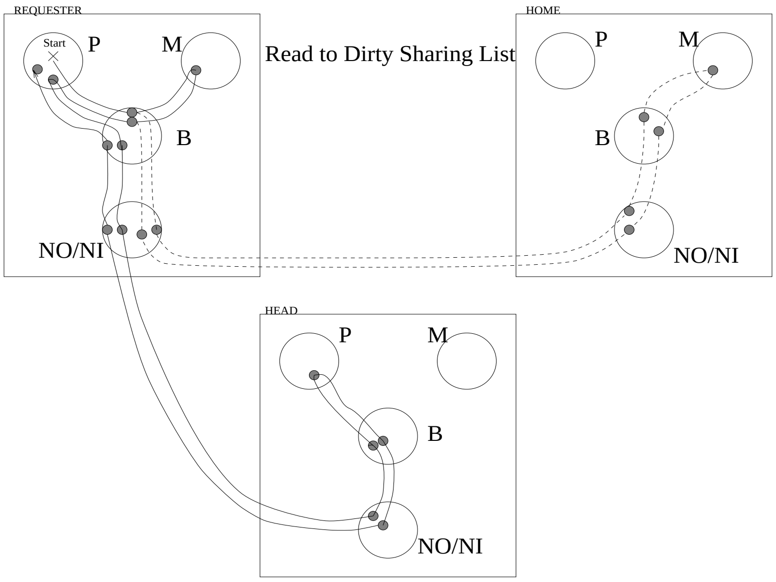

Figure 3: This figure shows the progression of a RDSL request through the system.

Using Figure 3 and the technique described above, the following equation can be derived for the reponse time contribution of primary request type $RDSL$:

$$
\begin{aligned}
(4) \qquad \overline{R}_{i,RDSL} = p_{i,RDSL} \Bigg[& (\overline{W}_{i,B_i} + S_{B,mread64\_rq}) + \sum_{j \in P} (\overline{W}_{i,M_j} + S_{M,mread64\_rq}) p_{ij} + \\
& (\sum_{\substack{j \in P \\ j \neq i}} p_{ij}) \Big\{ (\overline{W}_{i,NO_i} + S_{NO,mread64\_rq}) + \\
& (NL_{mread64\_rq} + NL_{mread64\_rs\_f}) + (\overline{W}_{i,NI_i} + S_{NI,mread64\_rs\_f}) \Big\} + \\
& \sum_{\substack{j \in P \\ j \neq i}} \Big\{ (\overline{W}_{i,NI_j} + S_{NI,mread64\_rq}) + (\overline{W}_{i,B_j} + S_{B,mread64\_rq}) + \\
& (\overline{W}_{i,B_j} + S_{B,mread64\_rs\_f}) + (\overline{W}_{i,NO_j} + S_{NO,mread64\_rs\_f}) \Big\} p_{ij} + \\
& (\overline{W}_{i,B_i} + S_{B,mread64\_rs\_f}) + (\overline{W}_{i,C_i} + S_{C,mread64\_rs\_f}) + \\
& (\overline{W}_{i,B_i} + S_{B,cread64\_rq}) + (\overline{W}_{i,NO_i} + S_{NO,cread64\_rq}) + (NL_{cread64\_rq}) + \\
& \sum_{\substack{j \in P \\ j \neq i}} \Big\{ (\overline{W}_{i,NI_j} + S_{NI,cread64\_rq}) + (\overline{W}_{i,B_j} + S_{B,cread64\_rq}) + \\
& (\overline{W}_{i,C_j} + S_{C,cread64\_rq}) + (\overline{W}_{i,B_j} + S_{B,cread64\_rs}) + (\overline{W}_{i,NO_j} + \\
& S_{NO,cread64\_rs}) \Big\} p_{ij=head} + (NL_{cread64\_rs}) + (\overline{W}_{i,NI_i} + S_{NI,cread64\_rs}) + \\
& (\overline{W}_{i,B_i} + S_{B,cread64\_rs}) + (\overline{W}_{i,C_i} + S_{C,cread64\_rs}) \Bigg]
\end{aligned}
$$

A coherence transaction conceptually involves 3 nodes, as depicted in Figure 3. The *requester node* initiates the transaction, the transaction is initially routed to the *home node* (directory node), and if the list is dirty, the *head node* gets involved as a third party. It may be that the *requester node* and *home node* are the same. Both cases are depicted in the diagram: accessing a remote home directory is shown with dashed lines, while the local memory access uses solid lines.

