Estimating Performance of

Single Bus, Shared Memory Multiprocessors

Garth Gibson

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

Submitted in partial satisfaction of the requirements of the Masters of Science degree in Computer Science, type II, from the University of California at Berkeley.

Table of Contents

Estimating Performance of Single Bus, Shared Memory Multiprocessors

Garth Gibson

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

Abstract: *Given standard characteristics of processors and memory, we present two simple ways of estimating the performance of shared memory multiprocessors. At the cost of a few simple arithmetic operations, a computer designer can estimate the range of performance using our "4-point bound" model. If more accuracy is required, we show that a one page program can estimate performance within 3% of trace-driven simulation, while reducing software development time, disk space, and CPU time by orders of magnitude. To demonstrate the use of our models, an application to the SPUR multiprocessor design is presented.*

1. Introduction

Multiprocessor computers have long been attractive because of their cost-performance advantages [Fuller76]. Unfortunately, their success has been hampered by the difficulties of parallel programming, particularly when the system interconnection has to be considered explicitly [Deminent82]. Consequently, many researchers and designers look to shared memory for simplifying parallel programming [Baskett86, Bell85].

Faced with designing a memory system for multiple processors, a computer architect must estimate the contention for shared memory. This can be done with hardware monitors, trace-driven simulation, distribution-driven simulation or mathematical models. The most accurate estimate would come from measuring a similar system. However, even when a similar system is available, trace-driven simulation is often preferred, because it allows for the evaluation of alternative designs by repeatable experiments [Smith85]. Because trace-driven simulation represents at least one real workload, it is often preferred over distribution-driven simulation or mathematical models. Finally, either simulation technique usually induces more confidence than mathematical models, because they make fewer simplifying assumptions.

Trace-driven simulation, however, is not without its disadvantages [Clark83,Clark85,Smith85]. Obtaining processor reference traces is no small matter. It usually involves a software project to construct a simulator that will produce traces and it may require modifications to the operating system or microcode. It is typi-

cally difficult to trace system effects such as I/O, interrupts, context switches and interactions between multiple processes. The very act of tracing may affect the system being traced and thus, the trace itself. And once the traces have been obtained, a further software project is required to construct a simulator that will process the traces and mimic a target system to record important events. Since the target system may not exist, there is no simple way to verify the correctness of this simulator. Finally, analyzing traces demands significant CPU time, sometimes measured in units as large as "CPU weeks".

The advantage of mathematical models, such as queueing networks [Allen80, Kleinrock75], is their relatively low cost. By modeling event durations as random variables, these models may be used to explore many design choices in CPU seconds. Rules of thumb provide even cheaper alternatives to simulation, if the elimination of large classes of design choices is critical. At the cost of reduced accuracy, "back of the envelope" calculations based on rules of thumb can be invaluable [Bentley84].

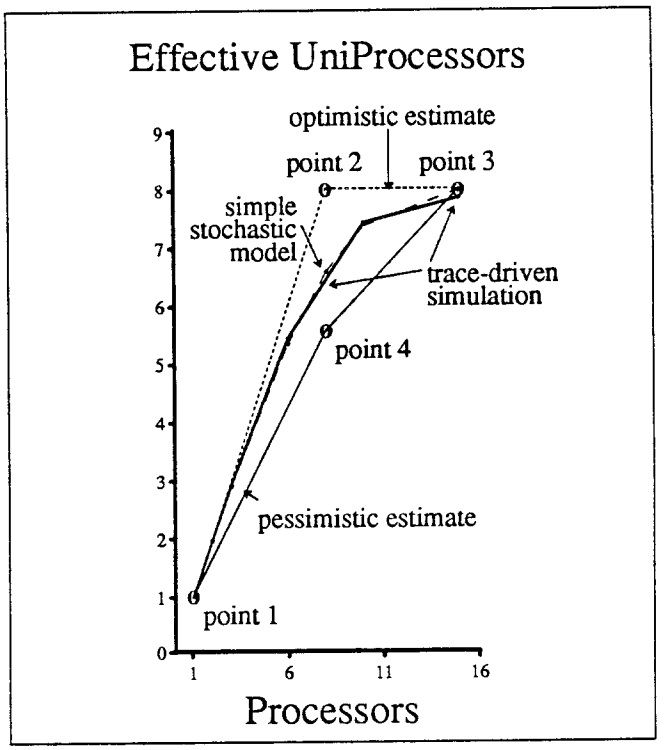

Figure 1

This paper presents a simple stochastic model and a "4-point bound" rule of thumb that predict the potential speedup of shared memory multiprocessors. Both compare well with results obtained by trace-driven simulation. This figure shows speedup, as measured in terms of effective uniprocessors.

In this paper we present two models for predicting potential shared memory multiprocessor performance. Our premise is that mathematic models are acceptable alternatives to simulation if their performance estimates are close to trace-driven simulation estimates regardless of structural differences and simplifying assumptions. Figure 1 demonstrates our models.

(1) The "4-point bound" is a rule of thumb that gives "back of the envelope" calculations for performance bounds. The pessimistic edge of the bound differs from the optimistic edge by no more than 35%.

(2) A simple queuing model that is similar to the "interactive computer system" model of [Allen78], and predicts performance to within 3% of trace-driven simulation estimates in orders of magnitude less CPU time.

The most frequently used performance metric in this paper is *effective uniprocessors* (EU). Effective uniprocessors is a measure of speedup; it is the ratio of time it would take a uniprocessor to process N copies of the workload to the time it takes the N-way multiprocessor to process the workload once per processor.

While our models are applicable to parallel programs, we do not examine parallel programs in this paper. Parallel program interactions, data sharing and synchronization, can slow the rate that useful work is done, so our results predict *potential* speedup. We also exclude I/O traffic from our study, as we assume the impact of I/O traffic on a high speed memory system is small.

