To appear in 25th Annual International Symposium on Computer Architecture

Analytic Evaluation of Shared-Memory Systems with ILP Processors *

Daniel J. Sorin†, Vijay S. Pai‡, Sarita V. Adve‡, Mary K. Vernon†, and David A. Wood†

†Computer Sciences Dept
University of Wisconsin - Madison
{sorin, vernon, david}@cs.wisc.edu

‡Dept of Electrical & Computer Engineering
Rice University
{vijaypai, sarita}@rice.edu

Abstract

This paper develops and validates an analytical model for evaluating various types of architectural alternatives for shared-memory systems with processors that aggressively exploit instruction-level parallelism. Compared to simulation, the analytical model is many orders of magnitude faster to solve, yielding highly accurate system performance estimates in seconds.

The model input parameters characterize the ability of an application to exploit instruction-level parallelism as well as the interaction between the application and the memory system architecture. A trace-driven simulation methodology is developed that allows these parameters to be generated over 100 times faster than with a detailed execution-driven simulator.

Finally, this paper shows that the analytical model can be used to gain insights into application performance and to evaluate architectural design trade-offs.

1 Introduction

Shared memory multiprocessors are gaining wide popularity as platforms for technical and commercial computing. Computer architects have generally relied on simulation for designing shared-memory systems. However, an architectural simulator for shared-memory systems with processors that aggressively exploit instruction-level parallelism (ILP) requires several hours to simulate a few seconds of real execution time with reasonable accuracy [6]. This severely restricts the application and architectural design space that can be explored.

*This research is supported in part by DARPA/ITO under Contract N66001-97-C-8533 and by the National Science Foundation under Grants HRD-9896132, MIP-9625558, CDA-9623632, CCR-9410457, CCR-9502500, CDA-9502791, and CDA-9617383. Sarita V. Adve is also supported in part by an IBM University Partnership award and by the Texas Advanced Technology Program under Grant No. 003604-025. Vijay S. Pai is supported by a Fannie and John Hertz Foundation Fellowship.

This paper takes an alternate, potentially complementary, approach. We develop and validate an analytic model for evaluating architectural trade-offs for shared-memory systems with ILP processors. The model validates extremely well against detailed execution-driven simulation and produces each result in a few seconds. Thus, the model can be a useful tool for culling the design space, and then simulation can be used for further studies of the important regions. In addition, the model input parameter values yield insight into how particular applications (and current compiler technology) interact with the memory system.

One challenge in developing a high-fidelity analytical model of a complex architecture is to create a tractable system of equations that represents all of the system details that have non-negligible impact on the predicted performance measures. Another significant challenge is to create a model that has a relatively small set of input parameters that are easy to measure or estimate. A key question addressed in this research is whether a highly accurate yet tractable system of equations with fairly simple input parameters can be created for complex parallel ILP-processor architectures.

We are aware of two previous analytical models of multiprocessors that have non-blocking caches [4, 27]. The model by Albonesi and Koren [4] was not validated and has at least two significant drawbacks: (1) the number of memory reads that are issued before the next read blocks is assumed to be fixed, whereas that number changes dynamically for ILP processors, and (2) some of the fixed model input parameters, such as the probability the write buffer is full or the percent overlap between memory read latency and computation, depend on the outputs the model is supposed to compute. The model by Willick and Eager [27] also assumes a fixed limit on the number of outstanding memory requests (e.g., a hardware upper bound), and was validated against a simulation that had statistical workload assumptions similar to the analytical model. We extend the Willick and Eager model in a number of significant ways, and we validate our model against detailed simulation of applications on a modern architecture.

The key features of the model in this paper are:

- The ILP processor and its associated two-level cache system are viewed as a black box that generates requests to the memory system and intermittently blocks after a *dynamically changing* number of requests. Parameters that characterize this black box, including the time between level two (L2) cache misses, the distribution of the number of outstanding requests before a processor blocks, and the ratio of requests that are satisfied by the local vs. remote memory, lead to insights into application behavior (and current compiler technology) that are discussed in Section 6.

- We iterate between two submodels. One submodel computes the processor stall time due to load misses that cannot be retired until the data returns from memory. The other submodel computes the stall time due to the hardware constraint on the total number of outstanding memory requests.

- In each submodel, the memory system is viewed as a system of queues (e.g., the memory bus, DRAM modules and associated directories, and network interfaces) and delay centers (e.g., switches in the interconnection network). We create a set of intuitive *customized* mean value analysis (CMVA) equations [26] to obtain the estimates of processor stall time in each submodel. The CMVA technique has proven to be accurate in validation experiments for a number of simpler architectural models [26, 5].

- We show that reasonable approximations of key input parameters that characterize the application behavior can be obtained from a high-level simulator, *FastILP*, that runs two orders of magnitude faster than the detailed simulation. Moreover, other input parameters can also be obtained from very fast simulation or may be varied within ranges that have been observed for similar applications.

The analytic model estimates processor throughput within 1-12% of the estimates from detailed simulation for several complex applications and architectural configurations. Although the input parameter values that characterize application behavior are obtained from simulation, the model's architectural parameters can be varied quickly and easily using just the analytical model. We will illustrate the use of the analytic model to cull the system design space in Section 6.

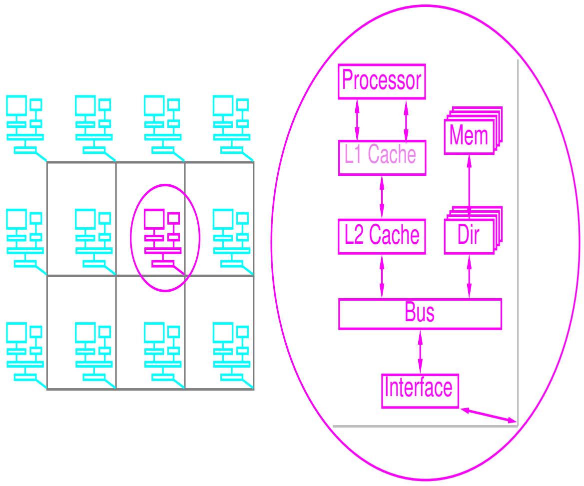

Figure 1. Parallel System Architecture

2 System Architecture

The architecture chosen for this study is a cache-coherent, release consistent shared-memory multiprocessor system where the processing nodes are connected by a mesh interconnection network, as shown in Figure 1. The architecture is modeled in RSIM [20], a detailed execution-driven simulator for shared-memory multiprocessors with ILP processors against which we validate our model. However, with fairly straightforward modifications, the model can easily be applied to variations on this architecture (e.g., other cache coherence protocols, changes in the interconnect or memory system organization, etc.)

