An Analytical Model of High Performance Superscalar-Based Multiprocessors

David H. Albonesi and Israel Koren
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003, USA
albonesi,koren@ecs.umass.edu

Abstract

Several shared memory multiprocessor models using approximate Mean Value Analysis (MVA) have been developed and used to evaluate a number of system architectures. Since this time, the complexity of multiprocessor systems has increased as superscalar processors and latency reduction techniques are employed in these systems. We present an MVA multiprocessor performance model which incorporates these new features and in addition, increases the level of modeling detail to improve flexibility and accuracy. We describe in detail extensions present in our model that allow us to analyze the impact of these new features. We then use the model to demonstrate some of the tradeoffs involved in designing modern multiprocessors, including the impact of highly superscalar architectures on the scalability of multiprocessor systems.

1 Introduction

An analytical modeling technique that has been frequently used to evaluate shared memory multiprocessors is approximate Mean Value Analysis (MVA)[12]. In MVA, a set of equations that represent the mean response times and mean waiting times of various performance elements are derived using the mean values of various system parameters as model inputs. For example, a simple multiprocessor MVA model may include the mean response time of a cache miss as a response time equation, the mean bus waiting time as a waiting time equation, and the mean time between cache misses as a model input. Once constructed, these equations have circular dependencies and must be solved iteratively. The power of MVA modeling lies in its computation efficiency. Convergence is usually achieved within a second, independent of the number of processors and memories in the model. Many models of multiprocessor systems have been created using this technique and have been used to evaluate the performance of both research prototype[9, 13] and commercial[10] multiprocessors. The results obtained from approximate MVA models have been shown[5] to correlate well with those obtained from trace-driven simulation models.

Since these models have been constructed, superscalar microprocessor-based multiprocessor systems using latency reduction techniques such as nonblocking caches and data wraparound on cache misses have been introduced. These new aspects of modern multiprocessors impact both the system latencies and waiting times at each design level, and as a result, require extensions to the basic techniques used in MVA modeling of shared memory multiprocessors.

In this paper, we extend existing models of these systems to include the above aspects of high performance, superscalar-based multiprocessors. We describe the modifications to the latency and waiting time calculations, and in addition, propose modifications to the basic assumptions that have been used in past models. We also discuss additional features that we have incorporated into our model in order to increase the flexibility and accuracy of the model. We show that the features incorporated and the modeling assumptions used have a great impact on the performance results obtained.

The rest of this paper is organized as follows. We first describe previous MVA models, and then discuss extensions to these models to take into account the effects of superscalar processors and latency reduction techniques. Next, we use the model to examine the performance impact of modifying system parameters included in the model but not in previous models. Then we describe the underlying assumptions and limitations of the model, and conclude and discuss future extensions to our work.

2 Previous Work

Several MVA models of bus-based, shared memory multiprocessors have been previously developed to study architectural tradeoffs and coherency protocols. The model developed by Vernon *et al*[20] was one of the first MVA multiprocessor models to appear in the literature. The results of the model were compared with those obtained using Generalized Timed Petri Nets and good correlation (within 3%) was shown. In [5], the results of an MVA model were compared with those obtained through trace-driven simulation. A large number of configurations were modeled and good correlation (within 3% in most cases for processor throughput) was achieved. In [6], the same model was used to examine tradeoffs involving block size, associativity, and other parameters. A prototype commercial multiprocessor was modeled and analyzed in [10], and several design tradeoffs using the model were explored. Inputs to the model were derived from hardware measurements and simulation. In [1], an MVA multiprocessor model was used along with cache and bus cycle time models to analyze performance/area tradeoffs

in the design of single chip multiprocessor systems.

Models of larger scale multiprocessors have been developed as well. An MVA model[14] of the Wisconsin Multicube[9] was developed in order to examine design tradeoffs and issues such as scalability of the architecture. Another similar model[21] was used to analyze the performance of a multiprocessor system using a hierarchy of caches and buses. An open queueing network model[19] loosely based on the DASH[13] multiprocessor was developed, and used to evaluate the effect of data locality, cluster bandwidth, and other parameters on system performance.

All of these models share several common characteristics which limit their applicability to the analysis of modern multiprocessors. All are based around scalar processors which generate misses serially and whose caches are blocking. In addition, the models are all fairly high level in terms of design detail (although [10] goes into more detail than the rest). For example, cache overheads are typically lumped into a single average number instead of being calculated using detailed knowledge of the underlying hardware operations of the hierarchy. Writebacks of dirty blocks are also assumed to cause the processor to stall for the entire duration of the operation as support for buffering is not included.

There are several significant differences between these models as well. Different levels of flexibility are incorporated. For example, since [20] was used to evaluate coherency protocols, several protocols are supported, while in others, a single protocol is assumed. Some models attempt to model cache interference while others ignore this effect. Jog[10] goes into greater detail in modeling the system hardware than the other models, by including processor overheads for various requests. The DASH model[19] accounts for the fact that the full network latency may not need to be paid on every transaction. We combine the merits of these previous models and in addition, expand them significantly to incorporate new features inherent in modern multiprocessors.

3 Model Description

3.1 General Features

The general philosophy behind the development of the model was to build in as much flexibility as possible in order to be able to analyze a wide design space and model in detail the hardware at various levels (processor, caches, etc) of the system.

Either scalar or superscalar-based multiprocessor systems can be modeled. The degree of superscalar execution and the amount of load/store activity generated by the processor and workload can be varied[1]. Unlike previous models which simply incorporate cache hit rates into their models, we model the cache hierarchy explicitly. Two general cache hierarchy configurations are supported: a single level of writeback, allocate-on-write cache (as used in most current HP-PA implementations, e.g., [4]), or a multilevel cache hierarchy. The single cache can be optionally backed by a displacement buffer for hiding the latency of dirty block writebacks. The multilevel hierarchy consists of either unified or split (Icache and Dcache) L1 caches and a unified, writeback, allocate on write L2 cache. The L2 cache can be optionally backed by a displacement buffer.

The write policy for the L1 data cache for the cache hierarchy is writethrough, non-allocate on write (as for example in the Alpha 21064[11]), and the cache is optionally backed by a write buffer. Support for writeback L1 cache write policies (as in the Pentium[3]) is currently being implemented. The overheads due to maintaining the write and victim buffers is variable, as is the cache coherence protocol. (Write update, write invalidate, or hybrid[18] protocols are supported.) Modeling a superscalar processor and a cache hierarchy explicitly affords significant advantages over previous models: we are able to rapidly explore the design space of a much wider range of machine parameters, and quickly gain insight into the interactions of processor and cache hierarchy architecture with multiprocessor performance.