The first six lines of Equation 4 represent the initial directory lookup, that is, the initial memory read. In the case of $RDSL$, memory returns a pointer to the head of the dirty list. This causes a cache-to-cache transfer which is modeled by the remaining five lines of Equation 4.

The equation for $\overline{R}_{i,WDSLn}$ is identical to Equation 4 except that it uses the probability $p_{i,WDSLn}$. The equations for $\overline{R}_{i,RNSL}$ and $\overline{R}_{i,WNSL}$ look like the first six lines of Equation 4 (no cache read is necessary). Lastly, $\overline{R}_{i,WDSLh} = 0$ since the write can proceed immediately – the only reason for having this primary customer type is that it spawns invalidations.

3.6.2 Asynchronous Tasks: Invalidates

The only asynchronous tasks modeled are invalidates. As described in Section 3.3, invalidates are modeled as follows:

- The arrival rate of the "open" class is set equal to the combined throughput of writes which spawn invalidations – $WDSLn$ and $WDSLh$.

$$(5) \qquad \lambda_{i,inv} = \sum_{<writes>} \Lambda_{i,<writes>} = \Lambda_{i,WDSLn} + \Lambda_{i,WDSLh}$$

- Utilization of system resources can now be computed using Little's Law. Notice that the equations assume uniform routing probabilities for invalidate requests. That is, an invalidate spawned by node i

visits node $j, j \neq i$, with probability $\frac{1}{P-1}$. Further, the visit count is multiplied by the input parameter $\overline{I}$, the average number of nodes which must be invalidated due to a write.

(6)
$$
\begin{aligned}
U_{i,inv,cread00\_rq,B_i} &= \lambda_{i,inv}\overline{I}S_{B,cread00\_rq} \\
U_{i,inv,cread00\_rq,NO_i} &= \lambda_{i,inv}\overline{I}S_{NO,cread00\_rq} \\
U_{i,inv,cread00\_rq,NI_j} &= \lambda_{i,inv}\frac{\overline{I}}{P-1}S_{NI,cread00\_rq} \\
U_{i,inv,cread00\_rq,B_j} &= \lambda_{i,inv}\frac{\overline{I}}{P-1}S_{B,cread00\_rq} \\
U_{i,inv,cread00\_rq,C_j} &= \lambda_{i,inv}\frac{\overline{I}}{P-1}S_{C,cread00\_rq} \\
U_{i,inv,cread00\_rs,B_j} &= \lambda_{i,inv}\frac{\overline{I}}{P-1}S_{B,cread00\_rs} \\
U_{i,inv,cread00\_rs,NO_j} &= \lambda_{i,inv}\frac{\overline{I}}{P-1}S_{NO,cread00\_rs} \\
U_{i,inv,cread00\_rs,NI_i} &= \lambda_{i,inv}\overline{I}S_{NI,cread00\_rs} \\
U_{i,inv,cread00\_rs,B_i} &= \lambda_{i,inv}\overline{I}S_{B,cread00\_rs} \\
U_{i,inv,cread00\_rs,C_i} &= \lambda_{i,inv}\overline{I}S_{C,cread00\_rs}
\end{aligned}
$$

- The utilization of each resource by all invalidations can be computed with the following sum:

  (7)
  $$U_{r,inv} = \sum_{\substack{i \\ s}} U_{i,inv,s,r}$$

- Finally, the utilizations computed in Equation 7 can be used to inflate the service times seen by primary customers (closed classes) at every resource.

  (8)
  $$S'_r = \frac{S_r}{1 - U_{r,inv}}$$

3.6.3 Iterative AMVA Solution

AMVA is an approximate iterative solution technique involving three phases [9]. In the first phase, mean response times are computed as in Section 3.6.1. In the second phase, Little's Law is used to compute utilizations and mean queue lengths at all resources and for all customer classes. In the third phase, these queue lengths and utilizations are used to compute new waiting times. The process then repeats: response times are recomputed using the new waiting times, etc. A convergence test is performed at the end of each iteration. For the current version of the analytical model, convergence is detected when the round-trip times computed in consecutive iterations differ by less than 0.001, for all processors. The set of equations below correspond to phases 2 and 3 of the iterative solution.

