# The TREEBus Architecture and Its Analysis

Rajeev Jōg
Gurindar S. Sohi
Mary K. Vernon

Computer Sciences Technical Report #747

February 1988

# The TREEBus Architecture and Its Analysis †

Rajeev Jōg
Gurindar S. Sohi
Mary K. Vernon

Computer Sciences Department
1210 W. Dayton St.
University of Wisconsin-Madison
Madison, WI 53706

*Abstract*

This paper investigates a cache coherence protocol for a hierarchical, general-purpose multiprocessor that we refer to as the TREEBus architecture. We apply the concept of non-identical dual tag directories to the specification of the protocol. We then develop a sophisticated mean-value queueing model that considers bus latency, shared memory latency and bus interference as the primary sources of performance degradation in the system. A unique feature of the model is its choice of high-level input parameters, which can be related to application program characteristics. Another nice property of the model is that it can be solved for very large systems in a matter of seconds.

Using the model we reach several important conclusions. First, the system topology has an important effect on overall performance. We find optimal two-level, three-level and four-level topologies that distribute the load uniformly across all bus levels, resulting in much higher performance. For a particular system size, the optimal topology is the topology with the minimum number of levels that has sufficient total bandwidth in the bus subsystem to support the memory request rates of the processors. Second, we provide processing power estimates for these topologies, under a given set of workload assumptions. For example, the optimal three-level TREEBus topology, supporting 512 processors each with a peak processing rate of 4 MIPS, assuming 22% of the data references are to globally shared data and that the shared data is read on the average by a significant fraction of processors between write operations, provides an effective 1400 (1700) MIPS in processing power, if the buses operate at 20 (40) MHz. Results of our study also indicate that for reasonably low cache miss rates (3% at level 0), and 20 MHz buses, the bus subnetwork saturates with processor speeds of 6-8 MIPS, at least for topologies of five or fewer levels. Finally, we present experimental results that indicate how performance is affected by the locality and pattern of sharing, and by the miss ratios that are due to cache size limitations.

## 1. Introduction

The demand for increased general-purpose computing power has led to the development of shared memory multiprocessors. In these systems, processors are connected to shared memory via some type of high-performance interconnection network. With processors based on the von Neumann model of computing, the speed of each individual processor is limited by the bandwidth of the processor's memory subsystem. The use of high-speed local memory greatly improves the memory bandwidth in such systems. If the local memory operates as a cache for

† This work was supported by the National Science Foundation under grant numbers DCR-8451405 and CCR-8706722, and by grants from IBM, Cray Research, INC. and the AT&T Foundation.

shared memory, then system software need not be concerned with mapping data into various local memory modules for efficient execution. However, the issue of multi-cache data coherence arises in this case [Hwan84]. When cache consistency is maintained in hardware, system software need not identify data blocks as "cacheable" (i.e. "private") and "non-cacheable" (i.e. "shared").

Currently, the only efficient hardware mechanisms for maintaining cache consistency are based on shared-bus interconnection networks. Cache controllers connected to the bus monitor the activities of the other processors on the bus, and take the appropriate actions (e.g. invalidate a cache block) to maintain coherence. Several such *snooping cache* protocols have been proposed [Good83, Papa84, Arch84, Rudo84, Katz85, Swea86, Bita86] and evaluated [Vern86, Arch86].

Until recently, shared-bus multiprocessor architectures were limited to a single bus and small-scale systems. However, within the past year several proposals have appeared for extending the snooping cache coherence protocols to large-scale multiple-bus multiprocessor architectures [Baer87, Wils85, Wils87, Good88]. In this paper we are interested in the hierarchical bus organization proposed by Wilson [Wils87]. We call this architecture, shown in Figure 1, the TREEBus architecture.

The purpose of this paper is two-fold. First, we apply the concept of *non-identical* dual tag directories to the specification of the TREEBus cache coherence protocol. Second, we have developed a sophisticated analytical model which is used to evaluate the performance of this machine. As we point out in section 2.1.2, the concept of non-identical dual directories can be used in single-level cache coherence protocols to integrate both ownership and exclusive states, using just two bits of tag information. In the context of the multi-level TREEBus system, the non-identical dual tag directory organization enables us to simplify and extend the specification of Wilson's write-back coherence protocol. Our protocol guarantees a unique responder to any request, and does not require *recall* requests (also called *flush* requests) to propagate to the lowest level caches in all cases.

Our performance model considers bus latency, shared memory latency, and bus interference as the primary sources of performance degradation in the system. Cache interference is not considered, but could be added in the future. An important feature of the model is that we have designed the input parameters to be easily related to application program characteristics. The input parameters include the fraction of memory references that are to data blocks shared locally within a cluster, the fraction of references to blocks shared globally among all processors, and the frequency that a processor finds the shared blocks it references invalid due to writes by other processors. Our

confidence in the viability of the model is derived from three sources. First, although the model is approximate and iterative, it uses a technique that was shown to be surprisingly accurate for evaluating single-bus cache-coherency protocols [Lazo88]. Second, the model produces results that can be explained, after careful thought, using intuition about how the system should behave. Third, our results in Section 4.1 agree with Wilson's simulation results [Wils87], assuming we have computed his input parameters correctly. Another important characteristic of the model is that the solution of the model is very efficient; performance estimates can be obtained for very large systems in a matter of seconds. We have used it to investigate systems with up to 2048 processors in this paper.

Section 2 presents the notion of non-identical dual tag directories, and contains a specification of the write-back cache coherence protocol. Section 3 presents our performance model and describes the input parameters and iterative solution technique. Section 4 contains the initial results of our experiments with this model. Finally, section 5 contains the conclusions of this study.

## 2. The TREEBus Architecture

In the TREEBus architecture (Figure 1), the processors form the leaves of the tree and the shared memory is at the root of the tree. All memories (with the possible exception of the global shared memory) are organized as

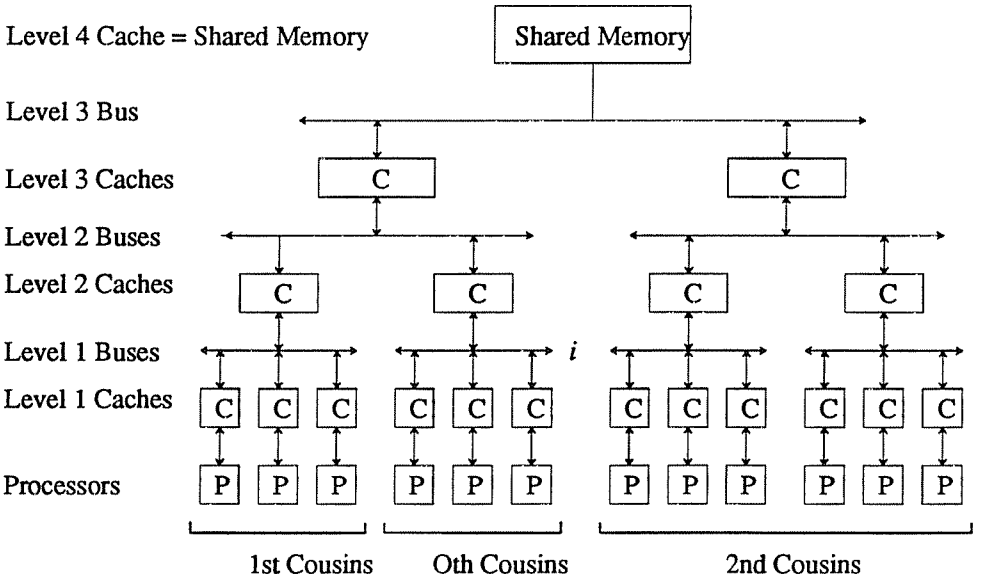


Figure 1. A 4-Level TREEBus System

caches. The processors form level 0 of the system. The cache connected to each processor is a level 1 cache. Several level 1 caches may be connected to a single bus, the level 1 bus. Such a collection of processors on a single bus forms a *super-processor* (also referred to as a *cluster* [Wils87] ). Several of these super-processors are now connected through caches to a shared bus to form the next level in the hierarchy. This can be applied recursively to an arbitrary number of levels in the system. The entire system forms a tree with a branching factor of $n_i$ at each level $i$. The *peer* caches of a cache $C$ at level $i$ are the $n_i - 1$ caches connected to the same bus. The *descendants* of a cache $C$ are all caches that are present in the subtree whose root is $C$.

If a processor request cannot be serviced by its level 1 cache, it appears on its level 1 bus. Such a request can either be serviced by peer caches or by the level 2 cache. The level 2 cache in turn may have to propagate the request to its higher level bus. As data is fetched, it filters through the levels of the tree until it reaches the cache of the requesting processor.