Our memory system model is intended for multiprocessors with per processor writeback caches and memory systems constructed around standard backplane buses such as the VME, Multibus-2, NuBus, or Futurebus. These buses do not overlap subsequent accesses and provide fair arbitration (except that the VME arbitration is fair for at most 4 processors). In such a system memory traffic will be dominated by transfers of cache blocks resulting from cache misses (that stall the processor). So in our simple memory system a processor stalls while awaiting memory service; memory services one request at a time; read and write events have constant, equal durations; and arbitration is fair (for example, first come, first serve).

In the next section we introduce some definitions and reference related work. Section 3 details the models, trace-driven simulation and the comparison experiment for estimating effective uniprocessors. The results of this experiment are presented in section 4 and discussed in section 5. In section 6, we present an

analogous development and evaluation of two other performance metrics, bus utilization and average wait for memory service. Section 7 presents a sample application of our simple queueing model on the SPUR multiprocessor [Hill86]. We conclude with a discussion of future work, a summary and acknowledgements in sections 8, 9 and 10. The first appendix provides raw data taken from the trace-driven simulation experiment for the interested reader. In the second appendix we examine the pessimistic estimate of the 4-point bound to determine its relationship to a true lower bound on performance.

2. Definitions and Related Work

Our multiprocessor has N processors, each with an average computation time between successive memory requests, *compute time*[1], of $t_{compute}$. In a system with caches local to each processor, this is the mean time between cache references that require memory service (misses or writeback events) excluding the memory service time. Computer designers can estimate $t_{compute}$ from the processor speed, the rate a processor references its cache, the cache miss ratio, the miss overhead, and the fraction of dirty blocks replaced in a writeback cache. Memory service time, the duration that the bus and memory are occupied servicing a processor's request, is $t_{transfer}$. Computer designers can estimate $t_{transfer}$ from the memory speed, the bus speed, and the cache blocksize. In queueing theory terminology, $t_{compute}$ and $t_{transfer}$ are called $1/\lambda$ and $1/\mu$, respectively.

A further note on our metric, effective uniprocessors, is needed. Since the same workload is being processed on all processors, the ratio of execution times is the reciprocal of the ratio of processor utilizations:

$$\frac{uniprocessor\ execution\ time}{multiprocessor\ execution\ time} = \frac{T_1}{T_N} = \frac{W/T_N}{W/T_1} = \frac{multiprocessor\ utilization}{uniprocessor\ utilization}$$

In this expression, T is the total cycles and W is the total cycles that the processor is busy (fixed by the workload). This gives us another way to measure effective uniprocessors, the ratio of the execution time of N units of a workload on a uniprocessor to a unit per processor on a N-way multiprocessor, when the workload on each processor is identical: N times the ratio of the per processor utilization in the multiprocessor to the uniprocessor utilization.

[1] Our compute time parallels the *think time* in the interactive computer system model [Allen80].

The same optimistic estimate of shared memory multiprocessor performance has been developed many times [Reyling74, Kinney78, Dubois85, Baskett86]. Each of these models is a special case of Kleinrock's *system saturation point*, $N^* = \frac{t_{compute}+t_{transfer}}{t_{transfer}}$ [Kleinrock75]. The optimistic edge of our 4-point bound uses this approach, also known as *bottleneck analysis* [Denning78].

A pessimistic estimate of shared memory multiprocessor performance has also been developed [Reyling74]. In Reyling's analysis, every time a processor requests memory service, it is assumed that the processor must wait for every other processor to access memory. This is similar to our paranoid bound given in the second appendix. In contrast, the pessimistic estimate in our 4-point bound will demonstrate empirically that on average a processor waits for no more than half of the other processors before getting memory service.

Marsan *et al* present performance analyses to compare four slightly different single bus, message passing multiprocessor architectures where shared memory is used to pass variable length messages [Marsan82]. They model two of their architectures with a simple M/M/1//N queueing model [Kleinrock75]. The M/M/1/N has also been used without validation to model contention for a shared I/O bus [Kinney78]. We also use the M/M/1//N queue, but we model a cache-based shared memory system rather than a private memory, message passing system or a shared I/O system. Using exponential distributions (i.e., M/M/1//N) in queueing models instead of non-exponential distributions (i.e., GI/G/1//N) allows substantial simplifications and often accurately estimates the more complex non-exponential model [Buzen77].

Queueing models with more complex structure can often offer more accurate performance estimates, but they usually also cost more to solve. Researchers in queueing theory are investigating approximations for some complex models, to reduce the cost of extracting performance estimates [Chandy78]. This research is primarily concerned with the accuracy of approximations relative to models and does not directly address the applicability of their models as predictors of shared memory multiprocessor performance [Marsan83, Towsley86].

3. Effective Uniprocessors Experiment

In this section we describe our models and trace-driven simulation experiment. In each model the parameters, number of processors (N), memory service time ($t_{transfer}$), and mean computation time between requests to memory ($t_{compute}$) are assumed to be fixed by the model user.

We begin with the 4-point bound. It is built from two curves, an optimistic estimate and a pessimistic estimate. These estimates rely on simple arithmetic only.

3.1. Optimistic Estimate

The 4-point bound's optimistic edge is based on Kleinrock's system saturation point and is similar to several published models [Reyling74, Dubois85, Baskett86]. In this model, processors schedule their requests to avoid waiting for the bus. Since there is never any contention, this model's effective uniprocessors estimates will always be better than what is actually obtained. One way to achieve this behavior is to have a constant compute time, $t_{compute}$, as shown in Figure 2.