The processor exploits ILP using features such as multiple issue, out-of-order scheduling, non-blocking loads, and speculative execution. Instructions are fetched into the instruction window, and they are issued to the functional units when all of their dependences are satisfied. The instructions are fetched into and retired from the window in program order, but they may be issued to the functional units out of program order. In particular, because the system is release consistent, loads and stores can execute out of order.

To maintain precise interrupts, stores issue to the memory system only when they reach the top of the instruction window. Release consistency allows a store to retire from the instruction window even while it is not yet complete (or even issued) in the memory system. Thus, stores do not directly block the processor. Except for stores, an instruction can retire from the instruction window only after it completes execution. An implication of this requirement is that when a load reaches the top of the instruction window, retirement must stall if the value of the load has not yet returned.

The L1 cache is write-through, multiported, and non-blocking. The L2 cache is writeback, write-allocate, non-blocking, and fully pipelined. The caches use miss status holding registers (MSHRs) to track the status of all out-

parameter	description	value
N	number of nodes	
m	memory modules per node	4
M_{hw}	number of MSHRs	8
$S_{NI_{out,r}}$	NI send occupancy for request	8
$S_{NI_{out,d}}$	NI send occupancy for data	8
$S_{NI_{in,r}}$	NI receive occupancy for request	8
$S_{NI_{in,d}}$	NI receive occupancy for data	8
$S_{bus,r}$	bus occupancy for request	4
$S_{bus,d}$	bus occupancy for data	12
S_{mem}	memory/directory (DRAM) access	40
S_{tag}	L2 tag check	4
S_{switch}	per-word network switch occupancy	8

Table 1. System Architecture Parameters

standing misses [14]. Misses to the same cache line are *coalesced* in the MSHRs; only one memory request is generated for such coalesced misses.

The memory and directory are interleaved. Directory accesses are overlapped with memory accesses. The bus is split transaction. All traffic out of the node goes through the send network interface (NI) via the bus, and all traffic into the node comes from the receive NI via the bus. The mesh interconnection network uses wormhole-routing. Separate request and reply networks are used for deadlock avoidance.

Cache coherence is maintained by a fairly standard three-state (MSI) directory-based invalidation protocol. The protocol supports cache-to-cache transfers in the case of a request for data that is dirty in a remote cache. For deadlock avoidance, writebacks use the reply network and do not generate acknowledgments. Consequently, they do not reserve MSHRs or other cache resources.

3 Model Parameters

Table 1 defines the system architecture parameters, including the values that are used in the baseline architecture in Section 5. Occupancies are in units of CPU cycles.

In any modeling study, defining a good set of application or workload parameters poses a significant challenge. Table 2 summarizes the application parameters that were developed with the following goals: (1) able to capture the principal performance-determining characteristics of the workload for the intended applications of the model, (2) relatively few and simple to measure (making the model practically useful), and (3) insensitive to changes in the architectural parameters that will be varied to cull the system design space.

The first five parameters in Table 2 characterize the ability of the processor to overlap multiple memory requests while running a given compiled application (or set of applications). These parameters, referred to as ILP parameters, are discussed in more detail below. The other parameters in the table are standard parameters for models of architectures based on directory coherence protocols [1]. Further description of those parameters is omitted due to space constraints.

Note that the parameters are defined for *homogeneous applications*; that is, each processor has the same value for each parameter in the table, and memory requests are assumed to be equally distributed across the relevant memory modules (local or remote) due to interleaving and effective data layout. There is a natural extension of these parameters for non-homogeneous applications, but for simplicity in the model exposition we use the given parameters and validate against homogeneous applications in Section 5.

The parameter τ is the average time between requests generated by the processor to the (main) memory subsystem, not including the time that the processor is stalled or is spin-waiting on a synchronization event such as a lock release, flag, or barrier completion. We also measure the coefficient of variation of τ, CV_τ. τ is well-defined for simple processors that block on each load and store, whereas the notion that a complex modern processor is stalled has several possible definitions. For the robust parameter τ that is needed for the model, the processor is defined to be stalled when it is *completely stalled*; that is, the functional units are completely idle, no further instructions can be retired or issued until data returns from memory, and all outstanding cache requests are waiting for data from main memory. The fraction of time a processor is completely stalled is one of the performance metrics estimated by the analytic model. The parameter τ does not include this time.

The $f_{synch-write}$ parameter is the fraction of write requests that are synchronous; that is, they are generated by a read-modify-write request or they coalesce with at least one later read miss. The significance of this parameter will be discussed in Section 5. Read misses that coalesce with earlier read requests are completely invisible to the model because they do not generate any memory system traffic and they do not cause any new blocking behavior. Thus, a parameter for the frequency of read-read coalescing is not needed. Likewise for writes that coalesce with previous transactions.

The set of parameters f_M, $M \geq 1$, measured for a number of MSHRs larger than the maximum value that will be evaluated with the model, are the *fractions of processor stall periods* that have M MSHRs occupied with read misses. Note that if a read miss occurs for a line that has a prior write miss outstanding, then the miss is counted as a read miss when measuring M. Also note that misspeculated reads are counted in M. The f_M parameters are unique to a system with non-blocking loads.

Parameter	Description
τ	Average time between read, write, or upgrade requests to memory, not counting the time when the processor is completely stalled or is spin-waiting on a synchronization event
CV_τ	Coefficient of Variation of τ
$f_{synch-write}$	Fraction of write requests that are generated by atomic read-modify-write instructions or that coalesce with at least one later read
f_M	Fraction of processor stalls that find M MSHRs with outstanding read requests
$P_{read}, P_{write}, P_{upgrade}$	Probability that a memory request is a read, write, or upgrade
P_{wb}	Probability that a read or write request causes a writeback of a cache block
$P_{L\|x}$	Probability directory is local for a type x transaction; x=read, write, upgrade, writeback
$P_{M\|x,y}$	Probability home memory can supply the data for a type x, y request; x=read, write; y=local home, remote home
$P_{3hop\|x\&not-memory}$	Probability that a request of type x to a remote home is forwarded to a cache at a third node; x=read,write
H	Average number of network switches traversed by a packet
X	Average number of invalidates caused by a write or upgrade to a clean line

Table 2. Application Parameters

We have verified that the application input parameters are *relatively insensitive to changes in the architectural parameters* that can be varied in the model (e.g., the number of MSHRs, the speed of the bus and interconnection network switches, main memory configuration, etc.). However τ, f_M, and $f_{synch-write}$ are sensitive to various parameters of the processor and cache subsystem, such as the instruction window size. This is a key motivation for investigating fast parameter estimation methods, as discussed next.