In order to simplify the implementation of the data-coherence protocol, the caches must satisfy an *inclusion* property [Baer87]. This means that a cache must contain tag information for all the blocks that are present in each of its descendant caches. Because of this inclusion property, the size of each cache (memory) must be quite large[1]. Given the density of modern-day dynamic RAMs, large caches for this purpose are feasible. In section 2.1, we look at the organization of the cache directories, and in section 2.2 we specify a data-coherence protocol for TREEBus. For the remainder of this section, a "processor" refers to an individual processor at level 0 or to the appropriate "super-processor" at a higher level.

### 2.1. Cache Directory Organization

In order to maintain data coherence, the tag for each block in the cache tag directory contains extra bits that identify the current state of the block. Since both the processor and the snooping mechanism need to access the tag directory, a single tag directory can easily become a bottleneck. This problem is usually resolved by maintaining two identical copies of the tag directory, one to service processor requests and one to monitor bus transactions. Both directories are simultaneously updated in one atomic operation. This is the concept of *identical dual directories* that is used by most published data coherence protocols. Unfortunately, the restriction of maintaining identical dual directories can complicate the state machine for the protocol.

[1] Wilson suggests that the cache size for a level $i$ cache is an order of magnitude times the sum of the level $i-1$ cache sizes [Wils87].

### 2.1.1. Non-Identical Dual Directories

Each cache in the system services four distinct types of requests: (i) Processor Reads, (ii) Processor Writes, (iii) Bus Reads and (iv) Bus Writes. Bus Writes may be either read-with-the-intent-to-modify (*READ-MOD*) operations or *INVALIDATE* operations, as explained in section 2.2. For processor requests, the cache needs to determine if it *can service* the request, i.e, is the desired data valid and accessible in the cache? This is determined on the basis of the tag bits in the *processor-side* directory, henceforth referred to as the *processor* directory. For bus requests, the cache needs to determine if it *should respond* to a particular bus transaction, i.e., (i) does the cache need to alter the state of a block and (ii) should it supply data in response to the bus request? This is done using the tag bits in the *bus-side* directory, henceforth referred to as the *bus* directory.

Since the two directories of the cache serve distinct purposes, we choose to distinguish between the information that is kept in the two directories in our TREEBus coherence protocol. This facilitates a uniform protocol state description for all levels of the TREEBus hierarchy. The concept of non-identical dual directories can also be used to derive simpler protocols for single-level systems; we comment briefly on this idea in the next subsection.

### 2.1.2. Directory States for Each Cache Block

The processor directory is mainly concerned with the validity and accessibility of data. Since our intention is to present *a* protocol that distinguishes between the tasks of the two directories, we currently use the following three states for a block in the processor directory[2]:

(1) **Clean**: The processor can read from the block but cannot write into the block without informing other caches. No write-back is necessary if the block gets purged.

(2) **Dirty**: The processor can read and write from this block without informing anybody else. A dirty block purged from a level $i$ cache must be written back into the level $i+1$ cache.

(3) **Invalid**: The block does not contain any valid data.

The bus directory is mainly concerned with: (i) supplying up-to-date data to other processors and (ii) invalidating copies of data that it has and might be present in its descendant caches. In our protocol, a cache block

[2] Additional states used by published protocols for single-level systems are not clearly advantageous for a multi-level system. For example, the *exclusive* state will reduce invalidation traffic on the buses, but will increase the traffic in many cases where one processor reads a block that is held in state exclusive, but not modified, by another processor in the multi-level system. We do not rule out the possibility of extending our protocol to include this and other states in the future.

in the bus directory can be in one of the following four states:

(1) **Owner Updated Sublet (OUS)**: This cache is responsible for supplying data in response to a bus request. The cache has an updated copy of the data which it can supply right away. For a write request, the cache must also invalidate copies of the block that have been supplied, i.e. *sublet* to its descendant caches.

(2) **Owner Not Updated Sublet (ONS)**: This cache is responsible for supplying data but it does not have the latest copy of the data. It must obtain the up-to-date date from one of its descendants, i.e., carry out a *recall* operation. In case of a write request, it must also invalidate copies of the block in its descendants. For a level 1 cache, the **ONS** state is merged with the **OUS** state.

(3) **Non-Owner Valid Sublet (NVS)**: The cache is not responsible for supplying the data. However, since the cache and possibly its descendants have a copy of the block, it must invalidate all such copies on a bus write request.

(4) **Invalid (I)**: The cache does not need to perform any action and can ignore the bus transaction.

The reader should note that if the concept of non-identical dual directories were applied to a single level system, both the exclusive state and the ownership information can be maintained using just 2 tag bits for each cache block in each directory. In this case, the processor directory would have states **Exclusive**, **Dirty**, **Clean** and **Invalid** and the bus directory would have states **Owner**, **Non-Owner** and **Invalid**.

### 2.2. Operation of the TREEBus Data Coherence Protocol

In a multilevel system, a single request originating from a leaf processor may manifest itself as belonging to different request types at different levels in the system. For example, a read request generated by a processor at level 0 of TREEBus appears as a Processor Read for the level 1 cache connected to the processor. If this request cannot be serviced by the cache, it will appear as a Bus Read on the level 1 bus. If no peer cache connected to the level 1 bus can satisfy the request, the level 2 cache connected to the bus treats the request as a [super-]Processor Read and so on till the request is serviced.

The operation of the protocol is described in terms of the state transition diagrams for the processor directory and the bus directory, shown respectively in Figures 2(a) and (b). In these figures, "memory" refers to the level $i$+1 cache, assuming the protocol is being applied to a level $i$ cache. Of course, the "memory" at the root of TREEBus, i.e., the highest level, is the global shared memory.

#### 2.2.1. Read Requests

A *READ* request is a *hit* if the block's state in the processor directory is either **clean** or **dirty**. If a *READ* request hits in a cache at level $i$, the requested block is read from the cache. If the block is not present in the cache

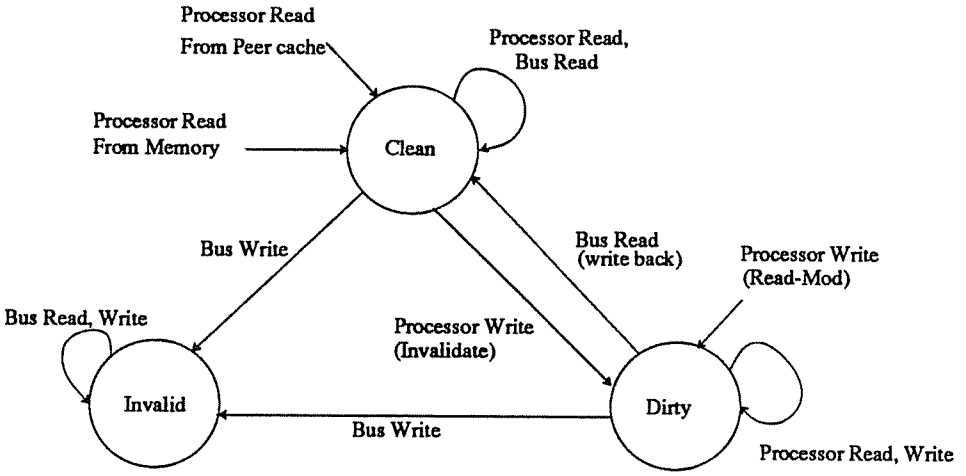


Figure 2(a): State Transitions for the Processor Directory

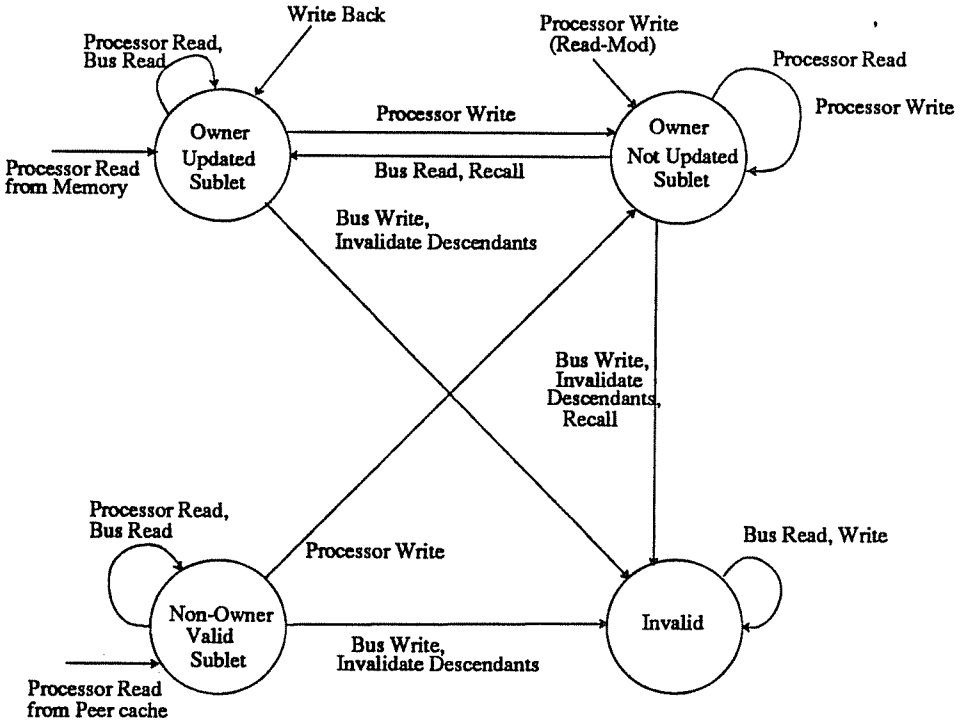


Figure 2(b): State Transitions for the Bus Directory

at level $i$, the request propagates to the bus at level $i$. A peer cache on the level $i$ bus will respond to this request only if its bus directory indicates that it is the Owner of the block. If a supplying peer cache has the block marked **dirty** in its processor side directory, it is instead marked **clean** and the block is also written back to the level $i$+1 cache. The requesting cache marks the block **clean** in its processor directory and **NVS** in its bus directory. If no peer cache is the owner, the level $i$+1 cache is responsible for supplying the block. In this case, the supplied block

is marked as **OUS** in the bus directory of the requesting cache.