To calculate the saturation point, notice that each processor requests $t_{transfer}$ service from memory every $t_{compute}+t_{transfer}$, so the memory is idle for $I = t_{compute}+t_{transfer}-Nt_{transfer}$ between successive requests from a processor. The system saturates when the memory is never idle, $I = 0$. Kleinrock names this point $N^* = \frac{t_{compute}+t_{transfer}}{t_{transfer}}$, the system saturation point.

While $N \leq N^*$ there is no contention, so the multiprocessor utilization will be the same as the uniprocessor utilization and effective uniprocessors is N. Once the system is saturated, we would expect that effective

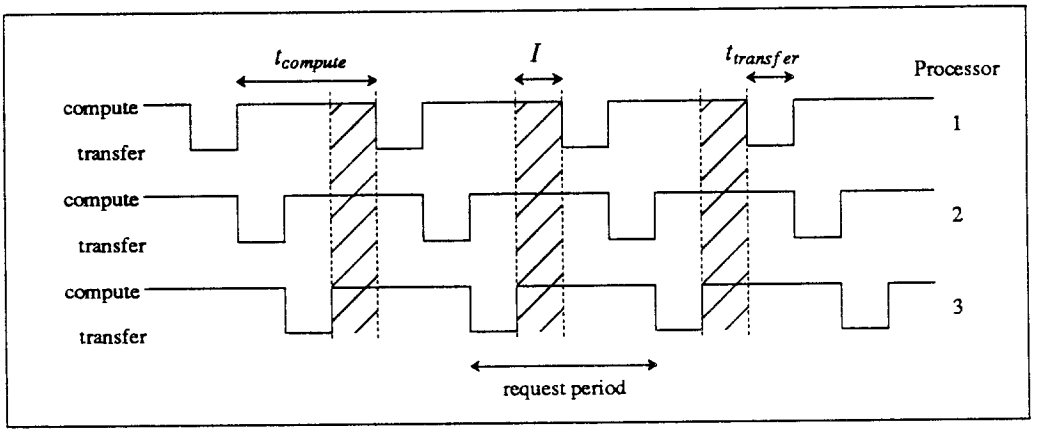

Figure 2

One way to minimize contention is to have each processor request memory precisely at the time its predecessor releases memory. This figure shows a 3 processor system with memory service time $t_{transfer} = 1$ and compute time $t_{compute} = 3$ under optimistic conditions. Because there is no contention, memory is idle (I) 1 out of the 4 cycles between successive requests from a processor. The example in this figure is directly comparable to the pessimistic example in Figure 4.

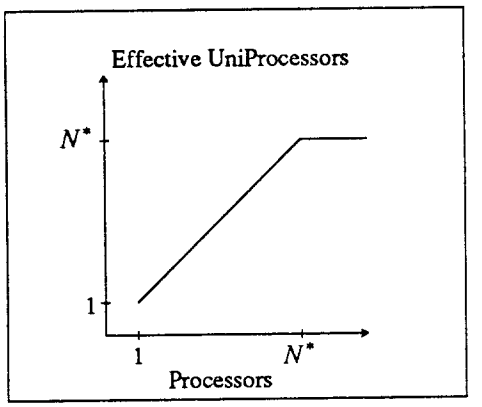

Figure 3

By the optimistic estimate, effective uniprocessors shows linear growth until there are $N^* = \frac{t_{compute}+t_{transfer}}{t_{transfer}}$ *processors. At this point, the memory system is saturated, and effective uniprocessors does not increase as more processors are added.*

uniprocessors should not increase with additional processors. This model verifies our intuition. Each additional processor causes every processor to wait for the additional processors' service times $(N-N^*)t_{transfer}$. So each processor in the multiprocessor is working for $t_{compute}$ out of each $t_{compute}+t_{transfer}+(N-N^*)t_{transfer}$. Effective uniprocessors, while the memory is saturated, is therefore:

$$EU = N\frac{multiprocessor\ utilization}{uniprocessor\ utilization} = \frac{N\ \frac{t_{compute}}{t_{compute}+t_{transfer}+(N-N^*)t_{transfer}}}{\frac{t_{compute}}{t_{compute}+t_{transfer}}} = \frac{t_{compute}+t_{transfer}}{t_{transfer}} = N^*$$

Effective uniprocessors, as shown in Figure 3, is summarized as: $EU = \min\left[\, N, N^* \,\right]$.

3.2. Pessimistic Estimate

The optimistic edge of the 4-point bound minimizes contention for memory. A corresponding pessimistic edge maximizes contention for the same parameter values, N, $t_{transfer}$ and $t_{compute}$ by issuing all memory requests with no intervening computing at the beginning of the execution then doing all the computing. We call this the *paranoid bound.* Unfortunately, it is an unreasonable model. A more reasonable model gathers requests into groups and has all processors begin each group simultaneously by issuing all of the requests then doing the processing for that group. This establishes a family of models based on the number of requests per group where the paranoid bound is the extreme model with all requests in one group. The most optimistic of these models has one request per group and we use it for the lower bound edge of our 4-point bound. We