The level of design detail provided in the model is much greater than that in previous models. This allows great flexibility in the studying of candidate architectures, and the precise modeling of hardware overheads at each design level. Every level contains parameters for calculating latencies and waiting times for *each individual operation type* (instruction read, data read, data invalidate, data writeback, etc). The motivation behind this is three-fold. First of all, resource overheads and latencies vary according to the operation being performed. For example, if the Icache and Dcache block sizes are different, then the duration of main memory instruction and data read operations will differ as well. Secondly, a higher degree of design detail can increase the accuracy of the model. Lastly, for nonblocking caches, waiting times and latencies need to be separately calculated for each operation type. The reasons behind this are further discussed in a later section.

Remote cache interference is modeled in detail as well. The workload parameters include entries that reflect the probability that an operation interferes with another cache. For example, the parameter $P_{L2remote\_du\_hit}$ is the probability that a cache update operation hits in the remote cache. Separate overhead and latency parameters exist for each of these remote operation types as for example, the overhead of a write invalidate may differ from that for a write update. The model also supports various cache tag organizations for bus snooping. If duplicate tags are assumed, then only those bus transactions that "hit" in these tags interfere with the cache[2]. If no duplicate tags are present, then the main tags must be checked for all transactions. Different overheads exist for transactions that hit, and those that miss.

We use the same approach to modeling cache interference as bus and memory contention. We treat each cache as a resource that is shared by the processor and the bus. Queueing, utilization, and waiting times are calculated in the same way as for the bus and memory, except that in the case of the caches, the waiting times for processor transactions will be less than those for bus transactions. This is because the processor more heavily utilizes the cache, and thus the probability of a busy cache due to processor operations is much higher than that due to bus transactions.

3.2 Modeling Superscalar Processor-Based Systems

The main difference between a scalar processor and a superscalar one (from a modeling standpoint), is that a su-

[1] Our intent is not to provide a detailed superscalar processor analytical model, but rather to represent the increased L1 cache utilization and system traffic of a superscalar processor. A detailed superscalar uniprocessor model can be found in [7].

[2] We also model the overhead required to maintain consistency between the main and duplicate tags.

perscalar processor consumes more than one instruction simultaneously. If the processor contains multiple load/store units, then it may also perform multiple loads or stores (or a combination of the two) simultaneously. Thus, the instruction and data level bandwidth requirements of the processor grow as the degree of superscalar execution increases. In order to compare various superscalar and scalar processors, we use an *instruction fetch cycle* as the basic cycle of operation executed by the processor, and use the measure *cycles per instruction fetch* (*CPIF*) to describe the length of this basic cycle. We use the input parameters N_{Ifetch} to describe the average number of instructions that are fetched by the processor each *CPIF* interval, and CPI_{proc} as the cycles per instruction rating of the processor with no memory system overhead. Thus, the value of *CPIF* is determined by multiplying CPI_{proc} and N_{Ifetch}, calculating reference rates of instruction and data cache operations for this interval, and adding in any additional factors due to memory system overheads induced by these references. The overall *CPI* value including memory overhead can then be computed by dividing *CPIF* by N_{Ifetch}.

A superscalar processor containing multiple load/store units can perform more than one load or store each *CPIF*. This requires that the Dcache be multiported, banked[17], or restricted to performing multiple Dcache operations in the same cache block. The input parameter $N_{Daccess}$ is the average number of Dcache accesses each *CPIF* and is a function of both the workload characteristics (average number of loads and stores in the program), and the architecture's ability to fill the load/store units. The maximum value for this parameter is one, which is the case where there is at least one load or store being accessed each *CPIF*, and represents an aggressively designed superscalar processor and/or a workload with a high degree of loads and stores. This parameter is used in calculating the portion of cache utilization due to load and store operations.

3.3 Modeling Latency Reduction Effects

Several techniques exist for hiding the latency inherent in high performance multiprocessors, including:

- Providing *concurrency* in the hardware, including returning the desired instruction or data in a cache block first when a miss occurs, and bypassing the cache with the desired instruction or data while loading the cache block in parallel;
- Non-blocking loads;
- Non-blocking stores (write buffers);
- Victim buffers.

We now discuss each of these in turn in terms of how latency calculations are performed. We then address how waiting time calculations need to be modified to account for these effects.

3.3.1 Hardware Concurrency

Previously developed multiprocessor models use the *transaction time* of a resource to calculate both its utilization and its latency. For example, if the bus data transaction time is 3 cycles, then a data transfer across the bus consumes 3 cycles of that resource and requires 3 cycles of latency. The problem with this approach is that it does not correctly represent the *hardware concurrency* of modern systems, in which operations at different design levels are overlapped with those in others. An example of this occurs when, on a cache miss, the desired word (the actual word within the block which caused the cache miss) is transferred at the head of the block, and once received, the processor can resume execution while the rest of the cache block is written to cache. Thus, the bus which we just described when operating in this manner may still have a transaction time of 3 cycles, but a *stall time* of only 1 cycle. The stall time is that portion of the transaction time which contributes to the stalling of the processor. In our model, we calculate resource utilization and waiting time in the usual manner using the transaction time parameters, but we decouple these calculations from the stall time calculations. The latter are used in the calculation of *CPIF*. Thus, for each transaction time parameter in the model there is a corresponding stall time parameter. Whether or not these values are equal depends on the capabilities of the resource. Furthermore, separate transaction and stall time parameters exist for each operation type. For example, while Dcache misses may use desired word first with bypass, Icache misses may require the entire block to return before the desired word is loaded into the pipeline.

Separate stall and transaction times are assigned at every level of the model, from the L1 cache to the main memory system, and for each operation type. This affords great flexibility in the modeling detail that can be described. For example, cache controller overheads can be precisely described in this way. The transaction and stall times of a cache read operation may be 4 cycles and 3 cycles, respectively, a cache write operation may be 3 cycles and 2 cycles, respectively, while a remote cache invalidate may require 4 cycles and 6 cycles (to propagate the acknowledgement), respectively. The differences can be more pronounced for other operations, such as main memory writebacks. When a writeback occurs, the cache controller may only need confirmation that the main memory can accept the operation (it has room in its queue), rather than needing to wait for the actual memory write to occur. Thus, the transaction and stall times for a memory write operation can differ dramatically. This, we believe, more accurately represents the machine hardware than using the transaction time as the stall time, and as we show in Section 4, can have a great impact on the performance results obtained. The model contains a total of 76 transaction and 76 stall time parameters to allow for this level of modeling flexibility.