3.1 Quickly Estimating Application Parameters

Application parameters other than τ, CV_τ, f_M, and the part of $f_{synch-write}$ that is due to writes coalescing with later memory read requests, do not depend directly on ILP features and can thus be measured using current fast simulators for multiprocessors with simple single-issue processors (e.g., [28]).

The remainder of this section provides an overview of FastILP, a fast high-level simulator for quickly estimating the ILP parameters τ, CV_τ, f_M, and $f_{synch-write}$. Since FastILP does not need to measure the exact cycle count for an execution, it can achieve very high performance by abstracting both the ILP processor and the memory system, and modeling only enough state to generate the required ILP parameters. FastILP differs from conventional cycle by cycle ILP-based multiprocessor simulators in three key ways.

First, FastILP speeds up processor simulation using techniques from DirectRSIM, a simulator designed for speeding up accurate timing simulation of ILP-based multiprocessors [6]. Each instruction in FastILP sets the timestamp of its destination register based on the completion time for that instruction. For non-memory instructions, the completion time is determined by the timestamps of the source registers of the instruction and the availability of the appropriate functional unit. For memory instructions, the processor keeps enough state information to simulate memory disambiguation. The completion timestamp calculation for a memory request is unique to FastILP, as described below.

Second, FastILP speeds up memory system simulation by taking advantage of two observations: the ILP parameters are not very sensitive to the exact latencies or configuration of the memory system, and L2 cache misses have high latencies that can be overlapped effectively only with other memory misses [21]. Using these observations, FastILP does not explicitly simulate any part of the memory system beyond the cache hierarchy. FastILP divides simulated time into distinct "eras," which start when one or more memory replies unblock the processor and end when the processor blocks again waiting for a memory reply. No memory replies return *during* an era. One or more replies return together at the beginning of each era, depending on whether the processor has enough work to completely overlap the time between incoming replies. For simplicity, in the experiments in Section 5.2, we assume for a 64-element instruction window that memory responses return in the order that the respective requests were generated, one at a time. For a 128-element instruction window we assume that all memory requests outstanding at the end of an era return together at the start of the next era (i.e., computation fully overlaps the time between these responses in the real system).

The use of eras allows FastILP to compute timestamps for load and store instructions, with each timestamp including both the era in which the data returns, along with the cycle within the era. The parameters τ and CV_τ are calculated according to the points within each era at which misses occur; f_M is measured by counting the read requests outstanding at the end of each era. In this fashion, FastILP can process all instructions in-order, while still simulating an out-of-order processor.

Third, FastILP further speeds up simulation time by using trace-driven (as opposed to execution-driven) simulation and by simulating the trace of only one processor. The use of trace-driven simulation is possible because parameters are estimated for homogeneous applications and synchronization spin time is not measured in τ. Further, FastILP makes an approximation that mispredicted execution paths do not have a significant impact on the ILP parameters; this assumption is valid for the applications validated in Section 5.2. FastILP assumes homogeneous applications, allowing it to use the trace of only a single processor, where the trace provides information about memory accesses known to be communication misses. As communication misses generally stem from application and data set characteristics rather than processor microarchitecture or system latencies, such traces can be quickly generated by an appropriately instrumented fast simulator for multiprocessors with simple processors or by a multiprocessor trace-generation utility.

Using the above optimizations, FastILP achieves two orders of magnitude speedup over RSIM, and more than an order of magnitude speedup over DirectRSIM.

4 The Analytic Model

The principal output measure computed by the model is the system throughput, measured in instructions retired per cycle (IPC). This throughput is computed as a function of the input parameters that characterize the workload and the memory architecture.

The baseline model defined in this section assumes that the directory is implemented in DRAM and that the directory lookup is coupled with memory access, so a single service time applies to the parallel memory and directory lookup. Variations on this directory organization are modeled in Section 6.

4.1 Model Overview

We use the term *synchronous* for read requests (and for read-modify-write requests) because the data must return before a load (or read-modify-write) instruction is retired from the instruction window. Other requests (writes, upgrades, writebacks, invalidates, and acknowledgments) are *asynchronous*.

A key question in developing the analytic model is how to compute throughput as a function of the dynamically changing number of outstanding memory requests that can be issued before the processor must stall waiting for data to return from memory. We address this issue by iterating between the following two submodels for each value of M, $1 \leq M < M_{hw}$:

- *the synchronous blocking submodel (SB)* that computes the fraction of time the processor is stalled due to load or read-modify-write instructions that cannot be retired until the data returns from memory,
- *the MSHR blocking submodel (MB)* that computes the additional fraction of time the processor is stalled purely due to the MSHRs being full.

For $M = M_{hw}$, we compute throughput from a modified version of the MSHR-blocking submodel alone, as explained below. Once these throughputs are computed, we compute the *weighted sum* of the throughputs, weighted by the frequency of each throughput value that would be observed for the number of MSHRs in the system. This frequency can in turn be computed from the model input parameters, f_M. The remainder of this section gives the most pertinent details of the two submodels as well as how slowdown due to synchronization delays is computed; the full set of equations for the submodels is given in [25].

Each of the two submodels (SB and MB) contains the same set of customized MVA equations [26] to compute the delay for a transaction in the memory subsystem (see Section 4.2). In the SB submodel, the number of customers per processor is equal to the maximum number of read requests that can be issued before the processor blocks (i.e., one of the observed values of M). The processor (and its associated cache subsystem) is a FCFS queue that initially has mean service time equal to τ. Note that this queue is only idle when M memory read requests are outstanding; otherwise it is generating memory requests at rate $1/\tau$. If the request is a write miss, the customer is routed immediately back to the processor while simultaneously forking an asynchronous memory write or upgrade transaction, using the technique proposed by Heidelberger and Trivedi [10].