A peer cache may be the Owner of a requested block without having an up-to-date copy. In this case, the data is *recalled* from the descendant of the owner cache to whom it has been sublet, by propagating the request down to the bus at level $i-1$. Note that a read request does not purge the block from the supplying cache or its descendents. On a given bus, only one of the caches connected to it can be the owner of the block. It is possible, however, that several caches all connected to different buses may simultaneously have the same block marked in an owner state. The owner caches are responsible for servicing requests that appear on the buses they are connected to.

### 2.2.2. Write Requests and Block Invalidation

If the processor issues a write request to a block and the block is marked **dirty** in the processor directory, the write can proceed without informing anybody else. If, on the other hand, the block is marked **clean**, other caches must be informed about the write. In this case, a *INVALIDATE* request is placed on the bus. The peer caches that have a copy of the block respond by changing the states of the bus and processor directories to **invalid**; the level $i+1$ parent cache changes the state of its bus directory to **ONS** and its processor directory to **dirty**. (The **ONS** state indicates that the cache is still the owner of the cache but it no longer has an up-to-date copy of the data since one of its descendants has updated it.)

Each peer cache at level $i$ that responds to the invalidation request may also need to propagate a *INVALIDATE* request *downward* to its level $i-1$ bus, to invalidate copies of the block in its descendants. Similarly the level $i+1$ cache may need to propagate the *INVALIDATE* request *upward* to its level $i+1$ bus. Therefore, a single *INVALIDATE* request at a level $i$ bus can possibly fork into one upward and several downward *INVALIDATE* requests. The downward requests continue to propagate on all branches that have valid copies of the block. An upward *INVALIDATE* request stops propagating when it reaches a cache that is already in an **ONS** state or when it reaches the shared memory at the root of the tree.

When a level $i+1$ parent cache determines that the *INVALIDATE* request need not propagate upward, it sends an *invalidate acknowledge* signal back to the level $i$ cache. The level $i$ cache in turn propagates the invalidate acknowledge signal to the level $i-1$ cache and eventually the processor that initiated the invalidate operation receives the invalidate acknowledge signal. At this point, the processor can proceed. Note that the processor does not wait for downward *INVALIDATE* requests. It is possible that a processor might read data from a cache block that

another processor is trying to invalidate. If the read occurs before the invalidate arrives, we assume this is a valid behavior of the system.[3] Our protocol guarantees that two or more processors never update the value of the same block simultaneously. The acknowledgement from the upward *READ-MOD* or invalidate will only occur for the first of two simultaneous requests to reach a shared cache common to their propagation paths. The other *READ-MOD* or invalidation request will generate a recall request when it reaches the common shared cache. Note that our model in section 3 ignores the case of colliding invalidations.

A *READ-MOD* (or *read-with-the intent-to-modify*) request is generated when a processor writes a block that is not present in its cache. Since the block must be read into the cache of the requesting processor, and since other caches that have the block must be informed about the write, a single *READ-MOD* request is treated as a combination of a *READ* request and an *INVALIDATE* request. That is, the changes in the states of the caches that take place when a *READ-MOD* operation is encountered are the same as the changes that would take place if the processor issued a *READ* request immediately followed by an *INVALIDATE* request. Since both read and invalidate operations need to be carried out, the processor must wait until both are completed, i.e., the delay for the parent cache to respond to a *READ-MOD* request is the maximum of the delay for the read and the invalidate operations.

## 3. A Model of the TREEBus System

In this section we describe a mean-value performance model for the TREEbus architecture, including the cache coherence protocol defined in the previous section. The model is an extension of the "customized mean value analysis" approach used in [Lazo88], which proved to be surprisingly accurate for the analysis of single-bus cache coherence protocols.

Parameters of the mean-value model include the degree of branching at each level in the bus hierarchy, various memory miss rates, the probability that a bus request will be satisfied by a peer cache at each level in the bus hierarchy, and bus access times for different request types. We treat cache access as being conflict free, and assume that contention arises primarily at the buses in the system. We believe, based on our earlier experience in analyzing single-bus cache consistency protocols [Vern86, Lazo88], that the former is a second-order effect as compared with the latter, and can be ignored in an initial analysis of system performance.

---

[3] Note that this problem exists in any multiprocessor that has more than one level of cache memory (for example an onchip and an offchip cache) and allows caching of shared data in all caches.

In the following development of the performance model, we first consider only *READ* requests. In section 3.4 we comment on how the equations are modified to account for *READ-MOD* and *INVALIDATE* requests. The experiments described in section 4 were run using the complete model, including the extensions for invalidation traffic. Section 3.5 discusses how the parameters of the mean-value model are computed from a set of model input parameters that characterize application program behavior, and section 3.6 discusses performance measures which can be computed from the model outputs. Throughout the remainder of this section, the term "processor" refers to a level 0 processor in the system.

### 3.1. Preliminaries

#### 3.1.1. Tree Organization

Let $L$ be the total number of levels in the bus hierarchy. For each bus at level $i$, we can partition the set of processors in the system into $L-i+1$ generations of "cousins" in the tree. The processor partition for a level $i$ bus is depicted in Figure 1. The 0th generation of cousins is the set of processors in the subtree rooted at the level $i$ bus. Requests from these processors may propagate up to the level $i$ bus. The $k$th generation of cousins of a particular level $i$ bus, $L-i \geq k \geq 1$, is the set of processors that are leaf nodes of the subtree rooted at the level $i+k$ bus that is an ancestor of the level $i$ bus, excluding the subtree rooted at level $i+k-1$ that contains the level $i$ bus itself. Requests from these processors may propagate up to the level $i+k$ bus, and then down to the level $i$ bus in the form of recall or invalidation requests.

We use the following notation:

- $n_i$ is the degree of branching from level $i$ to level $i-1$.
- $N_{k,i}$ is the number of $k$th cousins that can generate requests on a level $i$ bus.

The number of 0th cousins for a bus at level $i$ is the product of the degrees of branching for levels 1 through $i$:

$$N_{0,i} = \prod_{j=1}^{i} n_j,$$

and the number of $k$th cousins, $1 \leq k \leq L-i$ for a level $i$ bus is:

$$N_{k,i} = (n_{i+k}-1) \prod_{j=1}^{i+k-1} n_j.$$

Note that the model assumes a symmetric tree.

### 3.1.2. Request Propagation Probabilities

The following miss rates are computed from model input parameters (section 3.5), for each level in the bus hierarchy:

- $m_{M,i}$ is the miss ratio in the memory at level $i$. This is the probability that the request will be transmitted on the $i+1$th bus. We use $m_{M,0}$ to denote the probability that a processor request misses in its level 0 memory (i.e., it's level 1 cache).
- $p_{C,i}$ is the probability that a *READ* request on the level $i$ bus can be satisfied by a peer cache at the same level.
- $m_{R,i}$ is the miss ratio seen by requests that will be satisfied by a peer cache at level $i$. This is the probability that a block needs to be "recalled".

A read request that must propagate to another level requires two separate operations on the level $i$ bus: an "address" operation (type *addr*), and a data transfer operation (type *data*), which is the delayed response to the address operation. A bus request that does not propagate to another level before being satisfied requires a fixed number of cycles to transmit an address plus a data block. We label this type of operation as type *both*[4].

Using the above miss rates, we can compute the probabilities that requests of various types propagate to a level $i$ bus. We use the following notation for the request propagation probabilities:

- $\delta_{k,i}$ is the probability that a request on the $i$th bus from a $k$th cousin will propagate up to the bus at level $i+1$.
- $\mu_{k,i}$ is the probability that a request on the $i$th bus from a $k$th cousin will propagate down to a bus at level $i-1$.
- $\alpha_{k,i}$ is the probability that a request from a $k$th cousin propagates to a particular level $i$ bus.
- $\beta_{k,r,i}$ is the probability that a request for the $i$th bus from a $k$th cousin will require an operation of type $r$.