(9)
$$
\begin{aligned}
\overline{Q}_{j,t,s,r} &= \left(\frac{\overline{R}_{j,t,s,r}}{\overline{R}_j}\right)N_j \\
U_{j,t,s,r} &= \left(\frac{S_{r,s}}{\overline{W}_{j,r} + S_{r,s}}\right)\left(\frac{\overline{R}_{j,t,s,r}}{\overline{R}_j}\right)N_j
\end{aligned}
$$

(10)
$$
\begin{aligned}
\overline{W}_{i,r} = &\sum_{\substack{j,t,s \\ j \neq i}} \left(\overline{Q}_{j,t,s,r} - \frac{1}{2}U_{j,t,s,r}\right)S_{r,s} + \\
&\left(\frac{N_i - 1}{N_i}\right)\sum_{t,s}\left(\overline{Q}_{i,t,s,r} - \frac{1}{2}U_{i,t,s,r}\right)S_{r,s}
\end{aligned}
$$

3.7 Model Approximations and Simplifications

This section lists both model approximations (non-separable and heuristic features) and aspects of real systems which are not modeled.

The following features cause the model to deviate from exact MVA:

- FCFS centers are modeled with deterministic service times.
- The Bard-Schweitzer approximation is used to reduce solution complexity.
- The method used for modeling asynchronous tasks is not exact, as descibed in [12].

Various features of a real SCI system are not modeled, as listed below. Only the first, the interconnection network, is a simplification made due to intractability. The other features were omitted due to time constraints.

- The network is modeled as an arbitrary interconnect with fixed latency.
- Pathological cases, such as retries and infrequent end-cases which can occur under rare conditions, are not modeled. As will be shown in Section 4, retries are not so rare, so this feature is added later.
- Contention in the cache between processor hits and external requests is ignored (however, this is generally only a concern in snoopy protocols).
- For a write by a node in the middle of a sharing list, the model ignores the two asynchronous transactions which are initiated to remove the node from the list (messages to right and left neighbors).
- Asynchronous writebacks (writing a dirty line back to memory due to a cache block replacement) are not modeled.

4 Validation

In order to determine the accuracy of the SCI analytical model, mean response times computed by the model are compared against the response times obtained from a detailed SCI simulator. The detailed simulator serves two functions. First, it is used to obtain detailed (per processor) input parameters for the analytical model. The simulator was instrumented to obtain routing probabilities, customer type probabilities, think times, and mean number of invalidations per write. The second function of the simulator is to measure actual mean response times for cache misses. Additional instrumentation was put in place to measure response times.

The SCI simulator only models two kinds of resources – caches and memory. Therefore, service times for the network interface and bus are set to zero in the analytical model to hide these resources. Table 1 shows the service times used by the analytical model. The service times match all costs used by the simulator (these costs were verified by precisely measuring the miss penalties for a microbenchmark).

resource	$mread64\_rq$	$mread64\_rs$	$mread64\_rs\_f$
CACHE	8	19	8
MEMORY	15	16	0

resource	$cread64\_rq$	$cread64\_rs$	$cread64\_rs\_f$	$cread00\_rq$	$cread00\_rs$
CACHE	24	19	8	8	3
MEMORY	0	0	0	0	0

Table 1: Cache and memory service times (in cycles) for relevant subactions. These are taken from the SCI simulator: 3 cycles for every cache access, 5 cycles to issue a message (setup), 16 cycles for fills (1 per word), 10 cycles for directory access.

Four SPLASH benchmarks are used in the validation process. These are *ocean*, *barnes*, *mp3d*, and *water*. All experiments are run on 32 processors with 1MB 4-way set associative L2 caches and a network latency of 91 cycles.

4.1 Initial Comparison

Results of the simulator–model comparison are presented both as absolute response times – Figures 6 through 9 – and average percent errors in Table 2. There are two sets of numbers for the analytical model; this section considers only the base model.

Benchmark	base model	model w/ retries
ocean	-17.4	+5.6
barnes	-14.0	-5.4
mp3d	-23.0	-20.1
water	-31.3	-30.3

Table 2: Average percent error in the analytical model response times relative to simulator measurements.

Table 2 shows that for the SPLASH benchmarks, the analytical model underestimates response times by 14% to 30%. These errors are surprisingly high considering that the inputs to the analytic model come straight from the instrumented simulator. Further, one would expect higher accuracy due to having detailed, per processor statistics – that is, the inputs to the model should reflect precise sharing patterns exhibited by the benchmarks. It is apparent that the analytical model is not seeing as much resource contention as the simulator. The next section investigates one possible explanation for the discrepancy.

4.2 Effect of Retries

Certain aspects of a real SCI machine were intentionally left out of the analytical model due to time constraints and modeling complexity, despite the fact that their impact on performance is unknown. One feature of SCI that was skipped over due to simplifying assumptions is the *retry* – there are certain cases where a processor must reissue its request because the destination node could not service the initial request. Such pathological cases could occur frequently enough to cause a noticeable increase in mean reponse time. To investigate the effect of retries, additional probes were placed in the simulator to detect their occurrence (the symptom is a larger than expected number of cache read responses without data). Table 3 shows the mean number of retries before a cache read completes successfully.