In the MB submodel, the number of customers per processor is equal to the number of MSHRs, M_{hw}. MSHRs can be occupied by read, write, or upgrade requests; however, for architectures with non-blocking stores and in-order retirement of loads and for $M < M_{hw}$, all of the blocking time when MSHRs contain both read and write requests is accounted for in the SB submodel. In this case, the additional blocking time that needs to be computed by the MB model is for the case that all MSHRs are occupied by write or upgrade memory transactions (or writebacks, if they occupy MSHRs in the architecture of interest). The customers in the MB model thus represent the behavior of write and upgrade memory transactions. When read misses occur, these customers are immediately routed back to the processor (since the processor cannot stall on read misses in this submodel) while simultaneously forking a read transaction to the memory system, again using the technique in [10].

The mean time that each customer occupies the processor in the MB model is equal to τ adjusted to reflect the fraction of time that the processor is stalled due to load or read-modify-write instructions that cannot be retired (computed from the SB model). That is, $\tau_{MB} = \frac{\tau}{U_{SB}}$, where U_{SB} denotes the fraction of time the processor is not stalled in the SB model. Once the measures are computed from the MB model, the SB model is solved again using $\tau_{SB} = \frac{\tau}{U_{MB}}$. The alternating solution of the submodels is repeated until the estimated throughputs converge. This approach is similar to the method of complementary delays [9, 13].

The SB and MB submodels are each similar to Willick and Eager's model [27] except that: (1) transaction routing is according to the cache coherence protocol, (2) the switching network is configured as a two-dimensional mesh and the delay per switch is modeled as an average quantity measured directly in the system or by simulating over a number of applications, (3) contention for the memory bus and network interface is modeled, (4) the service time at the processor node is inflated to account for the stall time estimated by the other submodel, and (5) we use an approximation to account for high measured coefficient of variation in τ, CV_τ, which is discussed in Section 5.

For the case that $M = M_{hw}$, all processor stalls can be attributed to full MSHRs. In this case, we solve a modified MB model in which there are M_{hw} customers per processor and these customers represent the behavior of read, write and upgrade memory system transactions. For any of these memory requests, the customer leaves the processor and visits the appropriate memory system resources. (Write-backs are forked asynchronously for the base architecture in RSIM).

Once throughput is computed from the weighted average of the value at each M, synchronization effects are accounted for as follows. If there are any locks that have significant contention, a separate simple queueing model is used to estimate mean delay for each lock, using (1) the number of processors that compete for the lock, (2) a measured average number of instructions between accesses to the lock, and (3) the average lock holding time (in number of instructions). Finally, the throughput slowdown due to barriers [3] is computed from the average number of instructions and lock delays between each barrier (for load-balanced barriers) or the number of such instructions and lock delays for each processor participating in the barrier (unbalanced barriers), in addition to the estimated time to execute a perfectly load-balanced barrier on the given memory architecture. Calculation of throughput slowdown due to pairwise synchronization is beyond the scope of this paper.

4.2 Model Equations

As mentioned above, the SB and MB submodels use a set of customized MVA (CMVA) equations to compute the mean delay for customers at the processor, local and remote memory buses, directories (and associated memory modules), and network interfaces. Fixed delays are assumed for resources that have negligible contention (e.g., cache tag checks) and for the approximate average delay at each network switch (observed during measurement or simulation of several applications).[1] The CMVA equations, explained in detail in [25], are briefly outlined below.

The total average round-trip time in either submodel is the sum of the customer's mean residence time at each of the resources that it visits. Thus, the average round-trip time for customers in the SB submodel has the form:

$$R_{SB} = R_{processor} + R^{synch}_{NIout} + R^{synch}_{network}$$
$$+R^{synch}_{NIin} + R^{synch}_{bus} + R^{synch}_{dir-and-mem} + Z,$$

where R^{synch}_j is the total average residence time for a read transaction at a (set of) memory system resource(s) j, and Z is the total fixed delay for a read request. A similar equation holds for the average round-trip time for customers in the MB submodel. Furthermore, system throughput, measured in number of read transactions retired per unit time, is equal to M/R_{SB}.

To illustrate the calculation of R^{synch}_j, we show how R^{synch}_{NIout} is computed, where $NIout$ denotes the queues that are used to transmit a message into the switching network. R^{synch}_{NIout} is the sum over all such queues of the total mean delay for each type of synchronous transaction that can visit the queue as either a short (request, r) or long (data, d) message:

$$R^{synch}_{NIout} = \textstyle\sum_y (R^{synch}_{NIout_{local,y,r}} + R^{synch}_{NIout_{local,y,d}})$$
$$+(N-1)\textstyle\sum_y (R^{synch}_{NIout_{remote,y,r}} + R^{synch}_{NIout_{remote,y,d}})$$

The $N-1$ factor in the remote terms represents the $N-1$ remote NIs for any particular processor's transactions. The memory transactions, y, are defined in Tables 4 and 5.

To compute $R^{synch}_{NIout_{local,y,x}}$, we multiply the average number of visits that a synchronous type x message from a type y transaction makes to the local NI, by the sum of the average waiting and service times at the queue:

$$R^{synch}_{NIout_{local,y,x}} =$$

[1] Note that the model is validated in Section 5 against detailed simulation of the contention at the network switches for each application, including applications that were not used to estimate average switch delay.

$$P_y V^{synch}_{NIout_{local,y,x}} (W_{NIout_{local}} + S_{NIout,x})$$

The utilization of the local outgoing NI queue by type x messages of a type y transaction is equal to the average total number of visits for these messages (per round trip in the SB model) times their service time ($S_{NIout,x}$), divided by the average round trip time of the transactions in the SB model (R_{SB}):

$$U^z_{NIout_{loc,y,x}} = (\frac{P_y}{R}) V^z_{NIq_{loc,y,x}} S_{NIout,x}$$

Note that in the above equation, z denotes either the synchronous (S) or the asynchronous (A) part of the type y transaction. This equation and the next two equations illustrate how the interference by the asynchronous transactions is accounted for in the SB model.