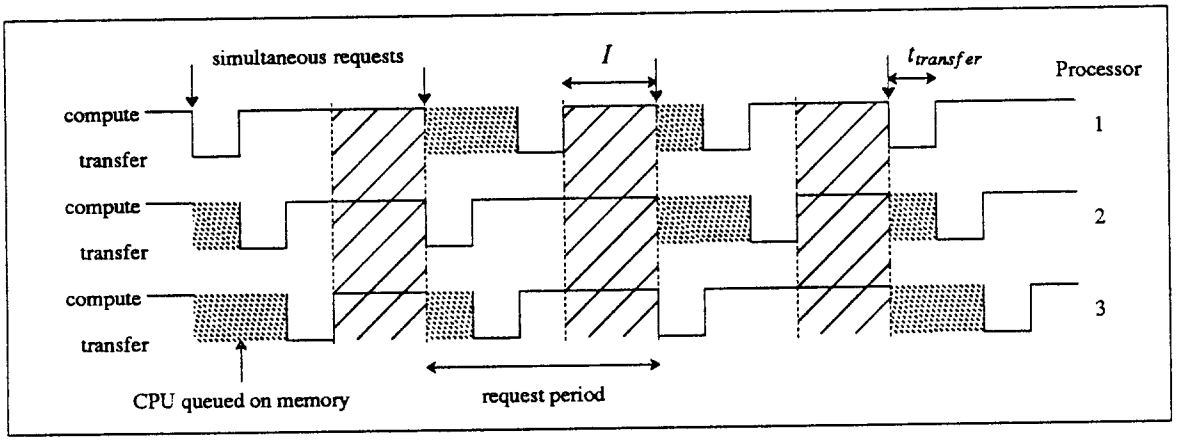

Figure 4

One pessimistic model that has maximum contention occurs if processors have deterministic compute durations such that all processors always request memory simultaneously. In this figure, as in Figure 2, the 3 processors have an average $t_{compute} = 3$ and a constant service time of $t_{transfer} = 1$. But now memory is idle (I) for 2 cycles between successive requests and each processor is taking 25% longer to complete its computation (request period = 5).

examine this family of pessimistic bounds in more depth in the second appendix of this report.

We can maximize contention within a group if all processors request memory simultaneously, as shown in Figure 4. Thus all compute periods must be timed to end together. To ensure that every processor gets the same amount done on average, we use a round robin arbitration scheme (so that $t_{compute}$ is the same for each processor). For example, processor 1 uses memory immediately on its first request, must wait for $2t_{transfer}$ on its second request, then waits for $t_{transfer}$ in its third request period, before using memory immediately again in

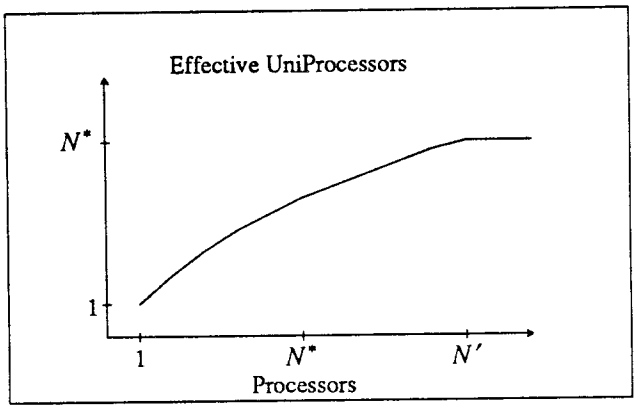

Figure 5

By the pessimistic estimate, effective uniprocessors does not reach the saturation value of N^* until there are N' processors.

its fourth request period.

In Figure 4 we see that over N requests, each processor computes for I, $I+t_{transfer}$, $I+2t_{transfer}$, ... $I+(N-1)t_{transfer}$. So the average compute time between requests, a fixed parameter, is used to determine the idle time, I, as follows:

$$t_{compute} = \frac{1}{N}\sum_{i=0}^{N-1}(I+i\ t_{transfer}) = I + \frac{N-1}{2}\ t_{transfer}\ ,\ so\ \ I = t_{compute} - \frac{N-1}{2}\ t_{transfer}$$

Saturation is reached when memory is never idle, e.g., $t_{compute} = (N-1)t_{transfer}/2$. By solving for N we find the number of processors that first saturate the pessimistic model is $\frac{2t_{compute}+t_{transfer}}{t_{transfer}}$. We name this point N', as shown in Figure 5 relative to N^*.

While $N \le N'$, a request period is $I+Nt_{transfer}$ or, using the value of I determined above, $t_{compute}+(N+1)t_{transfer}/2$. During this time each processor gets $t_{compute}$ work done. So effective uniprocessors, N times the ratio of multiprocessor utilization to uniprocessor utilization, is:

$$EU = \frac{N\frac{t_{compute}}{t_{compute}+(N+1)t_{transfer}/2}}{\frac{t_{compute}}{t_{compute}+t_{transfer}}} = \frac{N\,(t_{compute}+t_{transfer})}{t_{compute}+\frac{N+1}{2}\ t_{transfer}}$$

At N', the pessimistic and optimistic estimates have both reached saturation and have the same effective uniprocessors. For $N > N'$ the requirement to compute an average of $t_{compute}$ between requests prohibits simultaneous requests, so we define the pessimistic estimate to match the optimistic estimate. Therefore, the equation for the pessimistic estimate of effective uniprocessors is:

$$EU = \min\left[\ \frac{N\,(t_{compute}+t_{transfer})}{t_{compute}+\frac{N+1}{2}\ t_{transfer}}, N^*\ \right].$$

3.3. 4-Point Bound

The 4-point bound is a rule of thumb. Its value is its simplicity, so we select four points from the optimistic and pessimistic estimates to enscribe a region that should contain the actual effective uniprocessors performance curve. Figure 6 shows the region and 4 points with the following coordinates (first coordinate is