Benchmark	avg number of retries
ocean	0.599
barnes	0.200
mp3d	0.077
water	0.029

Table 3: Average number of retries before a cache read completes successfully.

One can conclude from this experiment that at least for *ocean* and *barnes*, the impact of retries on performance cannot be ignored. To put this hypothesis to the test, additional visit count parameters were added to the cache read response time equations of the analytical model (also, a new subaction needs to be defined: $cread64\_rs\_f$). The column labeled "model with retries" in Table 2 shows the effect of this model upgrade. The predicted response times for *ocean* and *barnes* are both within 6% of the measured response times. *Ocean* switches from underestimating to overestimating, most likely because of its high retry visit count. Notice that the performance estimates for *mp3d* and *water* do not change much at all – this is expected since these benchmarks do not exhibit many retries.

Although the previous experiment seems convincing enough, a more detailed analysis yields even more insight. The simulator was instrumented to gather a *histogram* of cache miss response times. The histogram shows how response times are distributed, which can be much more insightful than the average. The histogram counts the frequency of every cache miss ranging from 1 to 999 cycles (an overflow counter shows there are not many misses longer than 999 cycles). The normalized histogram for *ocean* is shown in Figure 4. Notice that there are three large peaks. The first peak occurs at 237 cycles – this is exactly the response time without queueing delays for $RNSL$ and $WNSL$ requests. The second peak at 435 cycles corresponds exactly to the response time for $RDSL$ and $WDSLn$ requests. The third peak is at 633 cycles, *198 cycles away from the second peak* – this is exactly the time required to retry a cache read request. Though much less noticeable, a fourth peak (bump) appears at 831 cycles, which is 2*198 cycles away from the second peak (two retries).

The histogram for *ocean* also seems to indicate that there is not much contention for resources, because of the sharp peaks at points with no queueing delay. There is some queueing, however, indicated by the small peaks which trail off after each large peak.

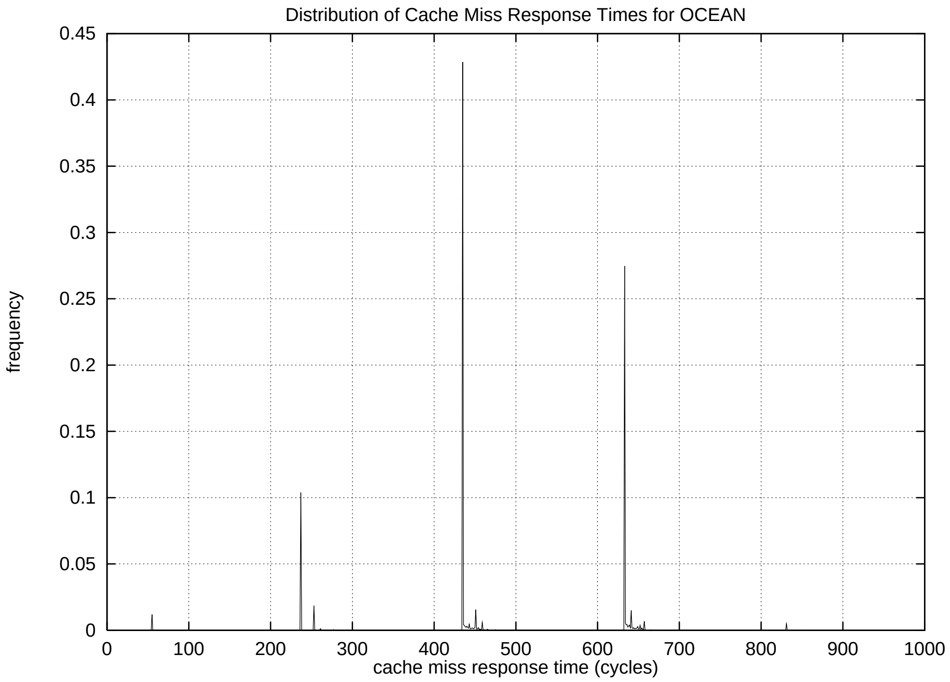

Figure 4: This figure shows the distribution of cache miss response times for ocean.