The average waiting time at the outgoing local NI queue due to traffic from remote nodes is equal to the sum over all transaction types of the synchronous and asynchronous traffic generated remotely that is seen at the queue:

$$W^{remote}_{NIout_{local}} = \sum_y (W^{remote,y^S}_{NIout_{local}} + W^{remote,y^A}_{NIout_{loc}})$$

The contribution to the waiting time at queue q of the local NI by remote traffic of type y that is either synchronous or asynchronous (depending on z) is equal to the sum over all message types x of the total number of remote customers times the waiting time that their type y transactions (synchronous or asynchronous) cause on local queue q. This waiting time is equal to the time that a customer would wait for those customers in the queue plus the time that the customer would wait for the customer in service. The residual life, $\frac{S_{NIq,x}}{2}$, assumes a deterministic service time at the NI.

$$W^{remote,y^z}_{NIout_{local}} = \sum_x (N-1) M \Bigg[(\frac{R^z_{NIout_{remote,y,x}}}{R} - U^z_{NIout_{remote,y,x}}) S_{NIout,x} + (U^z_{NIq_{remote,y,x}})(\frac{S_{NIout,x}}{2}) \Bigg]$$

5 Model Validations

In this section we present the results of validation experiments that assess the accuracy of the analytic model and of the FastILP parameter estimates. The validations were performed against the RSIM execution-driven simulator [20]. Section 5.1 presents the applications used in the validations and the model input parameters for those applications measured by RSIM. Section 5.2 compares the inputs generated by FastILP to those obtained by RSIM, and Section 5.3 presents the results of the analytic model validations.

5.1 Applications Used in Model Validations

The validation experiments include the following applications: FFT, LU, and Radix from the SPLASH-2 suites [29], Water from the SPLASH suite [24], and Erlebacher from the Rice parallel compiler group [2].[2] We also use versions of LU and of FFT (denoted by *opt*) that are optimized for ILP systems by applying loop interchange to schedule read misses closer together, thus better overlapping their latencies [21]. The optimization in FFTopt has the side effect that all read requests overlapped at any given time from a single processor go to the same memory bank. This causes the effective number of memory modules per node to be equal to one. Additionally, both versions of LU are modified to achieve better load balance by using pairwise synchronization (implemented with flags) instead of global barriers.

The RSIM-measured values of the first five model input parameters for each of the applications, for various system configurations (i.e., number of processors, n, and instruction window size, w), are shown in Table 3.[3] The input sizes for FFT and Radix are greater than or equal to those specified in the SPLASH/SPLASH2 distributions. The input sizes for LU and Water are slightly smaller than recommended due to the long time for running the RSIM simulations; however, the number of processors is appropriately scaled down to ensure a reasonable speedup. The $f_{synch-write}$ parameter is omitted from the table since it is very small except as noted below.

Tables 4 and 5 provide the RSIM-measured probabilities of the various locations where read misses, write misses, upgrades, and writebacks are serviced in the memory system, for a system with a 64 entry window size. The stats are nearly identical for the 128 entry window size configuration. These probabilities are computed in a straightforward way from the basic application memory request parameters (see Table 2). Note that the probabilities (not including writebacks) sum to one. Writebacks are additional asynchronous transactions that are forked off from reads and writes. Collectively, the input parameter values reflect a set of applications with fairly diverse characteristics.

[2] We also attempted to validate the analytic model against MP3D, but we discovered that MP3D has relatively minor but still significant non-homogeneity in its memory access behavior, which led to approximately a 20% error in throughput predicted by the analytic model. Model modifications for non-homogeneous applications are straightforward but beyond the scope of this paper. We therefore omit those results in the remainder of this section.

[3] The baseline measures for the model input parameters used the following processor/cache subsystem configuration: maximum fetch/decode/retire rate = 4, instruction window size = 64, L1/L2 cache size = 16KB/64KB (scaled based on application input sizes [29]), cache line size = 64 bytes, L1/L2 associativity = 1/4, L1/L2 hit time = 1/13 cycles, L1/L2 ports = 2/1. The measures with instruction window size of 128 entries also doubled the other processor resources (e.g., decode width) compared to the baseline.

app	input size	configuration	τ	cVar(τ)	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	$\Sigma f_i, i > 8$
Erle	64x64x64	n16, w64	39.0	10.9	.65	.17	.09	.08	0	.01	0	0	0
		n16, w128	26.9	5.0	.64	.11	.10	.10	.02	.01	.01	.01	.01
FFTopt	64K	n16, w64	64.9	12.8	.42	.17	.03	.04	.35	0	0	0	0
		n16, w128	37.6	11.8	.12	.28	.18	.03	.03	.03	.03	.03	.25
FFT	64K	n16, w64	63.9	12.8	.53	.47	0	0	0	0	0	0	0
		n16, w128	37.0	12.0	.12	.48	.39	0	0	0	0	0	0
LUopt	256x256	n8, w64	120.8	6.3	.12	.07	.06	.06	.06	.06	.06	.06	.45
LU	256x256	n8, w64	108.1	8.1	.51	.49	0	0	0	0	0	0	0
		n8, w128	74.0	3.6	.10	.19	.70	.01	0	0	0	0	0
Radix	512K	n8, w64	80.9	2.0	.99	.01	0	0	0	0	0	0	0
Water	343	n8, w64	593.2	2.5	.73	.25	.01	0	0	0	0	0	0
		n8, w128	487.7	2.5	.49	.48	.01	.01	0	0	0	0	0

Table 3. Application Input Parameters

		Reads					Upgrades	
		local home		remote home			local	remote
app	config	memory	remote cache	memory	cache at home	cache at non-home		
Erle	n16	.49	0	.09	.04	.01	.23	.01
FFTopt	n16	.30	0	.19	.04	0	.14	0
FFT	n16	.30	0	.20	.04	0	.14	0
LUopt	n8	.16	0	.47	.01	.02	.08	.25
LU	n8	.16	0	.48	.01	.02	.08	.25
Radix	n8	.29	.01	0	.01	0	.01	0
Water	n8	.02	.12	.02	.11	.28	.11	.31

Table 4. Read and Upgrade Transactions

		Writes					Writebacks	
		local home		remote home			local	remote
app	config	memory	remote cache	memory	cache at home	cache at non-home		
Erle	n16	.13	0	0	0	0	.32	0
FFTopt	n16	.32	0	0	0	0	.42	0
FFT	n16	.32	0	0	0	0	.42	0
LUopt	n8	0	0	.01	0	0	.08	.23
LU	n8	0	0	0	0	0	.08	.23
Radix	n8	.10	0	.58	0	0	.10	.57
Water	n8	0	0	0	0	.03	0	0