The above request propagation probabilities are computed in a straightforward way from the miss rates and peer cache supply probabilities. The formulae are given in Appendix I.

## 3.2. Response Times for Block Requests

Below we develop a set of recursive equations for the total mean time between memory read requests. Each processor is assumed to continuously execute the following cycle : it spends some time executing, then makes a memory reference.

---

[4] Note that we have not explicitly modeled the bus traffic for writing back blocks that are purged in the level $i$ cache. This traffic can be included indirectly by adding the time for transmitting the purged data block, weighted by the probability that the purged block is dirty, to the time for the "address" cycle of *READ* requests from 0th cousins.

We use the following notation :

- $R$ is the mean total time between memory requests made by a processor.
- $\tau$ is the mean time a processor executes between memory requests, including the local cache access time if the request hits in the cache.
- $T_{r,i}$ is the (constant) duration of a type $r$ *READ* request on the level $i$ bus. A type $r$ request may be type *addr*, *data*, or *both*.
- $d_{cache,i}$ is the latency for retrieving a block of data from a level i cache in the system.
- $T_{supply,i}$ is the (constant) time it takes to transfer data from one side of a level $i$ cache to another.
- $t_{k,i}$ is the mean response time for class k requests at level $i$. These are the requests generated by the $k$th cousins of $i$. The response time includes the time for retrieving data from lower level buses or recalling data from higher level buses, if necessary.
- $t_{Q,k,i}$ is the mean time a request from a $k$th cousin spends waiting for and receiving service from the level $i$ bus.
- $\bar{w}_{k,i}$ is the mean waiting time at the level $i$ bus for a request from a $k$th cousin.

The following equations hold for the response times:

i) The total time between processor memory requests will consist of the processor's execution time, plus the delay to retrieve data remotely if the request misses in the cache:

$$R = \tau + m_{M,0}\,(\,t_{0,1} + T_{supply,1}\,) \tag{1}$$

ii) The mean response time for *READ* requests from a $k$th cousin at level $i$ is the sum of the time the request waits for and receives service on the bus, plus the mean delay for propagating the request to other bus levels whenever this is needed. The request will propagate to level $i-1$ and become a request from a $k+1$th cousin, with probability $\mu_{k,i}$. The request will propagate to level $i+1$ with probability $\delta_{k,i}$. At the level where the request is satisfied, there is an additional delay for the cache latency. Thus,

$$\begin{aligned} t_{k,i} = {} & t_{Q,k,i} + (1-\delta_{k,i}-\mu_{k,i})\,d_{cache,i} \\ & + \delta_{k,i}\,(t_{k,i+1} + T_{supply,i+1}) + \mu_{k,i}\,(t_{k+1,i-1} + T_{supply,i}) \end{aligned} \tag{2}$$

iii) The mean time a *READ* request spends in the queue for the level $i$ bus is the waiting time plus the service time for the request:

$$t_{Q,k,i} = \beta_{k,both,i}\,(\,\bar{w}_{k,i} + T_{both,i}\,) + \beta_{k,addr,i}\,(\bar{w}_{k,i} + T_{addr,i} + \bar{w}_{k,i} + T_{data,i}),$$

which can be rewritten as:

$$t_{Q,k,i} = \sum_{r \in \{addr,data,both\}} \beta_{k,r,i}\,(\bar{w}_{k,i} + T_{r,i}) \tag{3}$$

since $\beta_{k,data,i} = \beta_{k,addr,i}$.

### 3.3. Modeling Bus Contention

To complete the response time calculations in equations (1) - (3), we need to compute the mean bus waiting times, $w_{k,i}$, for each class $k$ and each level $i$. We apply mean value techniques for queueing network models to calculate these waiting times [Lazo84]. The approach is outlined below. The reader is referred to[Vern88] for the actual equations.

We treat the buses in the system as queues with deterministic service times, determined by request type. The fraction of time a level $i$ bus is used by requests of type $r$ generated by processors that are $k$th cousins of the bus is given by $\frac{\beta_{k,r,i}\ T_{r,i}}{R}\ \alpha_{k,i}$. The fraction of time this class of requests spends in the queue for the bus is given by $\frac{\beta_{k,r,i}(\overline{w}_{k,i}+T_{r,i})}{R}\ \alpha_{k,i}$. (The waiting time in the latter expression is obtained from the previous iteration of the model solution). Using the first expression, we can compute total bus utilization, bus utilization by each request type, and the probability that a request of a given class finds the bus busy serving a request of type $r$, for each $r$. Using the second expression, we can compute the mean number of requests of each class that are found in the queue at the instant a request of a particular class arrives. From these values, the mean waiting time for a request of class $k$ is easily computed.

The waiting time values are used in equations (1)-(3) to compute a new estimate of R. At the end of each iteration, we compare the values of all waiting time estimates with the estimates from the previous iteration. When the sum of the absolute values of the differences in the estimates is less than some very small value, $\varepsilon$, the solution terminates. For experiments with up to five bus levels and 1536 processors, the solution terminates in a matter of seconds on a DEC VAXstation II.

### 3.4. Extensions to the Model for Invalidations

The equations developed in section 3.3 considered only *READ* requests. In this section we outline the changes needed to include *READ-MOD* and *INVALIDATE* requests in the model. Again, the reader is referred to[Vern88] for further details.

We use the same notation for classes of invalidation requests which are generated by $k$th "cousins" of a level $i$ bus, but add an additional *prime* (′) notation to distinguish these requests. Thus, a class 0′ request is an invalidation request from a 0th cousin (i.e., a direct descendent) of the bus. The propagation probabilities for

invalidation requests are similar in flavor to the read request propagation probabilities in Appendix I, and are given in [Vern88].

Four extensions to the equations are required to model the invalidation traffic. First, equation (1) must be modified to include the response time for invalidation requests that are generated by write hits to clean blocks in the level 1 cache. If we let $p_{INV}$ be the probability that the processor request is a write request to a data block that is currently in state **Clean**, then the additional term in equation (1) is given by $p_{INV}\ (t_{0',1} + T_{supply,1})$.

Second, equation (2) applies for *READ-MOD* requests, but must be modified to include a term for the mean response time when a *READ-MOD* request generates an invalidation request for the next higher level. The probability that a *READ-MOD* request requires an invalidation is computed as the probability that it hits in the memory, or that it is supplied by a peer cache and is not in state **ONS** in the memory. $p_{\overline{ONS}\,|\,cachesupply,i}$, the probability the block is not in state **ONS** in the level i memory, given that the block is supplied by a peer cache, is a parameter that must be computed from the model inputs in section 3.5.

Third, equations which are analogous to equations (2) and (3), must be developed for the invalidation traffic. Invalidations generate three types of requests on the bus: type *inv*, type *ack*, and the combined type *invack*. These types must be included in the straightforward formulation of the analogs to (2) and (3).

Finally, minor changes are required to include invalidation request types in the bus waiting time equations. Care must be taken in summing over the appropriate classes (primed and non-primed) and appropriate request types in those equations.

### 3.5. Model Inputs

The parameters for the mean value model presented in the previous subsections include the branching factor at each level ($n_i$), the mean time a processor executes between memory requests ($\tau$), and the fixed delay for various operations on the bus at each level ($T_{r,i}$). These parameters are dependent on the machine topology, and various characteristics of the hardware components (e.g. processor and bus speeds).

The model also includes a number of parameters that depend primarily on the characteristics of the application programs running on the system. These parameters include the probability that a processor writes to a block in state **Clean** ($p_{INV}$), the miss rates at each level in the memory hierarchy ($m_{M,i}$), the probability of peer cache supply at each level ($p_{C,i}$), the probability that a peer cache must initiate a recall for a data block it is

supplying ($m_{R,i}$), and the probability that a data block in the memory at level $i$ is not in state ONS, given that the block will be supplied by a peer cache ($p_{\overline{ONS}\,|\,cachesupply,i}$). Given the complexity of the system being modeled, this set of parameters is fairly small. However, there are quite a few of complex interdependencies among these parameters (i.e., they cannot be varied independently). In addition, the relation between these parameters and the application level program is very indirect and non-intuitive.

We have designed a set of higher-level input parameters that relate more directly to application program characteristics, and which contain only a few very weak interdependencies. We note that the characterization of parallel program behavior is a very difficult and open problem. We propose the parameters in this section as a starting point for developing a submodel that has an optimal set of independent parameters. The lower-level model input parameters are derived from the high-level parameters in a straightforward, albeit complex, manner. Appendix II contains examples to illustrate the equations. More detail is given in [Vern88].

The high-level input parameters, and the paradigm we use to characterize the sharing of data in application programs, are as follows:

(1) $f_P$ is the fraction of memory references which are to *private* cache blocks. Private cache blocks are cache blocks which are only accessed by one process.