3.3.2 Nonblocking Loads

A data cache that uses nonblocking loads has the ability to handle more than one load miss simultaneously. The amount of latency that can be eliminated due to this technique depends on two major factors[8]:

- The compiler's ability to schedule non-dependent instructions during the cache miss; this partially depends on the cache miss latency;
- The hardware implementation.

The first factor represents the compiler's aptitude at overlapping computation with the load miss service time. This depends on the compiler technology, workload characteristics, degree of superscalar execution, and the duration of the service time. (Longer service times require more nondependent instructions to be found to hide the latency.) The

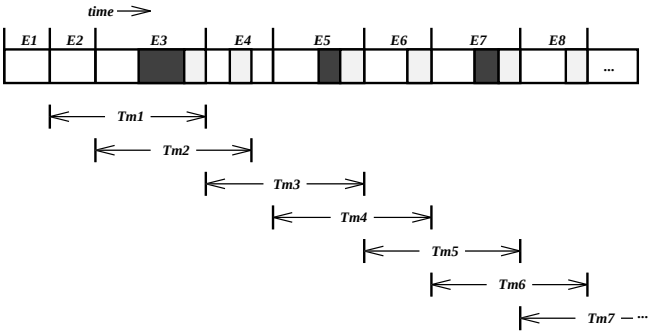

Figure 1: Effective Stall Time - Example 1

second factor represents the structural limits of the hardware in terms of how many misses it is equipped to handle simultaneously, and whether these misses can be to the same cache block, in which case they need not be issued but rather are *merged* with an already outstanding miss.

Three input parameters are introduced to account for these factors. The parameter $percent_{olap}$ is the average percentage of the load miss service time that is overlapped with computation. The second, N_{before_block} is the number of load *fetches* that can be issued before the next one blocks. While a *miss* refers to the absence of a datum that was attempted to be read from the cache, a *fetch* refers to the reading of a cache block in response to one or more misses. With a nonblocking cache with appropriate hardware support, one fetch can satisfy several misses to the same block[8]. For a blocking Dcache, N_{before_block} is zero. The probability that a load miss can be merged with another load miss is described by P_{load_merge}. A high degree of merges may reduce the average latency of a Dcache miss as well as the rate of fetches introduced into the system.

With these definitions in hand, we now illustrate how to calculate the *effective stall time* or the amount of stall cycles due to each load miss when the above effects are taken into account. Consider the case shown in Figure 1 in which $N_{before\_block} = 2$ and $percent_{olap} = 6/7$. Each E is the time between successive load misses, and T_m is the miss service time. The white areas represent computation time, the lightly shaded areas the time that computation is *not* overlapped with the miss service time, or $(1 - percent_{olap}) \cdot T_m$, and the dark shaded areas additional processor stalling due to architectural limitations. As Figure 1 illustrates, once a steady-state condition is reached, periods of processor stalling occur at regular intervals. Furthermore, we can calculate the average amount of stall time these periods contribute to each load miss as follows:

$stall\_period = T_m / N_{before\_block} - E_{avg\_ss}$

where E_{avg_ss} is the average time between successive load misses once steady state is reached. Thus, the total stall time for each load miss is

$T_{stall} = stall\_period + (1 - percent_{olap}) \cdot T_m$

Figure 2 shows another example where $N_{before\_block} = 3$ and $percent_{olap} = 14/15$. Here, we see that

$T_m = N_{before\_block} \cdot E_{avg\_ss}$

and so the stall period is zero as shown. Thus, the total stall time is simply

$T_{stall} = (1 - percent_{olap}) \cdot T_m$

Of course, we could have the situation in which $T_m < N_{before\_block} \cdot E_{avg\_ss}$. Here the total stall time is the same as that calculated from Figure 2. Thus, a lower bound on

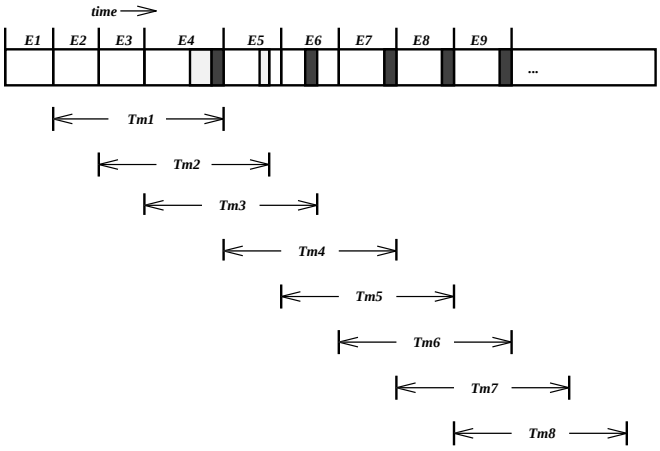

Figure 2: Effective Stall Time - Example 2

the total stall time for a nonblocking Dcache is

$T_{stall} \geq (1 - percent_{olap}) \cdot T_m$

Note that E_{avg_ss} is easily calculated as

$E_{avg\_ss} = CPIF / P_{load\_miss}$

where P_{load_miss} is the load miss reference rate in an instruction fetch interval. Clearly, the value of *CPIF* will depend on the total stall time, which in turn depends on the value of *CPIF*. Thus, we have the circular dependencies which are the trademark of MVA and must resort to iterative solution techniques.

3.3.3 Nonblocking Stores (Write Buffers)

In a multilevel cache hierarchy, consisting of a writethrough L1 data cache and a writeback L2 cache, it is common for a write buffer to be placed between the L1 and L2 caches so that writes only cause stalls when the buffer is full or when a Memory Barrier instruction[16] requires the buffer to be flushed. When the former situation occurs, the processor is stalled until enough writes are retired from the buffer to allow the new entry to be made. Several input parameters are used to describe the write buffer. P_{full_wbuf} is the probability that the buffer is found full when a write occurs; N_{wbuf} is the number of entries in the write buffer when full, and N_{wait_wbuf} is the number of entries that must be fully retired (which may require remote invalidations to propagate out from the L2 cache for example) before the new write can be loaded. Thus, the stall time for a write is:

$T_{stall\_write} = P_{full\_wbuf} \cdot (N_{wbuf} \cdot (T_{wait\_L2\_EWB} + T_{stall\_L2\_EWB}) + N_{wait\_wbuf} \cdot T_{retire})$

where $T_{wait\_L2\_EWB}$ is the waiting time at the L2 cache for an empty write buffer entry operation, $T_{stall\_L2\_EWB}$ is the corresponding stall time parameter, and T_{retire} is the time to retire each write. The latter depends on input parameters such as the L2 cache data write miss rate, the probability of writing to shared data, the invalidate protocol, the probability of displacing a dirty block from cache, etc.