4.3 Hypothesis for Large Errors in *Mp3d, Water*

Despite accurately predicting mean response time for *ocean* and *barnes*, the large errors still remain for the other two benchmarks. Table 4 in Section 5.1 gives a clue to the problem. This table shows the average request type probabilities for each application. In particular, make note of the last column P_{WDSLh}: the probability that a request is a write to a dirty sharing list, the writer being at the head of the list at the time of the write. P_{WDSLh} for *mp3d* and *water* is significantly higher than for the other two benchmarks; in fact, it would seem that the probabilities of 0.21 and 0.31 are in some way related to the 20% and 30% errors seen for these benchmarks.

The analytical model assumes that a request of type $WDSLh$ contributes zero time to the overall mean response time, since the only transactions generated are asynchronous invalidates. However, by isolating $WDSLh$ requests in the simulator, it has been determined that the measured response time contribution is *not* zero. This leads to the hypothesis that either the analytical model does not match the actual SCI protocol with respect to $WDSLh$ requests, or the instrumentation of the simulator is wrong in this case. In either case, the problem is simple to resolve.

5 Sample Benchmark Statistics

The statistics gathered for the purpose of parameterizing the analytical model are interesting to study in and of themselves. This section presents just a few such statistics: request type probabilities, routing characteristics, and mean number of invalidates per write.

5.1 Customer Types

Table 4 shows the average probabilities for each of the primary customer types. The interesting point to note is that the majority of accesses are to cache blocks in dirty sharing lists – this says there is a lot of sharing of data.

benchmark	P_{RNSL}	P_{RDSL}	P_{WNSL}	P_{WDSLn}	P_{WDSLh}
ocean	0.0086	0.4116	0.1098	0.4433	0.0267
barnes	0.0018	0.5874	0.0294	0.2878	0.0936
mp3d	0.0596	0.2626	0.0084	0.4569	0.2125
water	0.0006	0.3606	0.0021	0.3245	0.3121

Table 4: Breakdown of customer request types per application.

5.2 Invalidates

Table 5 shows the mean number of invalidates per write (not including $WNSL$) for each benchmark. The interesting point to note here is that although writes to dirty sharing lists are frequent, there are typically only 1 or 2 other sharers in the list.

benchmark	$\bar{I}$
ocean	1.04
barnes	2.06
mp3d	1.06
water	1.07

Table 5: Mean number of invalidates per write.

5.3 Sharing Patterns

The routing probabilities $p_{ij=head}$ give some insight into the sharing patterns which exist between processors. Figure 5 plots $p_{ij=head}$ for an arbitrary node i. It is apparent that *ocean* exhibits strong near-neighbor sharing. *Mp3d* has a recurring pattern of peaks every fourth processor. This kind of information may prove useful in interpreting the performance of benchmarks running on an SCI platform (or any other coherence protocol).

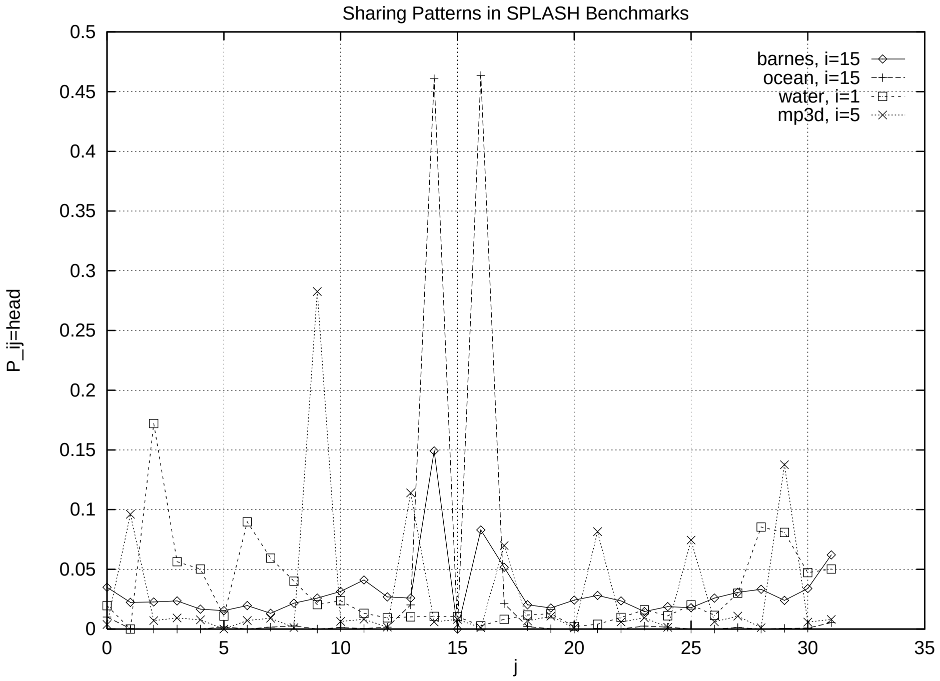

Figure 5: This figure shows routing probabilities for each of the benchmarks, for select nodes.