Table 5. Write and Writeback Transactions

application	config	τ	CV_τ	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	$\Sigma f_i, i > 8$	% error
Erle	n16, w64	42.1	10.7	.61	.18	.11	.08	0	.01	0	0	0	-1.1
	n16, w128	23.2	9.6	.82	.02	0	.13	.02	0	0	.01	0	-0.5
FFTopt	n16, w64	64.4	11.9	.42	.20	.03	.03	.33	0	0	0	0	0.7
	n16, w128	31.2	12.6	.18	.43	.23	0	0	.08	0	0	.08	-6.2
FFT	n16, w64	56.6	12.5	.50	.50	0	0	0	0	0	0	0	9.1
	n16, w128	30.8	12.6	.14	.44	.43	0	0	0	0	0	0	-11.9
LUopt	n8, w64	122.2	4.0	.12	.07	.06	.06	.06	.06	.06	.06	.47	-2.5
LU	n8, w64	104.6	3.8	.53	.47	0	0	0	0	0	0	0	-0.1
	n8, w128	78.6	3.4	.20	.06	.74	0	0	0	0	0	0	-8.6
Radix	n8, w64	69.2	2.0	1.0	0	0	0	0	0	0	0	0	-6.8
Water	n8, w64	418.8	3.1	.71	.26	.01	0	0	0	0	0	0	20.0
	n8, w128	270.4	3.8	.05	.93	0	.01	0	0	0	0	0	59.5

Table 6. Accuracy of the FastILP Input Parameters

5.2 Accuracy of the FastILP Input Parameters

Table 6 gives the input parameters generated by FastILP and the percentage error between the throughputs calculated using these parameters versus the parameters generated by RSIM. These results show that for all configurations studied and all applications except Water[4], FastILP can generate parameters that lead to less than 12% error in throughput relative to those generated by RSIM.

5.3 Accuracy of the Analytic Model

The experiments in which we compared the performance predicted by the analytic model against the performance reported by RSIM quickly revealed two additional behaviors that needed to be captured in the model:

- Since the time between memory requests (τ) has relatively high variance (see Table 3), the standard approximate MVA equation estimates that a customer arriving back to the processor waits, on average, a fairly long time for the residual life of a customer that is already in service.[5] However, since the average latency for a memory transaction in the modeled system is (in some cases, significantly) smaller than this mean residual life, the customer arriving back to the processor is not arriving at a random point in time relative to the service time at the processor. To produce a more accurate estimate of processor residence time, we approximate the residual life using an interpolation [7] between τ, which is the residual life when the memory transaction takes zero time, and the MVA residual life formula for a random arrival.[6]

- Water (and MP3D) has a non-negligible value for $f_{synch-write}$, the fraction of write requests that are generated by read-modify-write instructions or that coalesce with at least one later read miss. More detailed measures show that nearly all of these are due to *read-modify-write* instructions. Thus, the SB and MB submodels must be adjusted so that, with probability $f_{synch-write}$, a write request itself visits the memory system rather than forking a memory transaction in the SB submodel. Analogously, and with the same probability, the write request forks a memory transaction in the MB submodel rather than contributing directly to processor stall time.

[4]FastILP underpredicts τ for Water because rollbacks of misspeculated loads, triggered by disambiguating stores, are not yet accurately modeled.

[5]The estimated mean residual life equals the second moment of service time divided by 2τ [17].

[6]Note that the standard formula for mean residual life is assumed at all other queues in the model. Since the variance in service time at the bus, memory modules, and other memory system resources is low, the standard MVA approximation can be expected to perform well.

benchmark	config	model IPC	RSIM IPC	% error
Erle	n16, w64	1.38*	1.45	-4.8
	n16, w128	1.95*	1.83	6.6
	n16, w64, slbus	1.04*	1.08	-3.7
	n16, w64, sldir	0.96*	0.94	2.1
	n16, w64, 1GHz	1.03*	0.92	12.0
FFTopt	n16, w64	1.60	1.58	1.2
	n16, w128	2.42	2.29	5.7
	n16, w64, 1GHz	1.16	1.07	8.4
FFT	n16, w64	1.41	1.39	1.4
	n16, w64, slbus	1.10	1.01	8.9
	n16, w128	2.26	2.06	9.7
LUopt	n8, w64	2.59*	2.41	7.4
LU	n8, w64	1.90*	1.91	-0.5
	n8, w64, slbus	1.62*	1.58	2.5
	n8, w128	2.86*	2.93	-2.4
Radix	n8, w64	1.75	1.64	6.7
	n8, w64, sldir	1.41	1.43	-1.3
Water	n8, w64	1.85	1.74	6.3
	n8, w64, slbus	1.75	1.62	8.0
	n8, w128	2.11	2.13	-0.9

Table 7. Model Accuracy for Homogeneous Applications

Investigations of further discrepancies between the predicted throughputs for RSIM and the analytic model suggested two useful modifications in the simulated system: (1) more efficient layout of the data structures in Water (and MP3D) to increase the uniformity of memory access among the processors, and (2) a second bus queue was added to the processor so that requests to the local directory are never blocked behind requests blocked on a full NI buffer. The second modification ensures that system behavior is more commensurate with the model (since the analytic model does not include the blocking behavior of a single bus queue).

After making the changes to the model and the simulated system, we obtained the validation results shown in Table 7. Configurations denoted by *slbus* and *sldir* indicate that the occupancy of the bus (directory) is increased by a factor of three, in order to produce additional contention to stress the model. For the 1GHz configuration, the latencies for the L2 cache, main memory, and network are increased by a factor of two, reflecting a faster processor.

The results show that the model estimates throughput extremely well for the diverse set of applications, predicting application throughputs that range from 0.88 to 2.93 instructions retired per cycle with under 10% relative error. We believe that the model is a reasonably accurate representation of the system under study, although testing the model on further applications and for more system configurations will also increase confidence in its fidelity.

The analytic throughput estimates for Erle, LU, and LU-opt are marked with an asterisk because the estimates include the RSIM-measured pair-wise synchronization delay (i.e., flag spinning time) in τ, rather than explicitly calculating the synchronization delay in the model. Extending the model to compute pairwise synchronization delay is beyond the scope of the paper. The model does accurately compute the effects of synchronization delay for locks and for the global barriers in all of the applications.