We note that the above describes stall time calculations. The transaction time calculations for writes are carried out using the usual MVA techniques (with a few exceptions as noted later).

3.3.4 Victim Buffers

A victim buffer (or displacement buffer) provides temporary storage for dirty blocks displaced from cache when a miss occurs. The motivation for victim buffers is similar to that for write buffers (prevent displaced blocks from slowing down cache misses). The corresponding parameters P_{vbuf_full}, N_{vbuf}, and N_{wait_vbuf} are used along with the stall and waiting times in the same way as described for write buffers.

3.4 Waiting Time Calculation Modifications

The models described in Section 2 are suitable for multiprocessor systems made up of scalar processors. These models assume that requests from a given processor occur serially; in other words, a request introduced into the system will not encounter any traffic due to other requests from the same processor. In a multiprocessor system using superscalar processors with decoupled instruction and data streams and nonblocking data caches, this assumption may not be appropriate. In this section, we describe modifications to waiting time equations in order to take these factors into account.

The terminology we adopt is the following. Caches can have local and remote requests. A request is local when it is proceeding down its originating processor's cache hierarchy, onto the bus, into the main memory system, and in the case of a cache miss, back to itself. A request that travels up another processor's cache (such as a cache invalidate or a read request for data found modified in that cache) is remote. Thus a local request can be impeded by remote requests *and* possibly other local requests.

We define three *request categories*: Icache request, Dcache load request, and Dcache store request. Each category consists of several different request types. For example, the request types in the Icache request category are instruction miss, writeback due to instruction miss, empty victim buffer due to instruction miss, and return of cache block due to instruction miss. Similar types exist for the Dcache load request category, while the Dcache store request category has additional types due to the many request types (invalidates for example) that may result from store operations.

The amount of local traffic observed by a local request varies depending on the request type. We note the following:

1. Each request category observes all the traffic due to other request categories. For example, an Icache request can be impeded by Dcache load and store requests since the instruction and data streams are assumed decoupled.

2. Icache request and Dcache store requests can be considered blocking and therefore observe no other local Icache or Dcache store requests, respectively. Icache requests are assumed to be handled serially, and the transaction time of Dcache store requests is generally shorter than the time between successive store requests (except in cases of workloads with high store instruction rates). Thus, previous store requests are retired before the next store request occurs.

3. Dcache load requests in a blocking cache observe no other local Dcache load requests. For a nonblocking cache, other Dcache load requests may be observed depending on the transaction time relative to the time between successive requests.

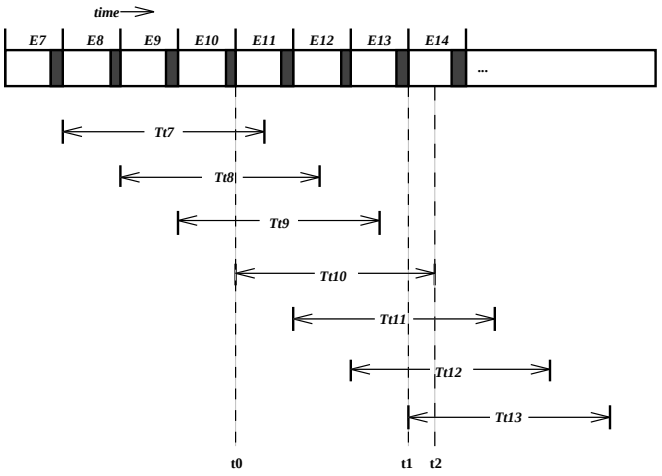

Figure 3: Local Dcache Load Miss Traffic

Thus, for Dcache load requests with a nonblocking cache, we must take into account other local load requests that are present in the system when the new request occurs, and calculate the fraction of the local Dcache load request traffic observed by the new load request.

To illustrate the calculation of this fraction, we study the example shown in Figure 3, which depicts the situation of Figure 2 after steady state conditions are reached. Here we have shown the transaction times (T_t) which are assumed to be 33% longer than the miss service times. (The latter are not shown for readability, but can be seen in Figure 2.) Or in other words, $T_t/E_{avg\_ss} = 3.5$. We evaluate the time period between t0 and t2, or the time during which transaction 10 takes place (T_{t10}). We observe that the average total Dcache load miss traffic between times t0 and t1 is 3.5, and between times t1 and t2 is 4.0. Thus, the average total load miss traffic between t0 and t2 is

$traffic_{total} = (3 \cdot 3.5 + 0.5 \cdot 4.0)/3.5 = 3.57$.

We also observe that transaction 10 sees an average of one less transaction than the average total traffic, or a value of 2.57. Thus, the fraction of the total Dcache load miss traffic observed by transaction 10 is ($traffic_{total}$-1)/$traffic_{total}$ = 2.57/3.57 = 0.72. Thus, transaction 10 is impeded by 72% of the local Dcache load miss traffic, as well as all of the local Icache miss and Dcache store traffic.

In general, we can calculate the fraction observed as ($traffic_{total}$-1)/$traffic_{total}$ by using the following formula to calculate $traffic_{total}$:

$traffic_{total} = (\lfloor (T_t/E_{avg\_ss}) \rfloor \cdot (T_t/E_{avg\_ss}) + ((T_t/E_{avg\_ss}) - \lfloor (T_t/E_{avg\_ss}) \rfloor) \cdot \lceil (T_t/E_{avg\_ss}) \rceil)/(T_t/E_{avg\_ss})$

This formula holds for the conditions $N_{before\_block} > 0$ and $T_t > E_{avg\_ss}$. The first ensures that the cache is nonblocking, and the second that the transaction time is longer than the average time between successive Dcache load misses. If one of these conditions does not hold, then a Dcache load miss sees no other Dcache load misses in the system.

4 Performance Results

Now that we have described the extensions made to the model, we examine the performance impact of various system parameters. All of the results reported in this section could not have been obtained using previous MVA mod-

els, as our model extends these models to take into account new parameters. We would like to emphasize that although the capability of the model has been expanded considerably compared to other models, this has very little impact on processing time. All of the runs performed for this paper converged within 50 iterations (in most cases in less than 10 iterations), and each run produced results in less than a second. Thus, a wide design space could be examined in a small fraction of the time required for simulation.