(2) $f_S{}^{\kappa}$ is the fraction of memory references which are to class $\kappa$ shared cache blocks. Class $\kappa$ cache blocks contain data that is shared among the processors in a subtree rooted at the level $\kappa$ bus.

(3) $f_{IW}{}^{\kappa}$ is the fraction of references to class $\kappa$ shared blocks that miss in the cache due to being written by another processor. This parameter is a measure of the frequency with which all other processors write a block, relative to the frequency at which a particular processor reads and writes the block, averaged over all blocks in class $\kappa$.

(4) $f_{RI}$ is the overall fraction of processors which *read* a shared block between writes to the block by a particular processor. At first glance, it might appear that this parameter is determined the previous parameter, $f_{IW}{}^{\kappa}$. However, this is not the case. For example, $f_{IW}{}^{\kappa}$ is small if one processor reads a block many times between another processor's write to the block. In this case, $f_{RI}$ could be large (if other processors also read the block) or small (if no other processors read the block).

(5) $f_{W\,|\,P}$ is the fraction of references to private blocks that are *write* operations.

(6) $f_{W\,|\,S}$ is the fraction of references to shared blocks (of any class) that are *write* operations. Note that the unconditional probability, $f_W$, is easily computed from parameters (1), (2), (5), and (6).

(7) $m_{T,i}$ is the "traditional" miss rate for memory references at the level i cache. This is the probability a memory reference misses in the cache if it is a reference to a private block or if it is a reference to a shared block and does not miss for cache-coherence reasons.

(8) $f_{INV\,|\,PWH}$ is the fraction of writes to private data blocks that find the block in state **Clean**. In general, we expect this parameter to be very small, since we are assuming very large caches, and since the first access to a data block by a process tends to be a write request. Note that the corresponding input parameter for writes to shared blocks that find the block in state **Clean**, is related to the $f_{IW}{}^{\kappa}$ parameters, and is computed from them in an approximate way.

We note that real values of the above application-related parameters are unknown at this time. This is partly due to the fact that the development of large-scale parallel applications necessarily lags behind the design of these systems. It is also partly due to the lack of a well-defined workload model specifying which parameters are important to measure. However, we have found the above parameters to be very useful for performing parametric studies of the TREEbus architecture. They allow us to gain an understanding of which types of parallel applications might run well on the machine. (See section 4).

### 3.6. Model Outputs

The performance measure we are primarily interested in is the overall processing power, which can be expressed as *speedup*, *efficiency*, or *effective MIPS*. We are also interested in bus utilization measures at each level in the bus hierarchy, since these measures can help elucidate causes of performance degradation. The bus utilizations are computed as described in section 3.3. Efficiency is computed as $\frac{\tau}{R}$. Speedup is computed by multiplying the efficiency by the number of processors in the system, and the effective MIPS rate is computed by multiplying the speedup by the peak MIPS rate of the individual processors.

## 4. Performance Results

In this section, we use the model developed above to analyze several aspects of the TREEBus architecture. First, we attempt to validate our model against previous simulation results. We then perform several parametric studies to evaluate overall system performance, and to gain an understanding of how performance is affected by key model parameters. We are especially interested in the question of whether, and under what application characteristics, the bus at the root of the hierarchical tree structure becomes a bottleneck.

### 4.1. Validation Against Previous Studies

Given the information in[Wils85, Wils87] , we were unable to determine precisely the input values to use in our model to corroborate Wilson's simulation results. However, using the data given, and reasoning about the application (PDE, a program that solves partial differential equations using the nearest-neighbor method), led us to the following assignment of parameter values : 1 MIPS processors, $T_{supply}$ = 100 nanoseconds, $d_{cache}$ = 1.0 microseconds, $m_{T,0}$ = 0.03, $f_{W|P}$ = 0.1 (since "private" includes instruction references), $f_{W|S}$ = 0.20 (since each data point is read by four neighbors before it is updated), $f_{RI}$ = 0.01 (since sharing occurs only between pairs of

processors), and the fraction of data references to shared blocks, $f_{S,0}^1 = f_{S,0}^2 = 4.33\%$.

Our model predicts a speedup of 110 for a 128 processor 16×8 system, which is 8% above Wilson's value of 102. Our model also estimates bus utilizations of 35.8% and 74.8% for levels 1 and 2, respectively, as compared with Wilson's values of 26% and 61%. However, we are in close agreement on the cluster hit ratio: 56.6% as compared with 57.4%. We surmise that the small differences in our results are attributable to inexact input parameters and/or to cache/memory interference effects that we have not captured in our experiment. We thus conclude that, within the accuracy of this experiment, our model agrees well with his results.

### 4.2. The Effect of Topology

We would like to know how the performance of TREEBus changes as we vary the topology of the system. In order to evaluate the effect of topology, we have solved the model for selected topologies, varying the number of processors, but holding all other inputs fixed. In this set of experiments, we use the following input parameter values:

(i) The bus speed is 20 MHz, giving a bus cycle time of 50 nanoseconds.

(ii) The processors each have peak performance of 4 MIPS, and each processor makes an average of 1.3 memory references per instruction, counting the instruction fetch. Note that $\tau$ is the reciprocal of the product of these two values, or 3.85 bus cycles.

(iii) The service times, measured in bus cycles, for the various classes of bus requests are: $T_{addr}$=2.0, $T_{data}$ = 4.0, $T_{both}$=6.0, $T_{inv}$ = $T_{ack}$ =1.0 and $T_{invack}$ =2.0. These values hold for the buses at all levels.

(iv) The memory latencies are: $T_{supply,i}$ = 100 nanoseconds, and $d_{cache,i}$ = 750 nanoseconds, for all levels $i$.

(v) The *traditional* miss ratios for the level 1, 2, 3 and 4 caches are 3%, 1%, 0.1% and 0.01%, respectively. The values of these miss ratios are based on the expected sizes of the caches at the various levels. Note that invalidations of shared blocks will result in a higher total miss ratio.

(vi) 5% of all references (or about 22% of all data references) are to blocks containing globally shared data [Dare87].

(vii) 10% of private references (including instruction references) are writes and 30% of references to shared data are writes.

(viii) The probability of finding a shared data block invalidated, $f_{IW}$, is 0.25.

(ix) 50% of the processors that share a data block read it between successive invalidations of the block.

Figure 3a shows the speedup achieved for the selected system topologies versus the total number of processors in the system. Figure 3b gives results for 40 MHz buses and $d_{cache}$ = 500 nanoseconds. Each topology is identified by the notation N1 × i × j × k, where N1 is the number of processors connected to each level 1 bus, and i (j,k) is the branching factor on the level 2 (3,4) bus. Consider, for example, the N1×12 topology in Figure 3a. When N1=20, this system has a total of 240 processors, and achieves a speedup of about 150. At this point, one or

more of the buses becomes saturated. To connect more processors, we must use an alternate TREEBus topology.