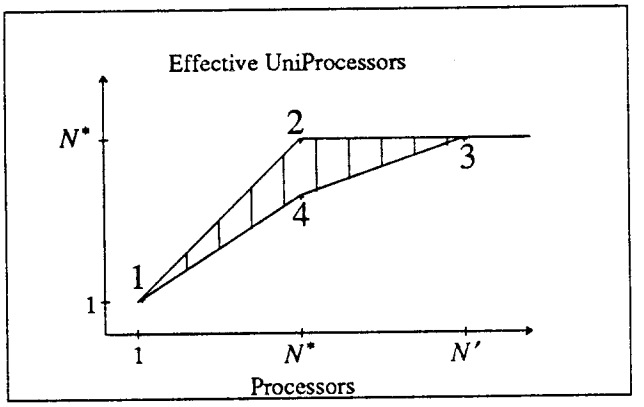

Figure 6

The region formed by joining the 4 points (1,1), (N^*,N^*), (N',N^*), *and* $(N^*,(N^*)^2/(1+3t_{compute}/2t_{transfer}))$ *contains the effective uniprocessors curve for a simple shared bus multiprocessor with N processors, memory service time of* $t_{transfer}$, *and mean computation between memory requests of* $t_{compute}$. $N^* = \frac{t_{compute}+t_{transfer}}{t_{transfer}}$, *and* $N' = \frac{2t_{compute}+t_{transfer}}{t_{transfer}}$.

number of processors, the second, effective uniprocessors):

Point 1: (1,1) is just the uniprocessor performance.

Point 2: $(N^*,N^*) = (\frac{t_{compute}+t_{transfer}}{t_{transfer}}, \frac{t_{compute}+t_{transfer}}{t_{transfer}})$ is the optimistic estimate at its saturation.

Point 3: $(N',N^*) = (\frac{2t_{compute}+t_{transfer}}{t_{transfer}}, \frac{t_{compute}+t_{transfer}}{t_{transfer}})$ is the pessimistic estimate at its saturation.

Point 4: is the pessimistic estimate at N^*,

$$(N^*, \frac{(N^*)^2}{1+\frac{3t_{compute}}{2t_{transfer}}}) = (\frac{t_{compute}+t_{transfer}}{t_{transfer}}, \frac{(t_{compute}+t_{transfer})^2}{t_{transfer}^2+3t_{transfer}t_{compute}/2}) = (N^*, N^* \frac{1+\frac{t_{transfer}}{t_{compute}}}{\frac{3}{2}+\frac{t_{transfer}}{t_{compute}}})$$

Since both $t_{compute}$ and $t_{transfer}$ are positive, $0 < \frac{t_{transfer}}{t_{compute}} < \infty$, so the EU value at point 4 is between $\frac{2}{3}N^*$ and N^*. This allows the further pessimistic simplification,

Point 4: $(N^*, \frac{2}{3}N^*)$

Once again, the attraction of the 4-point bound is that we can estimate shared memory multiprocessor performance at the cost of just a half dozen additions, multiplications and divisions.

3.4. A Simple Queueing Model

In this section we discuss a very simple queueing model of a shared memory multiprocessor with a finite population of processors. In queueing theory this model is called the M/M/1//N queue [Kleinrock75]. It has been applied to computers as the interactive computer system model for predicting response time in a timesharing system [Allen78] and for predicting performance in a message passing multiprocessor [Marsan82]. A more precise model called the *machine interference* [Saaty61] (or machine repairman) model, M/G/1//N in the terminology above, could be used in place of the M/M/1//N. However, the M/M/1//N is computationally simpler and our results will show that its estimates have more precision than we would expect of its parameters.

In the 4-point bound, memory service time and processor compute time are deterministic. In our simple queueing model, memory service time and processor compute time take stochastic values; that is, their values are selected based on probabilities. Service time is distributed as an exponential random variable with mean $t_{transfer}$. Compute time is distributed as an exponential random variable with mean $t_{compute}$. The memory module uses a first come, first serve method for processing requests and processors that are waiting for memory service are idle. The solution to this model is given in terms of the proportion of time that there are n processors waiting for service, p_n, [Kleinrock75].

$$p_n = \frac{u_n}{\sum_{i=0}^{N} u_i} \quad where \quad u_n = \frac{N!}{(N-n)!}\left[\frac{t_{transfer}}{t_{compute}}\right]^n$$

To compute effective uniprocessors from p_n, first compute the average number of processors waiting on memory, L_N,

$$L_N = \sum_{n=1}^{N} n\, p_n .$$

Effective uniprocessors is the ratio of N times the per processor utilization in the multiprocessor to the uniprocessor utilization. However, N times the per processor utilization in the multiprocessor is the number of active processors, $N-L_N$, and the uniprocessor utilization is the number of active processors in a uniprocessor, $1-L_1$. Thus the M/M/1//N queue predicts an effective uniprocessors of:

$$EU = (N-L_N)/(1-L_1) = (N-L_N)\frac{t_{compute}+t_{transfer}}{t_{compute}}$$

```
input t_compute , t_transfer , N

sum = 1; L = 0; temp = 1;
for( i=1; i <= N; i++ ) {
      temp = (N-i+1) * temp * t_transfer / t_compute ;
      sum = sum + temp;
      L = L + i * temp;
}
L = L / sum;

printf( "Effective Uniprocessors = %f\n", ( 1 + t_transfer / t_compute ) * ( N - L ) );
printf( "Bus Utilization = %f\n", 1 - 1 / sum );
printf( "Average Wait on Memory = %f\n", L * t_compute / ( N - L ) );
```

Figure 7

Simple code (suitable for a programmable hand calculator) to compute the number of effective uniprocessors, bus utilization and average wait (stall) during a request for memory service using the simple queueing model given the number of processors (N), mean memory service time ($t_{transfer}$) and mean time from completion of a request to generation of the next request by a processor ($t_{compute}$). This estimates effective uniprocessors, bus utilization and average wait within 3%, 4% and 10%, respectively, of trace-driven simulation.

Figure 7 shows a small program that calculates effective uniprocessors and other performance metrics for this simple queueing model.