4.1 Fixed Parameters and Assumptions

In these illustrative examples, we fix a number of the architectural parameters to limit the design space. Although the model is capable of modeling single-level caches, multi-level caches are more common in modern multiprocessors, and therefore we limit our examples to this organization. We also model modest multiprocessor configurations (no more than eight processors), and unless otherwise noted, we limit the processor architecture to a two-way superscalar implementation with blocking caches.

The cache coherence protocol that we model is a combined write invalidate and write update protocol[2]. Here, a write to shared data in the L2 cache causes a bus write update operation to be performed. If the snooping L2 cache has a copy of the block, and its associated L1 cache does as well, then it accepts the update and invalidates the copy in the L1 cache. If it has a copy, but the L1 cache does not, then the copy in the L2 cache is invalidated. The idea here is to eliminate updates to blocks that may have migrated from one processor to another[18]. We model a system with a duplicate set of L2 cache tags for bus snooping. This prevents the main L2 cache tags from being interfered with except for cases in which a coherency operation must be performed.

Unless otherwise stated, each L1 cache is backed by a 4-entry write buffer and each L2 cache by a 2-entry victim buffer. With both of these, the probability is 10% that the buffer has to be emptied when accessed, and when the last entry is being removed from the buffer, the waiting entry can be loaded.

In order to isolate memory contention from the other aspects of the design we were interested in, we allocated twice as many memory modules as the number of processors in all runs. This ensured that memory interference effects would be of second order and not skew our results.

In the results that follow, the processing power curves include results representative of those obtained using earlier models. These models use scalar processors, blocking caches, no local traffic, no write or victim buffers, and use transaction times for stall times (no hardware concurrency). The curve labeled "Old model (no L2)", uses a single level of writeback cache, while the one labeled "Old model (L2)" uses the same multilevel hierarchy as the new results. Both use the default parameter values for the caches. The purpose of providing these curves is to quantify the increase in performance when calculated using the new parameters incorporated into the model, and to demonstrate the additional results the model is able to obtain.

Tables 2 through 7 at the end of this paper give default values for most of the parameters used in this section. Only the main memory stall and transaction time parameters are provided for brevity. Parameters which are not applicable to the organizations we have chosen are omitted as well.

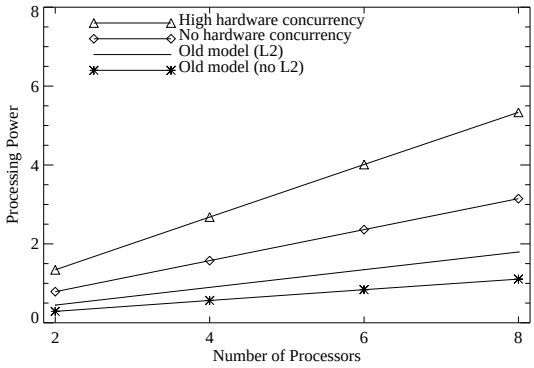

Figure 4: Effect of Hardware Concurrency Level on Processing Power

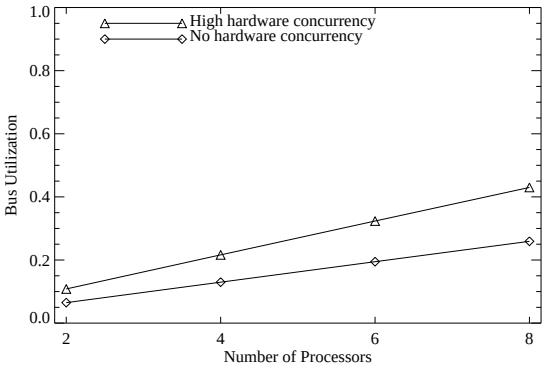

Figure 5: Effect of Hardware Concurrency Level on Bus Utilization

4.2 Hardware Concurrency

In this section, we investigate the impact of the level of hardware concurrency on the processing power (defined as n/CPI where n is the number of processors and CPI the total cycles per instruction count of each processor) and bus utilization of the machine. The latter is monitored in order to assess to what degree scalability projections are affected by concurrency variations.

We compare two configurations, one in which the stall time parameters are equal to the transaction time parameters, and another in which it is assumed that all caches use bypass, all resources transfer the desired word first in a block, and overlap occurs when transferring information from one resource to another. These organizations represent the two extremes of hardware concurrency, and thus provide an upper bound on their performance difference (for the assumptions and parameters we have described in the previous section). As shown in Figure 4, this upper bound is quite large, the upper curve yielding an increase of about 70% in processing power over the lower. This performance improvement however increases bus utilization (Figure 5) by about 65%. Thus, we see that both processing power and scalability projections can be greatly impacted by the design of the machine hardware. A large amount of concurrency in the hardware improves the processing power, but as a result, requires more aggressive interconnect resources in order to connect large numbers of processors.

We also see how results for the old models compare with the present one. The presence of an L2 cache as expected has

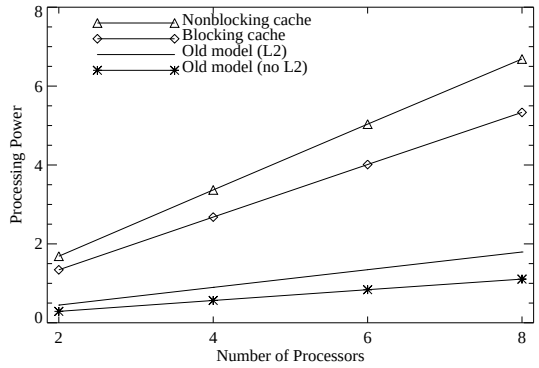

Figure 6: Processing Power of 2-Way Superscalar-Based Systems

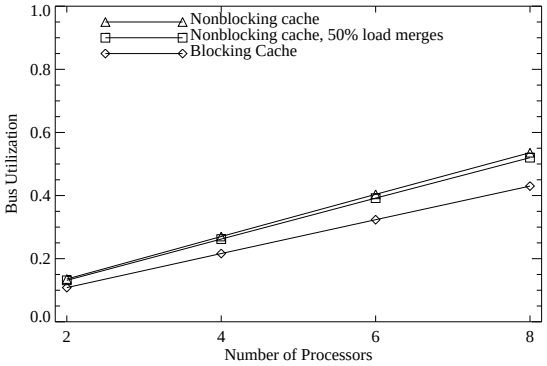

Figure 7: Effect of Merging on Reduction of Bus Utilization (2-Way Superscalar)

a large impact on the processing power obtained. The presence of a 2-way superscalar processor and hardware buffers has an even greater impact, almost doubling the performance of the old model with an L2 cache. Thus, we see that the new architectural parameters we have incorporated into the model have a tremendous impact on the performance results obtained.