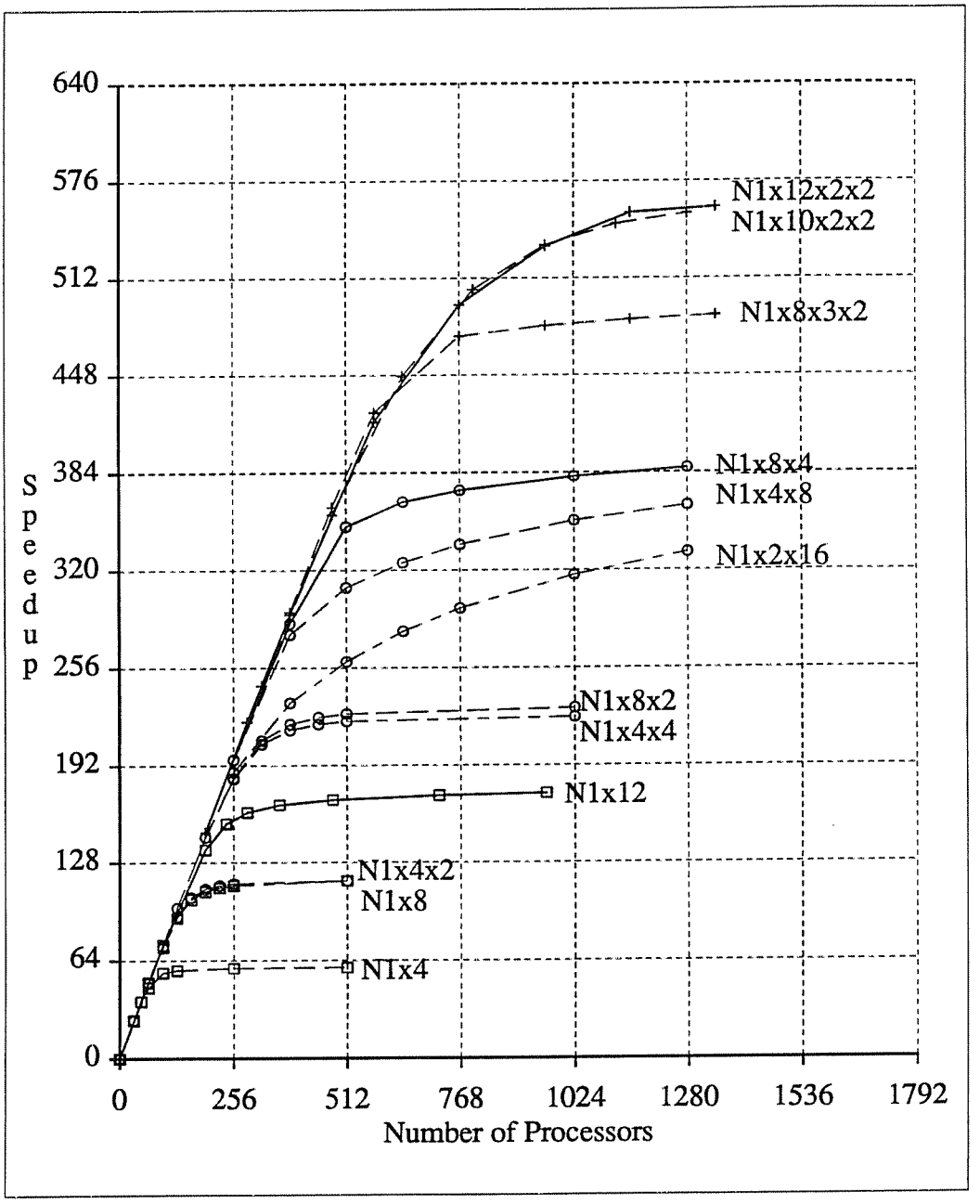


Figure 3a: The Effect of the TREEBus Topology

4 MIPS, 20 MHz, $f^{L_3} = 0.05$, $f_{IW} = 0.25$, $f_{RI} = 0.5$

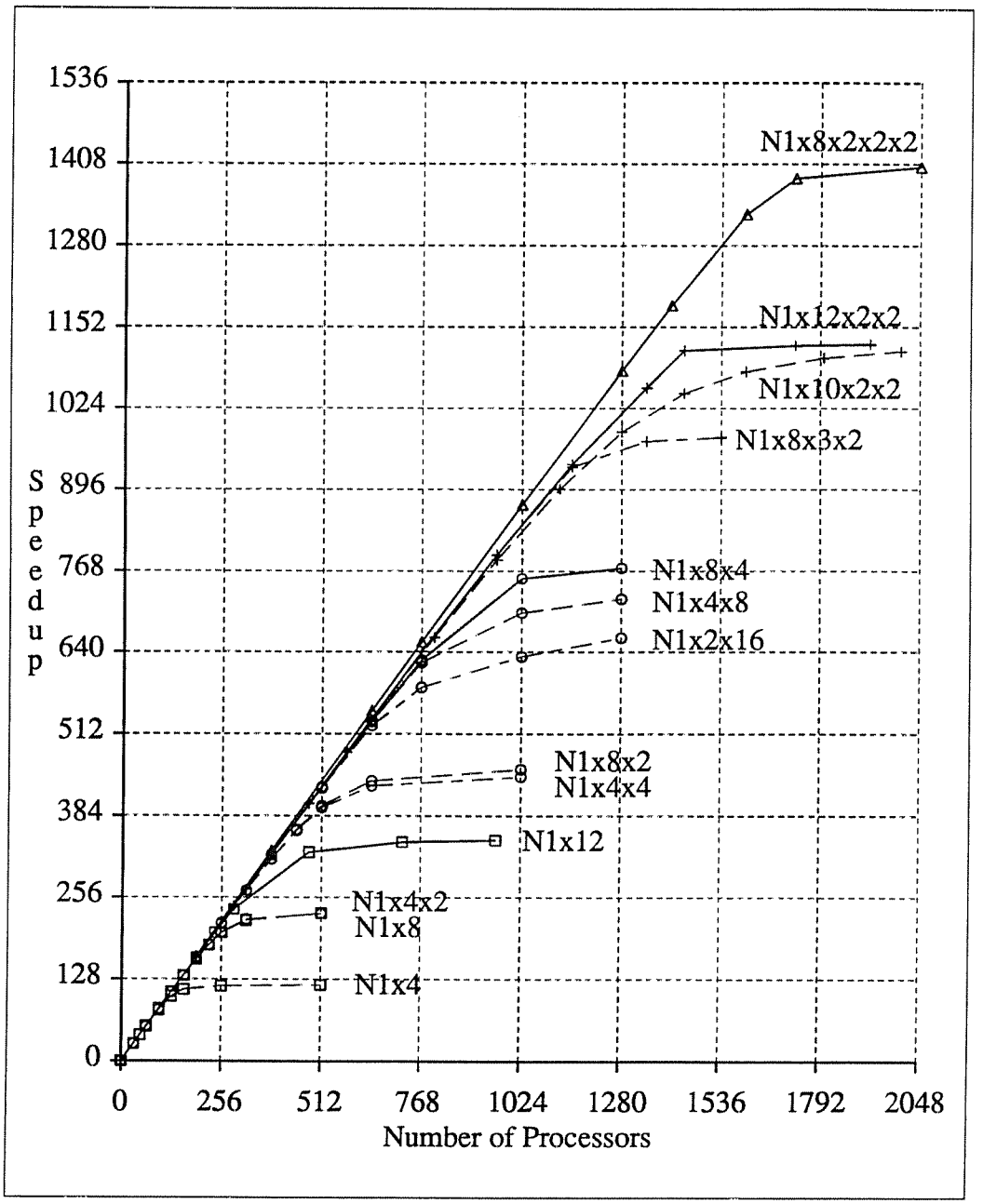


Figure 3b: The Effect of the TREEBus Topology

4 MIPS, 40 MHz, $f^{L}{}_{S} = 0.05, f_{IW} = 0.25, f_{RI} = 0.5$

Our first observation is that, for a fixed number of levels, the point at which bus saturation occurs is dependent on the topology. For example, the N1×8×4 topology saturates at 512 (768) processors, whereas the the N1×4×4

topology saturates at 256 (512) processors, if the buses operate at 20 (40) MHz. Below the saturation point, all topologies have the same performance.

The bus utilizations estimated by the model show that the topology determines which level buses become the bottlenecks in the system, and that the higher performance topologies have bus utilizations that are better balanced across the various levels. For example, the level 1 bus is the bottleneck in the N1×4×4 topology, the level 3 (root) bus is the bottleneck in the N1×4×8 topology, and bus traffic is more evenly distributed among the buses at different levels in the more optimal N1×8×4 topology.

The optimal configuration for a given number of processors is the topology with the minimum number of levels such that the speedup in Figure 3 is still in the unsaturated region. Conversely, for a fixed number of levels, the optimal topology is one which has the largest number of processors at the saturation point. This is also the topology that distributes the traffic most evenly across all levels in the hierarchy. In general, it appears that optimal configurations are those which look "pyramidal", with the branching factor being high at level 1 and decreasing at each succeeding level. We note that for the parameter values used in this experiment, it appears that in the 40 MHz system the optimal 5-level configuration can support well over a thousand processors. In the 20 MHz system, the four and five-level topologies can support well over 500, and perhaps as many as 1000 processors.

### 4.3. Increasing Processor Speed

We next address how the performance of TREEBus changes with increases in processor speed. The increases in total computing power achieved by faster processors can then be traded-off against the increase in cost.

In these experiments, we vary the number of the processors and the processor speed, considering only the optimal topology for each system size, and holding all other parameters fixed at the values used in Figure 3a. Figure 4 gives the results. The performance measure plotted on the y axis is the *effective MIPS rate*, which is defined in section 3.6.