3.5. Trace-Driven Simulation

To evaluate the accuracy and cost of the simple queueing model and the 4-point bound, a trace-driven simulation program was written. It processes memory request traces and simulates a simple, single bus and memory. It handles one processor request at a time and allocates the same duration for read and write events.

Table 1: Processor Reference Trace Sources

Name	Architecture	# References	Description
OPSYS	IBM 370	1,000,000	System reference trace of a collection of users calls into the MVS operating system [Smith85,Wood86]
DATABASE	M68000	12,582,912	User and system reference trace of SYNAPSE database machine executing a synthetic load composed of the database query and update benchmark TP1
LISPCOMP2	SPUR	20,000,000	User reference trace of SPUR LISP compilation (simulated) of a portion of itself (register allocation) [Taylor86]

The workload is homogeneous; each processor executes the same trace starting from different points and wrapping around to its place of origin.[2] The traces were generated by the application of a cache simulator program, DineroIII [Hill83], to the set of three processor reference traces described in Table 1. Notice that in this table a reference means that a processor requests a datum from its cache rather than directly from memory. If the cache does not have that datum, the cache issues a memory request. We call the post-cache trace a "memory request trace".

Although a queuing-novice might expect that traces from deterministic machines would be anything but stochastic, queuing theory provides the basis for expecting our experiment to agree with the stochastic models. First, the processor reference pattern is "thinned" by the cache to a series of misses that occur more randomly than the original processor references. Second, by wrapping a long trace around on itself and beginning at different points, the sequences can be approximated as independent. Third, merging independent random sequences tends to make the merged sequence appear Poisson (in agreement with our simple model).

Table 2 shows eight cache configurations, chosen to generate a varied selection of memory traffic, that were applied to each of the three traces. The resulting memory request traces, described in Table 3, vary in length from a few thousand requests to several hundred thousand requests. The standard metric of cache performance, the miss ratio, is the ratio of the number of cache misses to the number of processor references into the cache. It does not include writeback events (a modified block replaced in the cache must be written back to memory). The *request ratio*, is a more appropriate metric for estimating memory traffic. It is the ratio of the number of requests for memory service by the cache to the number of processor references into the cache.

Table 2: Cache Configurations

Number	Size (B)	Organization	BlockSize (B)	Associativity	Replacement
1	512K	Mixed	128	4	LRU
2	256K+256K	I & D Split	32	4	LRU
3	512K	Mixed	8	4	LRU
4	128K	Mixed	32	1	-
5	32K	Mixed	128	4	LRU
6	128K	Mixed	4	4	LRU
7	64K+64K	I & D Split	8	2	LRU
8	16K	Mixed	256	4	RANDOM

[2] We use this "wrap around" approach so that we can be sure that the same amount of work, W in the earlier discussion of our effective uniprocessors metric, is done by each processor, while avoiding unrealistically symmetric request sequences. This is similar to techniques for spreading memory module requests derived from a single address trace [Baskett76].

Table 3: Memory Request Traces (Post Cache Simulation)						
Trace Name	Miss Ratio	Request Ratio	Number of Reqs	Avg References Per Request	Coefficient of Variation	Output Size
OPSYS.1	.003	.004	3,638	275.9	6.5	.08 MB
OPSYS.2	.007	.011	10,472	96.5	6.0	.21 MB
OPSYS.3	.017	.024	23,506	43.5	11.7	.45 MB
OPSYS.4	.016	.023	22,579	45.3	3.2	.46 MB
OPSYS.5	.022	.028	27,605	37.2	2.3	.59 MB
OPSYS.6	.031	.045	44,534	23.5	11.0	.84 MB
OPSYS.7	.021	.031	30,618	33.7	5.8	.58 MB
OPSYS.8	.043	.053	53,392	19.7	2.0	1.12 MB
DATABASE.1	.001	.001	17,164	734.1	4.5	.40 MB
DATABASE.2	.004	.005	65,858	192.1	6.7	1.42 MB
DATABASE.3	.006	.009	115,086	110.3	11.5	2.26 MB
DATABASE.4	.011	.013	167,092	76.3	3.7	3.57 MB
DATABASE.5	.008	.009	116,468	109.0	3.5	2.62 MB
DATABASE.6	.034	.043	535,601	24.5	12.8	10.37 MB
DATABASE.7	.025	.028	345,985	37.4	9.2	6.72 MB
DATABASE.8	.012	.015	190,766	67.0	3.6	4.22 MB
LISPCOMP2.1	.002	.002	45,115	444.3	2.6	1.04 MB
LISPCOMP2.2	.006	.009	176,952	114.0	3.8	3.76 MB
LISPCOMP2.3	.015	.025	494,821	41.4	6.8	9.65 MB
LISPCOMP2.4	.019	.024	485,684	42.2	3.7	10.20 MB
LISPCOMP2.5	.018	.020	404,397	50.5	4.1	8.91 MB
LISPCOMP2.6	.028	.039	785,815	26.5	6.6	15.28 MB
LISPCOMP2.7	.029	.041	827,717	29.0	4.0	16.13 MB
LISPCOMP2.8	.032	.036	715,431	29.0	4.0	15.59 MB
Total			5,706,296			116.4 MB

Our comparison of trace-driven simulation to the models uses three different relative speed configurations. As shown in Table 4, fixed times between processor references of 100 nsec and 200 nsec was used. Also, memory speeds were varied by considering memory that takes the same length of time to get each (32 bit) word of a packet (cache block) and memory that provides the second and subsequent words of a packet faster than the first word.

Table 4: Processor and Memory Latencies (nsec)			
Configuration	Processor Inter-reference	Initial Word Latency	Subsequent Word Latency
A	100	400	100
B	100	400	400
C	200	400	100

The time unit need not be nanoseconds, as long as all parameters are evaluated in terms of the same time unit.