4.3 Nonblocking Caches

The impact of incorporating processors with nonblocking caches on multiprocessor performance is now examined. The nonblocking cache organization operates under an enhanced *hit under miss* scheme in which the processor can continue until a second fetch is required or the merge capability is exhausted. The latter can occur due to limited hardware resources allocated for merges for example, or a workload in which load misses are spatially "scattered" and not clustered into recently referenced cache blocks. We initially compare two organizations: one with a blocking cache and another with a nonblocking cache with $percent_{olap} = 0.85$ and $P_{load\_merge} = 0.0$. As shown in Figure 6, a significant performance improvement is obtained with a nonblocking cache, but the cost is an increase in bus utilization (Figure 7). Intuitively, we would expect that a high degree of merges would reduce the increase in bus utilization, since the amount of Dcache load misses would be reduced. However, as seen in Figure 7, this is not the case. The explanation for this is that the presence of a nonblocking cache reduces the latency of each processor, and therefore increases it processing rate, and consequently, the rate at which *all* misses occur. This increased processing rate is the dominant effect in the increase in bus utilization, while the lesser effect (the increase in the number of Dcache load misses) is the effect improved by merging. Thus, in this example, the impact of merging on bus utilization is small, as is its impact on processing power (only about a 1% improvement). The latter is due to the fact that with a large L2 cache providing a high hit rate, almost all of the miss latency is hidden by allowing a single outstanding miss, and not much more improvement can be gained by merging.

Parameter	Value
$P_{L1\_ir\_miss}$	0.05
$P_{L1\_dl\_miss}$	0.20
$P_{L2\_ir\_miss}$	0.02
$P_{L2\_dl\_miss}$	0.10
$P_{L2\_ds\_miss}$	0.12

Table 1: Miss Rates for 4-Way Superscalar-Based System

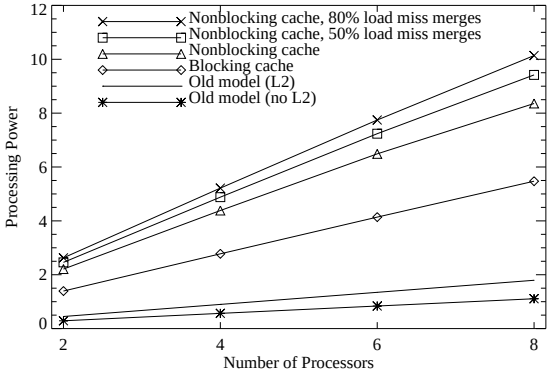

Figure 8: Processing Power of 4-Way Superscalar-Based Systems

For more aggressive superscalar designs which generate higher cache miss rates and whose performance is more dependent on latency reduction, merging may be more beneficial. For example, the recently announced Alpha 21164 microprocessor[15], a 4-way superscalar design, provides aggressive support for merging through its 6-entry Miss Address File. We examine the effect of nonblocking caches by comparing four 4-way superscalar-based multiprocessor configurations based around the following cache organizations: a blocking cache, a nonblocking cache with no merging, a nonblocking cache with 50% merging, and a nonblocking cache with 80% merging. In order to reflect the impact of a higher degree of superscalar execution on cache miss rates (due to higher bandwidth requirements that cause it to sweep through the cache at a higher rate), we use degraded cache miss rates as shown in Table 1. As Figure 8 shows, the performance of the 4-way superscalar-based multiprocessor improves greatly with the addition of a nonblocking cache and a high degree of merging. Whereas with a 2-way superscalar-based multiprocessor a 50% merging rate offered little additional performance benefit over a nonblocking cache without merging, the 4-way superscalar system shows a significant (about 20%) improvement in performance with 50% merges, and an additional smaller, but

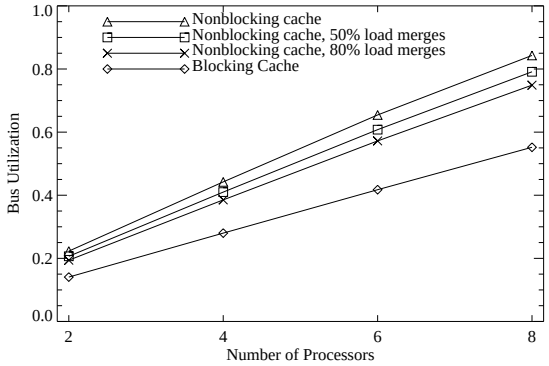

Figure 9: Bus Utilization of 4-Way Superscalar-Based Systems

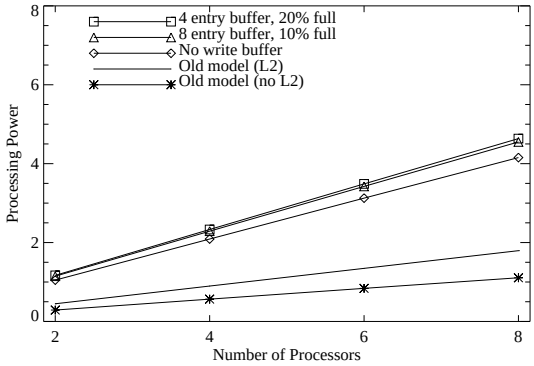

Figure 10: Processing Power of Various Write Buffer Configurations

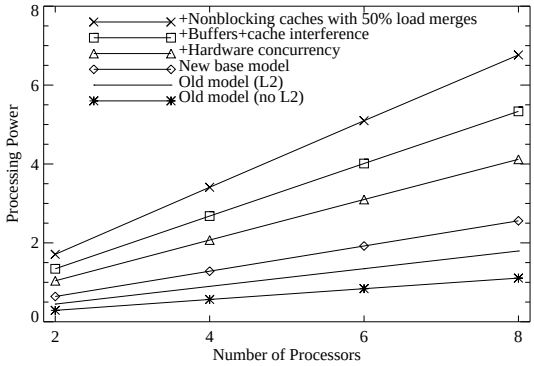

Figure 11: Impact of Combinations of Effects on Processing Power

still noteworthy (about 8%) improvement when the merging rate increases to 80%. Thus, the results we were able to obtain using the model appear to validate the decision to support aggressive merging on the Alpha 21164[3]. Bus utilization (Figure 9) is also reduced by merging (about 6-7% from no merging to 50% merging), but even so, the bus utilization increase from a blocking cache to any of the nonblocking configurations is dramatic, around 35-40%. This as mentioned previously, is largely due to the simple fact that the processor is executing at a faster rate, and generating misses at a higher rate as well. We note that the bus utilization values for the 4-way superscalar-based multiprocessor are much greater than those for the 2-way superscalar-based system (Figure 7). Thus, these examples illustrate how the degree of superscalar execution has a very large impact on the design of the memory system and interconnect in a multiprocessor system.