As long as the bus subsystem is not saturated, increasing the MIPS rate of the processors will increase the total effective MIPS delivered by the multiprocessor. In the 512-processor system, replacing 2-MIPS processors with 6-MIPS processors increases the total effective processing power from 895 MIPS to about 1730 MIPS. At this point, the system is close to saturation and further increasing the speed of the individual processors to 8-MIPS increases the total effective MIPS only marginally. Further investigation of 5-level topologies is needed to evaluate

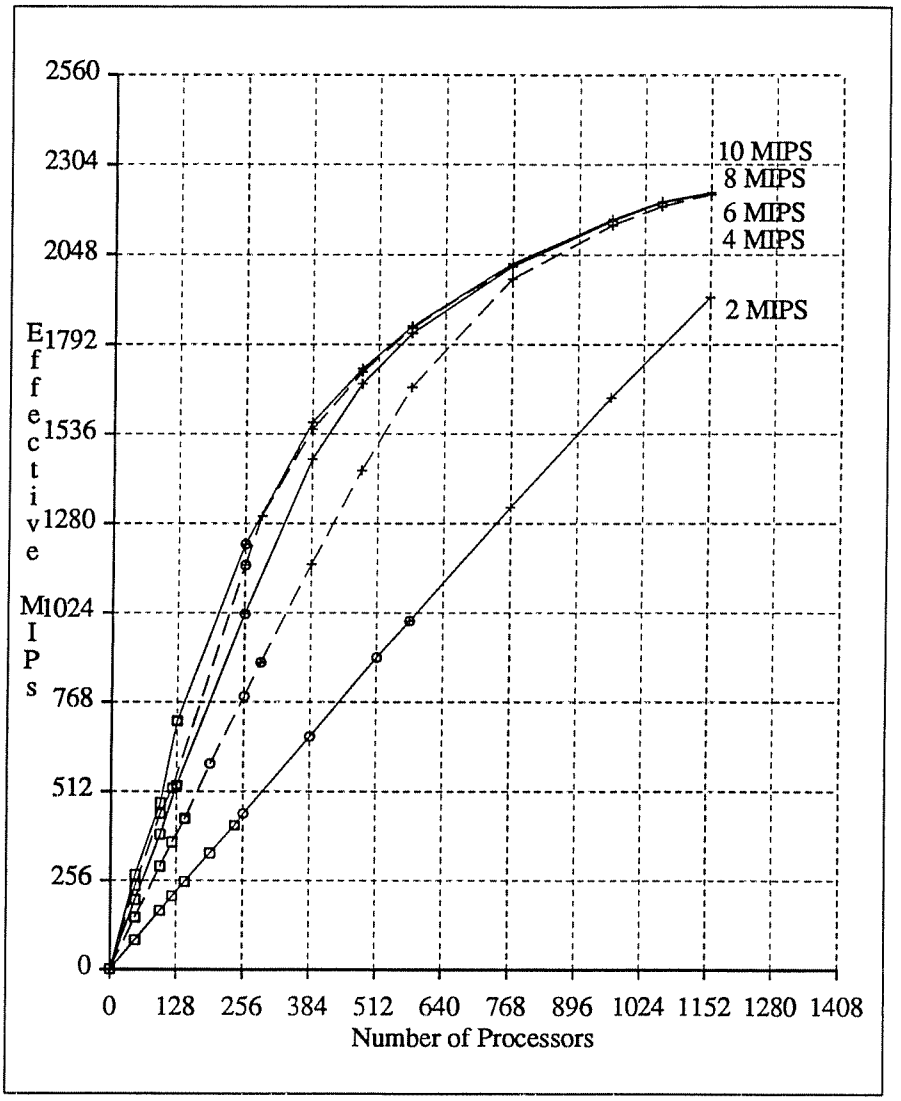


Figure 4: The Effect of Increasing Processor Speed

20 MHz buses, $f_{RI}$=0.5

the benefit of 6-MIPS processors for systems larger than 512 processors, and the benefit of processor speeds higher than 6 MIPS for systems larger than 256 processors.

Alternatively, instead of increasing hardware speeds, it may be possible to design the software in a manner that reduces the amount of globally shared data. This effect is studied in the next subsection.

### 4.4. The Importance of Global vs Local Sharing

In the above experiments, we have considered all shared blocks to be shared globally among all processors. Not only is the fraction of references to shared blocks important, but we also need to study the locality properties of the sharing of data.

Taking a 4-MIPS, 16×8×4 system (512 processors), with 22% of data references to shared blocks, we varied the sharing from 100% global (i.e. all shared blocks are uniformly accessed by all processors), to 100% local (i.e. shared blocks are only shared among processors connected to the *same* level 1 bus. This was done for "traditional" miss rates in the level 1 caches of 3%, 5% and 7%. All other parameter values are assigned as in Figure 3a. The results are presented in Figure 5.

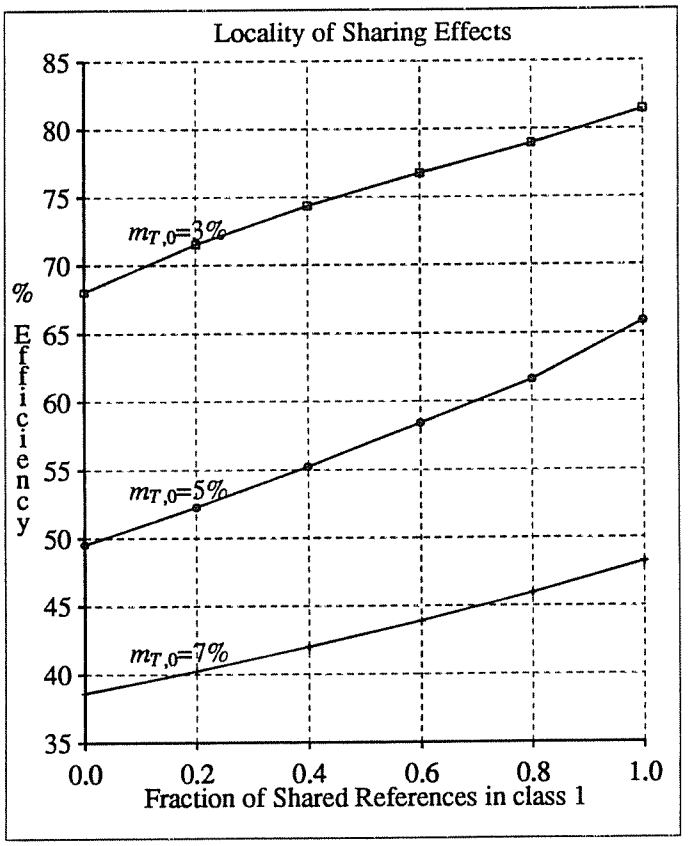


Figure 5: The Effect of Locality of Sharing

16×8×4 = 512 4-MIPS processors, 20 MHz buses

Figure 5 shows that performance improves with increased locality of sharing, and that the gains are more pronounced when the buses are highly utilized, but not yet saturated. For the case where $m_{T,0}$ = 5%, the efficiency of the 512 processor system increases from about 50% to about 66% as the sharing changes from being fully global to fully local. The results suggest that some of the lower efficiency caused by a higher traditional miss ratio can be offset if the degree of sharing is made as far local as possible. The other inference that can be drawn from this figure, at least for the input parameter values chosen, is that the traditional miss ratio (which is a function of the cache size) is a more dominant determinant of system performance than the locality of sharing.

### 4.5. The Effect of Interleaved Reads

Our final set of experiments investigates the implications of the value of $f_{RI}$ used in the earlier experiments. This parameter denotes the fraction of processors that share a data block which actually read the block between successive writes to the block (on the average). $f_{RI}$ was set to 0.5 in our previous experiments. The first read access by a processor after a block is written by another processor will cause the block to appear in all caches on the path from the supplier's bus to itself. Future read accesses to the block by other processors in the same subtree are satisfied within the subtree itself, thereby amortizing the cost of the initial read. Thus, increasing the value of $f_{RI}$ increases the probability that a request is satisfied at the lower levels of the TREEBus, thereby decreasing the latency of requests and reducing the traffic on the higher level buses, both of which contribute to a higher system performance.

Figure 6 shows the effect of varying $f_{RI}$ in a N1×12×2×2 system with 4 MIPS processors, 20 MHz buses, and 22% of data references to globally shared data (i.e. data that is shared by all processors).

High values of $f_{RI}$ may be appropriate for applications that incrementally build up shared data (i.e. shared knowledge or state information) that is then read by a large number of processors in the system. On the other hand, if the application involves mostly pairwise sharing of modified data, the performance degrades substantially with increasing system size. This is shown in Figure 6 in the case with $f_{RI}$ = 0.01. Our previous experiments have not considered low values of the $f_{RI}$ parameter. In particular, we note that if $f_{RI}$ is low, locality of sharing may have a more significant effect on performance than shown by the results in section 4.4. This is one of issues we are currently investigating.

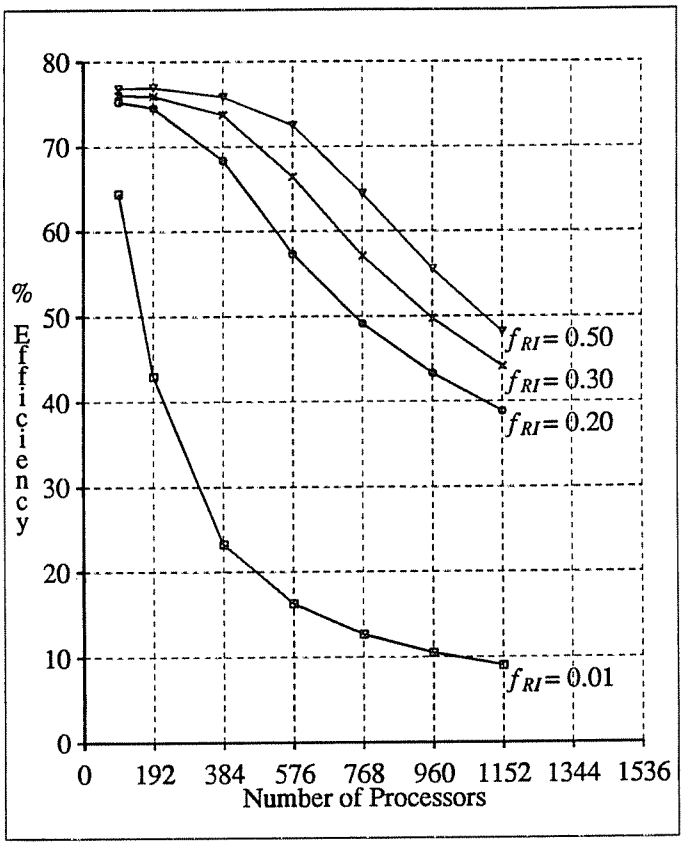


Figure 6: The Effect of Global Reads Between Invalidates

4 MIPS processors, 20 MHz buses, N1×12×2×2 topology

## 5. Conclusions and Future Work

In this paper, we specified a cache-coherence protocol for a tree-structured shared-memory multiprocessor. The protocol distinguishes between the information kept in the bus-side and the processor-side directories of each cache, which greatly simplifies the protocol specification.