Note that, strictly speaking, when the SB submodel is computed for a given $M = k$, a particular processor for which throughput is computed should have k customers while other nodes should perhaps have a time-varying number of customers dictated by the f_M parameters. However, the simpler implementation of the SB model appears to be adequate for the experiments reported in this paper. We will explore this issue further in future work.

6 Applications of the Model

There are many possible applications of the model developed in this paper, ranging from gaining insight into application behavior and current compiler technology to examining architectural design issues. Below we illustrate a few of these applications.

6.1 Insights into Application Behavior

Insight into application behavior can be gained from studying the transaction frequencies shown in tables 3, 4, and 5. Looking at f_M, we observe that the ILP-optimized versions of LU and FFT (LUopt and FFTopt) have significantly greater mass at higher values of M than the original versions. In contrast, the f_M values for Radix and Water reveal considerably less ability to exploit ILP hardware to overlap read memory requests. Erlebacher shows moderate ability for overlap. These observations were also noted in [21].

Regarding the estimated throughput (IPC) of the baseline configuration (w64) for each application, it appears that performance of Erlebacher and of each version of FFT is limited by frequent misses (low τ), whereas the performance of each version of LU is limited by a high probability that read misses must obtain the data from remote memory. The throughput for Water is limited by a high fraction of read misses that require a three hop transaction, and the performance of Radix is limited by the lack of parallelism in the memory system transactions ($f_1 \approx 1$). These observations may suggest future improvements in application designs.

The model and application parameter values can be used to guide the design of future systems and applications. For example, high values of CV_τ indicate a high degree of burstiness, suggesting that the system should provide good buffering capabilities for various types of memory system transactions. The model can also be used to determine the degree to which throughput is slowed down by MSHR blocking. As another example, poor performance and an unusually high fraction of reads to remote memory suggests that the data structures are not laid out efficiently; in fact, it was this type of data that led to the discovery of the load imbalances in Water and MP3D mentioned in Section 5.

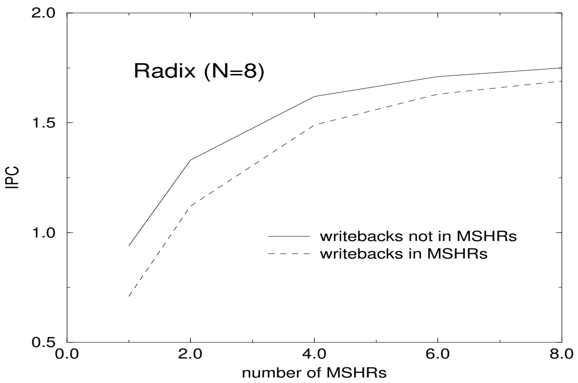

Figure 2. Varying Number of MSHRs (M_{hw})

6.2 Varying the Number and Content of MSHRs

Models are useful for predicting the effects of changes in the system hardware configuration. As an example, we evaluate the impact of changing the number of MSHRs in the system, and the impact of modifying the system such that writebacks occupy MSHRs. Figure 2 shows the impact of varying M_{hw} for Radix, both with and without including writebacks in the MSHRs. Radix was chosen for this evaluation because it exhibits a significant percentage of time (25%) stalled due to write and upgrade requests occupying all MSHRs and it has a high probability of writeback (70%). The results show that performance does not drop significantly until M_{hw} is decreased below 4. The results are qualitatively similar for LU (not shown), an application which frequently uses all of its MSHRs for read requests.

If this is the case for most applications of interest, then system designers could consider smaller sets of MSHRs to reduce both cost and MSHR lookup latency. Similar results have been obtained by Farkas, et al. [8] and Pai, et al. [21] using extensive simulation. Our analytic model (together with FastILP) is capable of obtaining the same results quickly over a wide range of applications.

6.3 Alternative Directory Configurations

The memories and directories at each node in the shared memory architecture may be *coupled* or *decoupled*. In the decoupled case, a transaction that requires just one of these two resources does not occupy both.

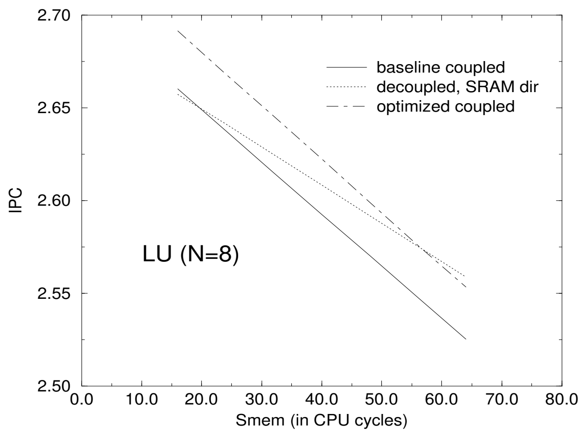

Figure 3. Impact of Directory Configuration

Figure 3 shows the relative performance of the baseline architecture with a coupled memory and directory against two optimized configurations: (1) a decoupled node architecture with an SRAM directory (8-cycle occupancy), and (2) a coupled directory/memory with a fast path to and from the NI. The latter optimization allows remote transactions to bypass the memory bus. (The SGI Origin 2000 effectively provides such a fast path [16].) We easily adapted the model to represent these different architectures.

The results are shown for LU, an application whose read misses are primarily remote; results for FFT (not shown) are qualitatively similar although requests in FFT are more often local. As can be seen in the figure, the coupled architecture with a fast path between the NI and the directory outperforms the other two architectures until memory latency becomes prohibitive for coupled designs. The estimated performance advantage, for LU or other applications, can be traded off against cost considerations.

6.4 Programmable Coherence Controllers

Several recent commercial and research multiprocessor systems [18, 15, 22] have employed programmable coherence controllers to reduce design time and/or support multiple protocols. However, the flexibility and generality of a programmable controller leads to slower coherence protocol execution, which in turn increases controller occupancy and memory latency [11]. The extent to which this degrades application performance has been the subject of several detailed simulation studies [12, 23, 19]. The analytic model can quickly assess the impact of higher controller occupancy.

We evaluate the impact of programmable controllers by modeling a decoupled node architecture with increased directory occupancy (i.e., 80 cycles). Figure 4 plots throughput (IPC) versus network occupancy for two applications (Water and FFT), for this architecture as well as for a coupled node architecture with the baseline memory/directory occupancy (i.e., 40 cycles). The results show that the performance differential is only on the order of 2-7%, depending on network occupancy, for the given applications.