In the trace-driven simulator, we calculate a compute period by multiplying the number of references since last memory request by the average time between processor references. For the purpose of computing $t_{compute}$ for the models, the appropriate entries in column 5[3] of Table 3 and column 2 of Table 4 are multiplied together.

Table 3 reports the coefficient of variation on the processor references per memory request. The coefficient of variation is a measure of the regularity of the request sequence and its value indicates a similarity to stochastic distributions [Allen80]. If the coefficient of variation is near 1 then modeling compute time as an exponential random variable would be natural, but this is not the case for our data. Instead the request sequence has a considerably more skewed distribution. Since our simple queueing model assumes compute time is exponential, we might suspect that its performance estimates may be inaccurate. However, memory service time in the trace-driven simulator is constant (it has a coefficient of variation of 0) and queuing theory indicates that a highly regular service distribution will tend to compensate for a highly irregular interarrival distribution [Marshall68]. Our results will show this to be valid for our data.

4. Results

For each memory request trace (3 processor reference traces times 8 cache configurations) and each relative speed configuration (A, B and C) trace-driven simulation was performed for a uniprocessor, 2-, 3-, 6-, 10-, and 15-processor system. Simulation consumed 1 CPU day of a VAX 8650 plus 1 CPU week of a SUN 3 (M68020). Based on the relative speed of these machines for this program[4], this is equivalent to 592 CPU ***hours*** of a SUN 3 as shown in Table 5. The simple queueing model was then run for each of the 432 data points. The complete set of performance estimates were generated by the simple queueing model in 75 CPU

Table 5: Resource Comparison

Resource	Simple Model	Simulation	Ratio
Code Size (lines)	50	2500	1/50
Size of Input Data	.011 MB	116 MB	1/10545
Execution Time (CPUhours)	.021	592	1/28190

[3] The entries in this field have had 1 added to them to account for restarting the instruction causing the request in the trace-driven simulator.

[4] For the bus simulating program that compiled the results we present, a VAX 8650 was 17 times faster than a SUN 3 without floating point assist.

seconds on a SUN 3. However, it should be noted that the trace-driven cache simulation that we used to generate our memory request traces (and $t_{compute}$) consumed approximately 33 CPU hours on a SUN 3.

$$MRE = \max_{1,15} \left| \frac{sim - model}{sim} \right|$$

$$ARE = \frac{1}{14} \sum_{1}^{15} \left| \frac{sim - model}{sim} \right|$$

To evaluate the accuracy of the simple queueing model, Table 6 presents two measures of error: the maximum relative error, MRE, and the average relative error, ARE. The maximum relative error is the largest ratio of the difference between the simulation and model estimate curves to the simulation value. The average relative error is the average value of this ratio. For the 72 data points we have explored, the average relative error in effective uniprocessors is uniformly ≤ 3%, and the maximum relative error is uniformly ≤ 6%.

In Figure 8, 3 sample comparisons of the 72 combinations are shown. These were selected to show good, medium and poor performance. On each graph, the optimistic (topmost line without symbols) and

Table 6: Accuracy of Simple Queueing Model's Effective Uniprocessors Estimates

	Maximum Relative Error			Average Relative Error		
Trace.cache	A	B	C	A	B	C
OPSYS.1	6%	4%	1%	3%	2%	0%
OPSYS.2	4%	5%	1%	2%	2%	0%
OPSYS.3	6%	6%	3%	3%	3%	1%
OPSYS.4	2%	3%	3%	1%	1%	2%
OPSYS.5	2%	1%	2%	1%	0%	1%
OPSYS.6	3%	3%	3%	1%	1%	2%
OPSYS.7	2%	2%	4%	1%	1%	1%
OPSYS.8	1%	0%	2%	1%	0%	1%
DATABASE.1	1%	2%	0%	1%	1%	0%
DATABASE.2	2%	2%	1%	1%	1%	0%
DATABASE.3	2%	4%	0%	1%	2%	0%
DATABASE.4	3%	3%	2%	1%	1%	1%
DATABASE.5	3%	3%	3%	1%	1%	1%
DATABASE.6	3%	3%	1%	2%	2%	1%
DATABASE.7	2%	2%	2%	1%	1%	1%
DATABASE.8	3%	1%	3%	1%	0%	1%
LISPCOMP2.1	2%	2%	1%	1%	1%	0%
LISPCOMP2.2	1%	1%	1%	1%	1%	1%
LISPCOMP2.3	2%	2%	1%	1%	1%	1%
LISPCOMP2.4	2%	2%	2%	1%	1%	1%
LISPCOMP2.5	2%	1%	2%	1%	0%	1%
LISPCOMP2.6	3%	3%	2%	1%	1%	0%
LISPCOMP2.7	3%	3%	3%	2%	2%	1%
LISPCOMP2.8	1%	0%	1%	1%	0%	2%

pessimistic (lower line without symbols) estimates are shown with the simulation (data points marked by X) and simple queueing model (data points marked by circles) curves. The good news is that it is difficult to distinguish the model from the simulation.