Next, we developed a sophisticated performance model of the system. The mean-value queueing analysis includes bus latency, shared memory latency, and bus interference, as the primary sources of system performance degradation. The model also has a small set of relatively independent input parameters that are easily related to application-level program characteristics.

Using our analytical model, we analyzed several performance aspects of the hierarchical architecture, for systems as large as 2048 processors. Our analysis shows that a tree-structured cache/bus architecture can be

configured for very high performance, as long as the total bandwidth in the bus subsystem can support the total load placed on it by the processors. Our results can be used to select an optimal topology for a given system size. For 20 MHz buses, our results also show that optimal topologies of more than 256 processors reach saturation with 6 to 8 MIPS processors. Finally, our results show that system performance for shared data is highly dependent on the fraction of processors that read a block between writes to the block by a given processor. If this fraction is high, the locality of sharing within a cluster appears to be less important than the "traditional" miss rate for non-shared data.

Our current work includes further development of the concept of non-identical dual directories, which can also be profitably applied to the specification of protocols for non-hierarchical systems. Our continuing work also includes further investigation of the performance of hierarchical cache-coherent multiprocessors, using the performance model developed in this paper. In particular, we are examining the effects of shared data locality and access patterns, and the interplay between these two properties, in more detail [Vern88]. Future plans include improving the model to include cache and memory interference, extending the model to study the benefits of adding an *exclusive* state (requiring read-notify operations) to the cache coherence protocol, and investigation of real workload parameter values.

**References**


[Arch84]
J. Archibald and J. -L. Baer, "An Economical Solution to the Cache Coherence Problem," *Proc. 11th Annual Symposium on Computer Architecture*, pp. 355-362, June 1984.

[Arch86]
J. Archibald and J. -L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, vol. 4, pp. 273-298, November 1986.

[Baer87]
J. -L. Baer and W-H. Wang, "Architectural Choices For Multi-Level Cache Hierarchies," Technical Report 87-01-04, Department of Computer Sciences, University of Washington, Seattle, WA 98195, January, 1987.

[Bita86]
P. Bitar and A. M. Despain, "Multiprocessor Cache Synchronization: Issues, Innovations, Evolution," *Proc. 13th Annual Symposium on Computer Architecture*, pp. 424-433, June 1986.

[Dare87]
F. Darema-Rogers, G. F. Pfister, and K. So, "Memory Access Patterns of Parallel Scientific Programs," *Proc. SIGMETRICS International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pp. 46-58, May, 1987.

[Good83]
J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Annual Symposium on Computer Architecture*, pp. 124-131, June 1983.

[Good88]
J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," in *The 15th Annual Int'l. Symp. on Computer Architecture*, Honolulu, Hawaii, May 30 - June 2, 1988, to appear..

[Hwan84]
K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

[Katz85]
R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a Cache Consistency Protocol," *Proc. 12th Annual Symposium on Computer Architecture*, pp. 276-283, June 1985.

[Lazo84]
E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance, Computer System Analysis Using Queueing Network Models*. Englewood Cliffs, New Jersey: Prentice Hall, 1984.

[Lazo88]
E. D. Lazowska, M. K. Vernon, and J. Zahorjan, "An Accurate and Efficient Analysis of Single-Bus Cache Consistency Protocols," in *The 15th Annual Int'l. Symp. on Computer Architecture*, Honolulu, Hawaii, May 30 - June 2, 1988, to appear..

[Papa84]
M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proc. 11th Annual Symposium on Computer Architecture*, pp. 348-354, June 1984.

[Rudo84]
L. Rudolph and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proc. 11th Annual Symposium on Computer Architecture*, pp. 340-347, June 1984.

[Swea86]
P. Sweazy and A. J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus," *Proc. 13th Annual Symposium on Computer Architecture*, pp. 414-423, June 1986.

[Vern86]
M. K. Vernon and M. A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets," *Proc. SIGMETRICS International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pp. 9-17, May 1986.

[Vern88]
M. K. Vernon, R. Jog, and G. S. Sohi, "Performance Analysis of Hierarchical Cache-Coherent Multiprocessors," 1988, (in preparation).

[Wils85]
A. W. Wilson, "Organization and Statistical Simulation of Hierarchical Multiprocessors," Ph. D. Thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, PA, 1985.

[Wils87]
A. W. Wilson, "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," in *Proc. 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 244-252, June, 1987.

## Appendix I: Request Propagation Probabilities

| | $k=0$ | $k=1..L-i$ |
|---|---|---|
| $\alpha_{k,i}$ | $m_{M,0}\prod_{j=1}^{i-1}\delta_{0,j}$ | $m_{M,0}\left(\prod_{j=1}^{i+k-1}\delta_{0,j}\right)\frac{\mu_{0,i+k}}{n_{i+k}-1}\prod_{j=i+1}^{i+k-1}\frac{m_{R,j}}{n_j}$ |
| $\delta_{k,i}$ | $(1-p_{C,i})m_{M,i}$ | 0 |
| $\mu_{k,i}$ | $p_{C,i}m_{PCS,i}$ | $m_{R,i}$ |

| | $k=0..(L-i)$ | |
|---|---|---|
| $\beta_{r,k,i}$ | $\delta_{k,i}+\mu_{k,i}$ | $r\in\{addr,data\}$ |
| | $1-(\delta_{k,i}+\mu_{k,i})$ | $r=both$ |

## Appendix II: Example Calculations of Low Level Model Parameters from the High Level Model Inputs

Shared blocks may miss due to invalidations caused by interleaved accesses by other processors. In addition, both private and shared blocks that do not miss for the above reason, may miss for "*traditional*" reasons such as startup effects and block purging.

$$m_{M,0} = f_{P,0} m_{T,0} + \sum_{\kappa=1}^{L} (f^{\kappa}{}_{S,0} f_{iW}{}^{\kappa})$$

A block that reaches the memory at level $i > 0$ will have missed in all the level $i$ caches. If this block is shared then it will be supplied by a peer cache at some higher level, and thus will miss in this memory. It will do so because it has been invalidated by processors outside the subtree rooted at the level $i$ bus. Hence the miss ratio at level $i$ is:

$$m_{M,i} = f_{P,i}\, m_{T,i} + \sum_{\kappa=i+1}^{L} f^{\kappa}{}_{S,i} + m_{T,i}\, f_{S,i}{}^{T}$$

Note that $f_{S,i}{}^{T}$ is the fraction of references that reach the level $i$ memory that are shared but have propagated because of the traditional miss ratios, and $f^{\kappa}{}_{S,i}$ is the fraction that have missed due to invalidations by other processors.

We now find the probability that an invalidate request will continue to propagate towards the root level when the block is supplied by a cache level $i$. This will be given by 1 - the probability that the block is in state ONS in the level $i$ memory given that the block is supplied by a peer cache at level $i$. The cache supply occurs only for shared blocks. The invalidate must have occurred by a processor in the subtree rooted at level $i$, excepting the subtree where the current request is coming from. (Otherwise, the memory would not be an "owner" of the block.) There are $N^{i} - N^{i-1}$ such processors, out of a total of $N^{\kappa} - N^{i-1}$ processors that can potentially have done the invalidate. This has to be adjusted for the fact that, after the invalidate, this access must be the first out of all requests for the block that makes it to the level $i$ bus. There are $n_i - 1$ such possible requests, adjusted by $f_{RI}$, plus the current request. Hence we have ;

$$p_{\overline{ONS}\,|\,cache\ supply,i} = 1 - \frac{\sum_{\kappa=i}^{L} \left[ \frac{N_{0,i-1}}{N_{0,i-1} + f_{RI}(N_{0,\kappa} - 2N_{0,i-1})} \frac{N_{0,i} - N_{0,i-1}}{N_{0,\kappa} - N_{0,i-1}} p^{\kappa}{}_{i\,|\,S} \right]}{p_{c\,|\,S,i}} \qquad i = 1..L$$