Note once again on these figures, the difference between results for the old models and the present one. The 4-way superscalar processor with a nonblocking cache and 80% merging has a processing power that is five times that for the old model with an L2 cache.

4.4 Write Buffers

The presence of a write buffer helps to hide the latency of writes, allowing read misses to proceed ahead of them. Choosing a size for the write buffer is a tradeoff between minimizing the percentage of time the buffer is full when a write occurs and the stall time in emptying the buffer when it is full. To illustrate this tradeoff, we compare the performance of three systems: one without a write buffer, one with a 4-entry buffer which is full on a write 20% of the time, and one with 8 entries that is full 10% of the time. We make several observations based on the results obtained in Figure 10. First of all, the performance improvement when employing a write buffer is smaller than that of other parameters we have studied so far, but still significant (about 11%). Secondly, we see that the 4-entry buffer performs slightly better than the 8-entry buffer due to the latter's higher penalty for emptying. We note however, that we assumed that the 8-entry buffer would be full half as much as the 4-entry buffer. This assumption is highly dependent on workload characteristics, processor architecture, and the L2 cache's emptying algorithm. Thirdly, the impact of a write buffer stays relatively constant as the number of processors is varied. This is due to the fact that in the model, while stall time calculations are varied with write buffer organization, transaction times, which are calculated using conventional MVA techniques, do not. Thus, in our model, the write buffer does not reduce traffic (the writes are simply delayed), only stall time. This is a limitation of MVA techniques as will be explained in Section 5.

Victim buffers can be studied in a similar manner, but the performance improvement obtained is smaller than that from write buffers due to the fact that they lie further down the cache hierarchy. Their impact would be greater for a single level of cache hierarchy, and the model allows this organization to be studied as well.

4.5 Combined Effects

Figure 11 shows how combinations of the previously studied effects can impact performance. In addition, we include the effects of cache interference. The curve labeled "New base model" represents a 2-way superscalar model with no hardware concurrency, no write or victim buffers, cache interference ignored, and blocking caches. Each successive curve progressively adds new features to the previous curve. We make two observations from this figure:

- Using large amounts of hardware concurrency and nonblocking caches have the greatest impact on performance;

[3]With an L3 cache (optional on the 21164), latencies would be reduced and the need for merging would perhaps be less.

- Effects which alone have a smaller performance effect (such as write and victim buffers) can add together to produce a much more significant impact.

5 Limitations of the Model

The model, although very flexible, is not without limitations, as we rely on several assumptions which we discuss in this section.

A fundamental assumption of the superscalar aspects of the model is that the cache data path widths scale as the degree of superscalar execution is increased. For instance, an n-way superscalar implementation will fetch half the amount of instructions from the Icache each *CPIF* as a 2n-way superscalar processor. This necessitates the increase in datapath width to meet the increased bandwidth. Thus, the impact of increased superscalar degree is primarily reflected in increased cache miss and buffer emptying rates, as the processor is able to sweep through these structures at a higher rate.

Because MVA modeling relies on average values to compute performance, certain effects are difficult to represent using this method. We assume that remote cache interference cannot be hidden through the existence of invalidate request queueing for example. (In systems employing a cache hierarchy, L1 cache invalidate requests can be queued and, in most cases, the emptying of the queue is delayed until the cache idles due to a miss.) Similarly, by calculating transaction and waiting times due to writes using conventional MVA techniques, we assume that the presence of a write buffer between the L1 and L2 caches only serves to reduce write latency and not to reduce L2 cache contention (through emptying the buffer during idle periods). A similar assumption holds for displacement buffers.

We have found in our modeling of nonblocking caches that when an L2 cache is present, the average analysis used in MVA modeling affects the amount of performance benefit received from allowing multiple outstanding misses. An L2 cache (with a reasonably low miss rate) reduces the average stall time of a Dcache load miss. This combined with a uniform distribution of these misses limits the benefits obtained from aggressive nonblocking schemes which allow many outstanding misses to be serviced simultaneously. This limitation arises because there is little overlap to hide, and allowing one outstanding cache miss serves to hide almost all of it. In real machines, misses may be "clustered" and therefore greater benefit received from allowing many misses to be outstanding. It is a subject of further investigation to more throughly investigate this aspect of the model.

6 Conclusions

An approximate Mean Value Analysis model which incorporates the features of modern multiprocessor systems has been developed. The model is extremely detailed and includes the effects of superscalar microprocessors and latency reduction techniques and has been used to study a wide variety of design tradeoffs. We have demonstrated how the model expands on earlier developed models, and how this effects the performance results obtained.

Using the model, we have examined the impact of using superscalar processors and nonblocking caches in multiprocessor systems. The model has been used to determine the impact of the degree of superscalar performance on the scalability of multiprocessor systems. We have demonstrated how contrary to intuition, a high degree of merging of outstanding misses in a nonblocking cache has little impact on bus utilization and explained the reasons for this result. We have also shown how the model can be used to study the way buffering, interference, and other effects can affect system performance and scalability.

Currently, we are expanding the model to include write-back L1 caches in a cache hierarchy, and other snooping tag structures. The incorporation of a third level (or an arbitrarily high level) of caching is another possibility for future expansion and research.

References

[1] D.H. Albonesi and I. Koren, "Tradeoffs in the Design of Single Chip Multiprocessors," *2nd International Conference on Parallel Architectures and Compilation Techniques (PACT94)*, pp. 25-34, 1994.

[2] B.R. Allison and C. van Ingen, "Technical Description of the DEC 7000 and DEC 10000 AXP Family," *Digital Technical Journal*, Vol. 4, No. 4, pp. 100-110, 1992.

[3] D. Alpert and D. Avnon, "Architecture of the Pentium Microprocessor," *IEEE Micro*, pp. 11-21, June 1993.