6 Conclusion

In this paper an analytical model for predicting mean response time of coherence transactions in an SCI system is presented. The model features flexible parameterization in that per processor statistics, detailed routing probabilities, and detailed service times can be used. In addition to deriving the model, experiments with a detailed SCI simulator were performed for obtaining input parameters and determining the accuracy of the model. The following conclusions can be drawn from the results:

- The model predicts mean response time to within 6% for two of the four SPLASH benchmarks used in the experiments, *ocean* and *barnes*.
- It was found that in order to obtain the above accuracy, *retries* have to be modeled. 60% of the cache read operations in *ocean* need to be reissued once, and occasionally twice, before data is returned.
- The model underestimates performance by 20% and 30% for *mp3d* and *water*, respectively. The apparent reason is that $WDSLh$ requests are not modeled according to the protocol, or the simulator is not instrumented properly for this request type.
- The process of instrumenting the simulator to obtain model input parameters also led to some interesting results. Routing probabilities show strong near neighbor sharing in some benchmarks, and regular patterns in others. Access type probabilities show that the majority of reads and writes are to dirty sharing lists (i.e. a lot of sharing). For three of the benchmarks, writes to dirty sharing lists on average only need to invalidate one processor – this may indicate either a pairwise sharing or a migratory sharing pattern (data passes from one node to the next).

7 Future Work

- The first issue that should be addressed is the error in response times for *mp3d* and *water*. The problem has been identified; all that remains to be done is implementation of the fix.
- None of the experiments stressed queueing in the analytical model, most likely because of the high network latency and think times. Additional experiments need to be run to emphasize waiting times. (Decrease cache size, decrease network latency, use non-uniform routing probabilities, etc.)
- The thrust of this paper has been *validation* of the model, rather than *sensitivity analysis* – for example, showing how response time varies with the number of nodes or showing the effect of a hot spot. The model can be used in many interesting ways to obtain useful results – and because it is an analytical model, turnaround time is quick (1 or 2 minutes).
- One of the unique features of the model is the ability to express detailed routing probabilities and other input parameters. The problem is that in most cases, these details will not be known; either uniform probabilities or educated guesses will be used. An interesting but extensive project would be to develop a way to extract these low-level parameters from higher-level parameters which are more related to the source program. Greenberg and Mitrani did this for snooping caches [13] by modeling the state of cache blocks with a Markov chain, and using a queueing network below the Markov chain. The Markov chain takes as input the high-level parameters and produces low-level parameters for use in the network model.

8 Acknowledgements

I am most grateful to Alain Kagi for pointing me in the right directions for instrumenting the simulator; Emmanuel Ackaouy for putting the base code in place for probing the simulator; and Todd Turnidge for taking the "bet-you-can't-write-a-statistics-parser-in-15-minutes" challenge.

References

[1] "IEEE Standard for Scalable Coherent Interface (SCI)", IEEE Computer Society, August 1993.

[2] Alain Kagi, Douglas C. Burger, and James R. Goodman, "Techniques for Reducing Overheads of Shared-Memory Multiprocessing," International Conference on Supercomputing , July 1995.

[3] Douglas C. Burger and David A. Wood, "Accuracy vs. Performance in Parallel Simulation of Interconnection Networks," ACM/IEEE International Parallel Processing Symposium, April 1995.

[4] A. Argarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," Proceedings of the 15th Annual Symposium on Computer Architecture (June 1988), pp. 280-289.

[5] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Henessy, Mark Horowitz, and Monica S. Lam, "The Stanford Dash Multiprocessor," IEEE Computer, 25(3):63-79, March 1992.

[6] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," Computer Architecture News, 20(1):5-44, March 1992.

[7] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," Proceedings of the 1993 ACM SIGMETRICS Conference, May 1993.

[8] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 64-75, April 1989.

[9] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik, *Quantitative System Performance: Computer Systems Analysis Using Queueing Network Models*, Prentice-Hall, 1984.

[10] Men-Chow Chiang and Gurindar S. Sohi, "Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput-Oriented Environment," IEEE Transactions on Computers, March 1992.

[11] Scott T. Leutenegger and Mary K. Vernon, "A Mean-Value Performance Analysis of a New Multiprocessor Architecture," ACM, 1988.