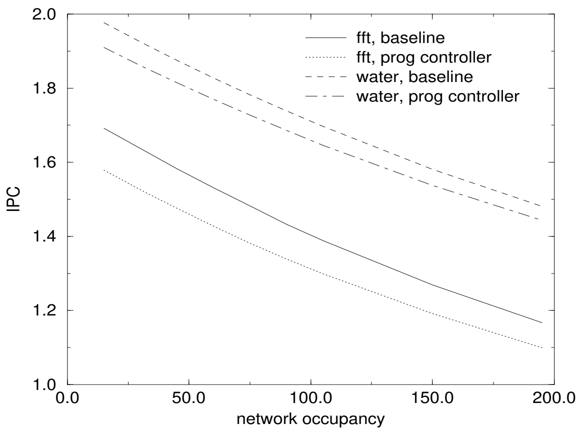

Figure 4. IPC vs Network Occupancy

7 Conclusions

This paper develops and validates a tractable analytic model with a relatively simple and robust set of input parameters for evaluating various types of architectural alternatives for shared-memory systems with processors that aggressively exploit instruction-level parallelism (ILP). The analytical model was compared with detailed simulation for a set of applications and system configurations that have diverse values of the model input parameters. The model yields estimates within 12% of the simulator estimates in seconds, as compared with hours for each simulation result. The study also shows that the analytical model can be used to gain insights into application performance and to evaluate architectural design trade-offs.

This paper also presents FastILP, a simulator for estimating the key ILP input parameters of the model, two orders of magnitude faster than possible with a detailed execution-driven simulator. FastILP achieves this speedup by being trace-driven and by abstracting out a large part of the complexity of both the memory system and the processor. The input parameters generated by FastILP yield calculated throughputs within 12% of those computed with inputs from a detailed execution-driven simulator for all configurations of all but one application studied.

Ongoing research includes extending the model to handle non-homogeneous applications, non-uniform memory access behavior, and slowdowns due to other types of synchronization such as the producer-consumer flags in LU.

One of the key conclusions of our initial validation experiments is that modeling non-homogeneity may be important for more applications than previously thought. Even MP3D, which appears on the surface to be a homogeneous application, has some non-homogeneous behavior that must be captured for the model to be highly accurate. We are also applying the analytic model to new applications and architectural issues. Extending the capabilities of FastILP is also an important topic for further research.

Acknowledgments

We would like to thank Derek Eager for many helpful discussions related to the work in this paper and Aaron Gresch for his work on an early version of the model.

References

[1] S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *Proc. 18th Int'l Symp. on Computer Architecture*, pages 298–308, June 1991.

[2] V. Adve et al. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs . In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

[3] V. Adve and M. Vernon. The Influence of Random Delays on Parallel Task Execution Times. In *Proc. ACM SIGMETRICS*, pages 61–73, May 1993.

[4] D. Albonesi and I. Koren. A Mean Value Analysis Multiprocessor Model Incorporating Superscalar Processors and Latency Tolerating Techniques. *Int'l Journal of Parallel Programming*, 1996.

[5] M. Chiang and G. Sohi. Evaluating Design Choices for Shared Bus Multiprocessors. *IEEE Trans. on Computers*, 41(3):297–317, Mar. 1992.

[6] M. Durbhakula, V. Pai, and S. Adve. Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. Technical Report 9802, Dept. of Elec. and Comp. Engineering, Rice Univ., Apr. 1998.

[7] D. Eager. Private communication, Nov. 1997.

[8] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-System Design Considerations for Dynamically-Scheduled Processors. In *Proc. 24th Int'l Symp. on Computer Architecture*, pages 133–143, June 1997.

[9] P. Heidelberger and K. Trivedi. Analytic Queueing Models for Programs with Internal Concurrency. *IEEE Trans. on Computers*, C-32(1):73–82, Jan. 1982.

[10] P. Heidelberger and K. Trivedi. Queueing Network Models for Parallel Processing with Asynchronous Tasks. *IEEE Trans. on Computers*, C-31(11):1099–1109, Nov. 1982.

[11] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *ASPLOS-VI*, pages 274–285, Oct. 1994.

[12] C. Holt, J. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proc. 23rd Int'l Symp. on Computer Architecture*, pages 134–145, May 1996.

[13] P. Jacobson and E. Lazowska. Analyzing Queueing Networks with Simultaneous Resource Possession. *Communications of the ACM*, 25(2):142–151, Feb. 1982.

[14] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. 8th Int'l Symp. on Computer Architecture*, pages 81–87, May 1981.

[15] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. 21st Int'l Symp. on Computer Architecture*, Apr. 1994.

[16] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. 24th Int'l Symp. on Computer Architecture*, June 1997.

[17] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative System Performance, Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, NJ, May 1984.

[18] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proc. 23rd Int'l Symp. on Computer Architecture*, pages 308–317, May 1996.

[19] M. Michael, A. Nanda, B. Lim, and M. Scott. Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors. In *Proc. 24th Int'l Symp. on Computer Architecture*, pages 219–229, June 1997.

[20] V. Pai, P. Ranganathan, and S. Adve. RSIM Reference Manual. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, Aug. 1997.

[21] V. Pai, P. Ranganathan, and S. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proc. Third Int'l Symp. on High Performance Computer Architecture*, pages 72–83, Feb. 1997.

[22] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. 21st Int'l Symp. on Computer Architecture*, pages 325–337, Apr. 1994.

[23] S. Reinhardt, R. Pfile, and D. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proc. 23rd Int'l Symp. on Computer Architecture*, pages 34–43, May 1996.

[24] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 1992.

[25] D. Sorin et al. A Customized MVA Model for ILP Multiprocessors. Technical Report 1369, Computer Sciences Dept., Univ. of Wisconsin - Madison, Mar. 1998.

[26] M. Vernon, E. Lazowska, and J. Zahorjan. An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols. In *Proc. 15th Int'l Symp. on Computer Architecture*, pages 192–202, 1988.

[27] D. Willick and D. Eager. An Analytical Model of Multistage Interconnection Networks. In *Proc. ACM SIGMETRICS*, pages 192–202, May 1990.

[28] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proc. ACM SIGMETRICS*, May 1996.

[29] S. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Int'l Symp. on Computer Architecture*, pages 24–36, June 1995.