[4] T. Asprey, et al, "Performance Features of the PA7100 Microprocessor," *IEEE Micro*, pp. 22-35, June 1993.

[5] M. Chiang and G.S. Sohi, "Experience with Mean Value Analysis Models for Evaluating Shared Bus, Throughput-Oriented Multiprocessors," *SIGMETRICS'91*, pp. 90-100, May 1991.

[6] M. Chiang and G.S. Sohi, "Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput-Oriented Environment," *IEEE Transactions on Computers*, pp. 297-317, March 1992.

[7] P.K. Dubey, G.B. Adams III, and M.J. Flynn, "Instruction Window Size Trade-Offs and Characterization of Program Parallelism," *IEEE Transactions on Computers*, pp. 431-442, April 1994.

[8] K.I. Farkas and N.P. Jouppi, "Complexity/Performance Tradeoffs with Non-Blocking Loads," *21st International Symposium on Computer Architecture*, pp. 211-222, April 1994.

[9] J.R. Goodman and P.J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *15th International Symposium on Computer Architecture*, pp. 422-431, June 1988.

[10] R. Jog, P.L. Vitale, and J.R. Callister, "Performance Evaluation of a Commercial Cache-Coherent Shared Memory Multiprocessor," *SIGMETRICS'90*, pp, 173-182, May 1990.

[11] E. McLellan, "The Alpha AXP Architecture and 21064 Processor," *IEEE Micro*, pp. 36-47, June 1993.

[12] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik, *Quantative System Performance, Computer Analysis Using Queuing Network Models*, Prentice Hall, Englewood Cliffs, N.J., 1991.

[13] D. Lenoski, et al, "The DASH Prototype: Implementation and Performance," *19th International Symposium on Computer Architecture*, pp. 92-103, May 1992.

[14] S.T. Leutenegger and M.K. Vernon, "A Mean-Value Performance Analysis of a New Multiprocessor Architecture," *SIGMETRICS'88*, pp, 167-176, May 1988.

[15] P. Rubinfeld, "An Overview of the 21164 Alpha AXP Microprocessor," *Hot Chips VI*, August 1994.

[16] R.L. Sites, "Alpha AXP Architecture," *Digital Technical Journal*, Vol. 4, No. 4, pp. 19-34, 1992.

[17] G. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 53-61, April 1991.

[18] C.P. Thacker, D.G. Conroy, and L.C. Stewart, "The Alpha Demonstration Unit: A High-performance Multiprocessor for Software and Chip Development," *Digital Technical Journal*, Vol. 4, No. 4, pp. 51-65, 1992.

[19] J. Torrellas, J. Hennessy, and T. Weil, "Analysis of Critical Architectural and Program Parameters in a Hierarchical Shared-Memory Multiprocessor," *SIGMETRICS'90*, pp. 163-172, May 1990.

[20] M.K. Vernon, E.D. Lazowska, and J. Zahorjan, "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache Consistency Protocols," *15th International Symposium on Computer Architecture*, pp. 308-315, June 1988.

[21] M.K. Vernon, R. Jog, and G.S. Sohi, "Performance Analysis of Hierarchical Cache-Consistent Multiprocessors," *Performance Evaluation*, Vol. 9, pp, 287-302, 1989.

Parameter	Description	Value
P_{load}	Prob instruction is a Load	0.25
P_{store}	Prob instruction is a Store	0.10
$P_{L2\_write\_shared}$	Prob Store hit to shared data	0.05
$P_{L2\_write\_private}$	Prob Store hit to private data	0.01

Table 2: Default Values for Local Cache Workload Parameters

Parameter	Description	Value
$P_{L1Rem\_di\_hit}$	Prob L1 data invalidate hit	0.02
$P_{L2Rem\_di\_hit}$	Prob L2 data invalidate hit	0.01
$P_{L2Rem\_du\_hit}$	Prob L2 data update hit	0.02
$P_{L2Rem\_dpriv\_hit}$	Prob L2 data mod hit (load)	0.01
$P_{L2Rem\_dmod\_hit}$	Prob L2 data mod hit (store)	0.03

Table 3: Default Values for Remote Cache Workload Parameters

Parameter	Description	Value
CPI_{proc}	CPI with no memory overhead	0.7
N_{Ifetch}	# of instructions fetched each CPIF	2.0
$N_{Daccess}$	# of Dcache accesses each CPIF	0.7

Table 4: Default Values for Processor Parameters

Parameter	Description	Value
$P_{L1\_ir\_miss}$	Prob of L1 Icache miss	0.03
$P_{L1\_dl\_miss}$	Prob of L1 Dcache load miss	0.13
$P_{L2\_ir\_miss}$	Prob of L2 cache instruction miss	0.01
$P_{L2\_dl\_miss}$	Prob of L2 cache data load miss	0.05
$P_{L2\_ds\_miss}$	Prob of L2 cache data store miss	0.08
$P_{L2\_ir\_victim}$	Prob L2 instr miss causes wrback	0.20
$P_{L2\_dl\_victim}$	Prob L2 load miss causes wrback	0.25
$P_{L2\_ds\_victim}$	Prob L2 store miss causes wrback	0.25
N_{wbuf}	# of write buffer entries	4
N_{wait_wbuf}	# entries retired before new write	3
P_{full_wbuf}	Prob full wr buffer on data store	0.01
N_{vbuf}	# of victim buffer entries	2
N_{wait_vbuf}	# entries retired before new write	1
P_{full_vbuf}	Prob full victim buffer on wrback	0.01

Table 5: Default Values for Cache Hierarchy Parameters

Parameter	Description	Value
N_{proc}	# of processors	x (variable)
N_{mem}	# of main memory modules	2x
T_{arb}	# of cycles for arbitration	2

Table 6: Default Values for Bus Parameters

Parameter	Description	Cycles
T_{stall_ir}	Icache miss stall time	45
T_{stall_dl}	Dcache load miss stall time	45
T_{stall_ds}	Dcache store miss stall time	45
T_{stall_dlwb}	Dcache load miss wrback ack time	6
T_{stall_dswb}	Dcache store miss wrback ack time	6
T_{trans_ir}	Icache miss trans time	47
T_{trans_dl}	Dcache load miss trans time	47
T_{trans_ds}	Dcache store miss trans time	47
T_{trans_irwb}	Icache miss wrback trans time	42
T_{trans_dlwb}	Dcache load miss wrback trans time	42
T_{trans_dswb}	Dcache store miss wrback trans time	42

Table 7: Default Values for Memory Stall/Transaction Time Parameters