[12] P. Heidelburger and K. S. Trivedi, "Queueing Network Models for Parallel Processing with Asynchronous Tasks," IEEE Transactions on Computers, November 1982.

[13] A. G. Greenberg and I. Mitrani, "Analysis of Snooping Caches," Proc. Performance '87.

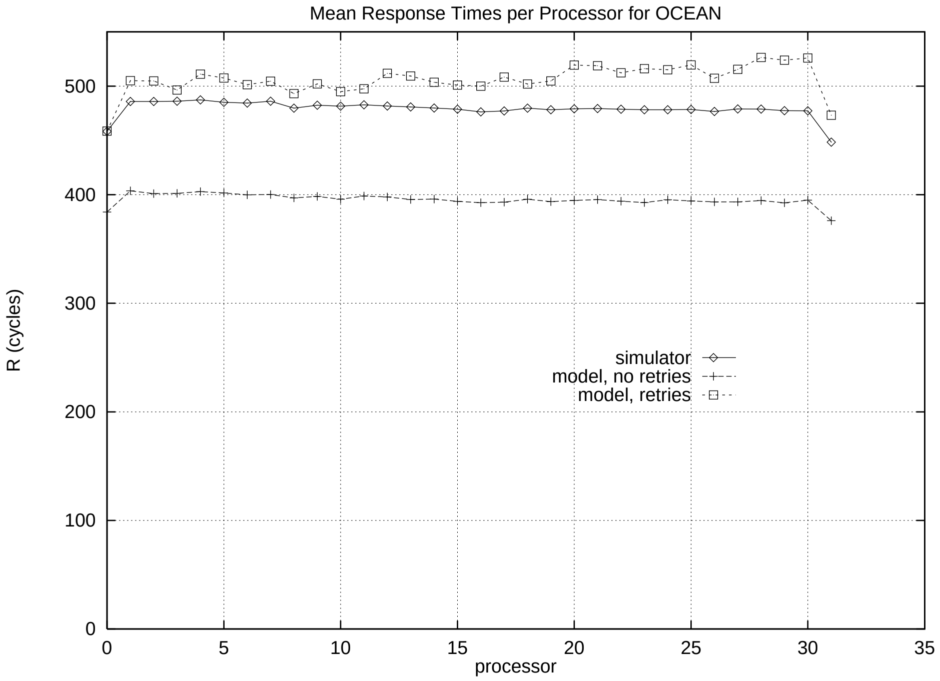

Figure 6: This figure shows mean response times from the simulator and the analytical model, for ocean.

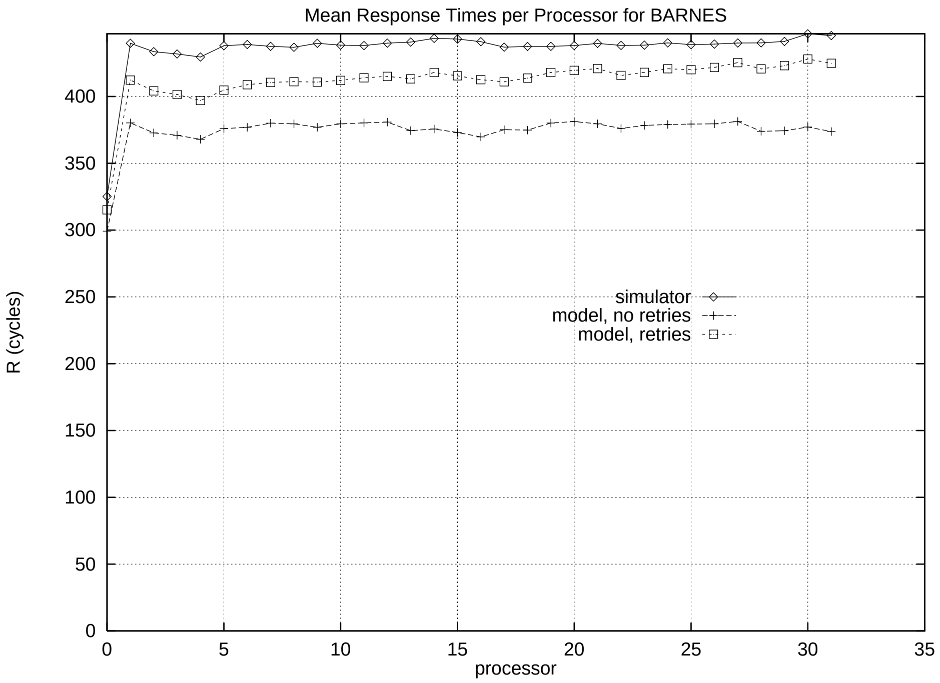

Figure 7: This figure shows mean response times from the simulator and the analytical model, for barnes.

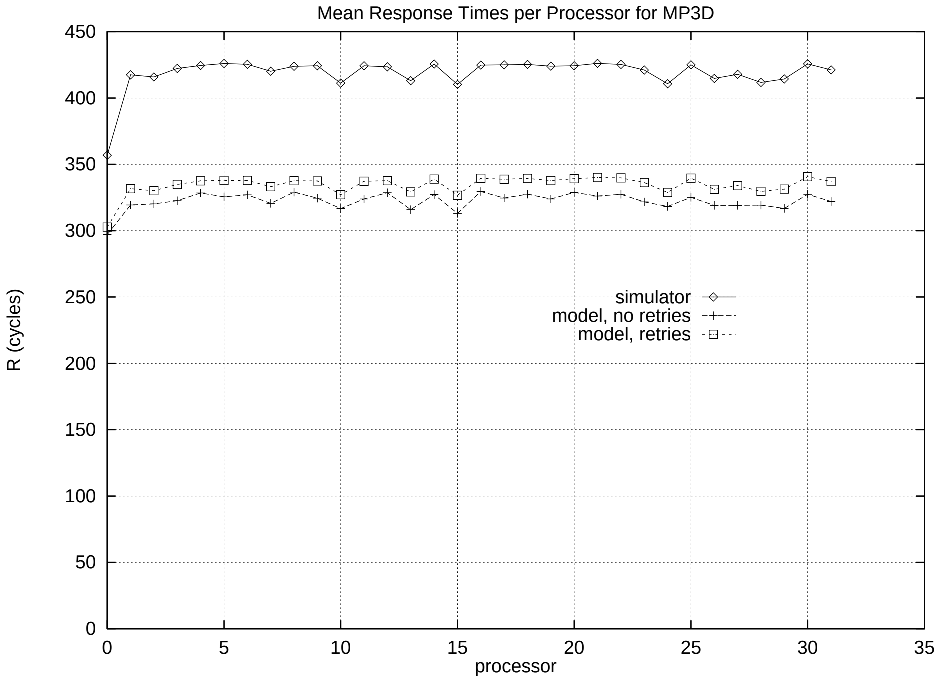

Figure 8: This figure shows mean response times from the simulator and the analytical model, for mp3d.

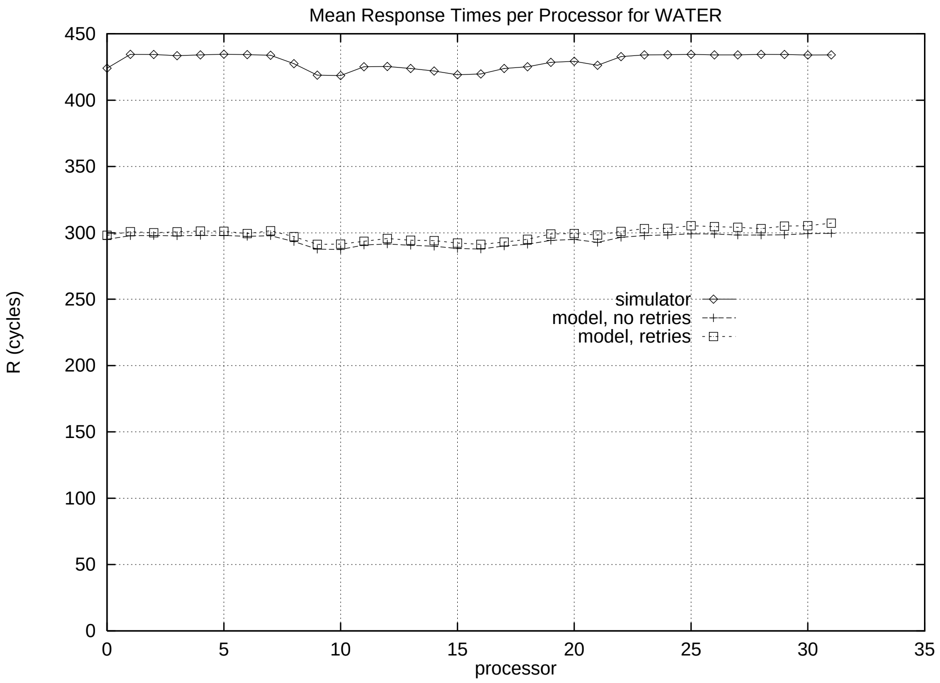

Figure 9: This figure shows mean response times from the simulator and the analytical model, for water.