5. Discussion

We found the accuracy of the simple queueing model better than we sought, for we believe most computer designers are satisfied by 10% accuracy. It appears that the exponential distribution assumptions for compute time and service time are affecting the effective uniprocessor performance measure even less than Buzen's results suggest [Buzen77]. The model predicts effective uniprocessors within 3% of our trace-driven simulation and requires 1/30,000 the CPU cost and 1/10,000 the storage cost. The traces could have been shorter, inducing savings in CPU time and storage space, but longer traces are generally preferred [Clark85]. We believe that for most traces that are "long enough" to be interesting, the savings of the model will be at

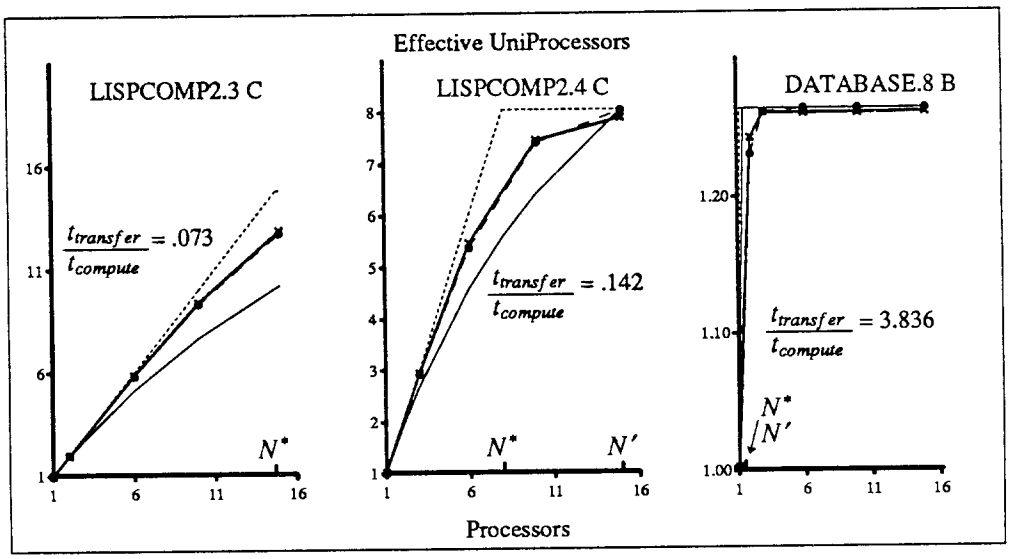

Figure 8

These three curves show the 4-point bound, simple queueing model and bus simulation predictions for effective uniprocessors across the spectrum of performance. The first graph shows very good growth; N^* is not reached until about 15 processors. The second graph shows a medium growth reaching N' by 15 processors and the third graph shows very poor growth, N^* and N' occurring at less than 2 processors. Notice that the pessimistic estimate exceeds trace-driven simulation and the simple queueing model near N', as in the second and third graph. When the performance is extremely poor, as in the third graph, it can happen that the pessimistic estimate is never pessimistic. See the second appendix for more discussion of pessimistic estimates.

least three orders of magnitude. Such time savings facilitate the analysis of a greater amount of data in the design process. In Section 7, we present an example application of the simple queueing model for quickly evaluating potential system performance of large benchmarks in the SPUR multiprocessor. The results of this application, shown in Figure 9, would have taken over five SUN 3 CPU days of SPUR trace-driven simulation instead of the 18 SUN 3 CPU seconds needed by the simple queueing model.

The first appendix reports raw data from the simulations. From this we can see that different processor reference traces, such as DATABASE.5A and LISPCOMP2.5A, have quite large differences in effective uniprocessors: 3.9 vs 2.4 at 6 processors and 4.0 vs 2.4 at 10 processors. Given this sensitivity to processor reference trace, we believe that the difference between trace-driven simulation and queueing model performance estimates is "in the noise". That is, given values for cache miss ratio, fraction of misses replacing

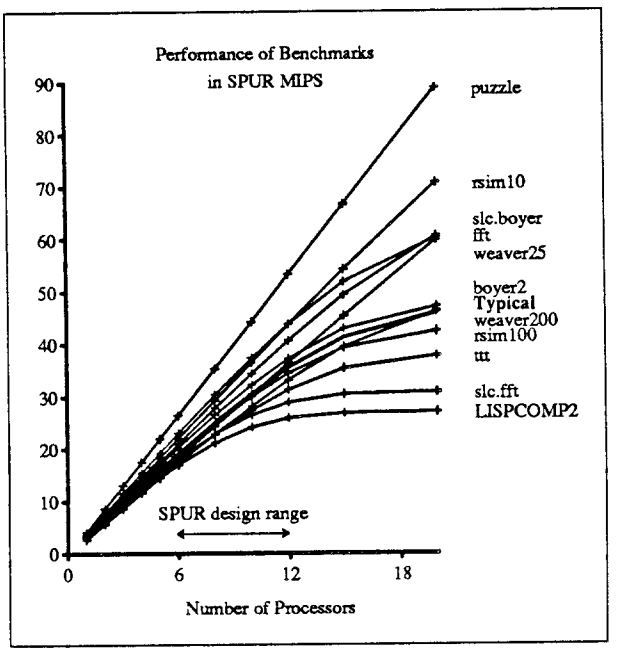

Figure 9

Given estimates of the SPUR uniprocessor characteristics derived from a set of LISP benchmark programs [Taylor86], we have spent 18 SUN 3 CPU ***seconds*** *to derive the equivalent of over 5 SUN 3 CPU* ***days*** *of trace-driven simulation. The SPUR system performance is reported in aggregate SPUR MIPS assuming that all processors have the same average compute time, $t_{compute}$, as the uniprocessor benchmark. Multiprocessor MIPS are computed as the product of effective uniprocessors and uniprocessor MIPS. Uniprocessor MIPS are computed as the average number of* ***instructions*** *executed between memory requests divided by ($t_{compute}+t_{transfer}$), assuming a 150 nsec processor cycle time. See Section 7 for more details.*

dirty blocks, cache reference rate, and memory transfer times, a computer designer can easily calculate the appropriate $t_{compute}$ and $t_{transfer}$ and estimate system performance using our simple queueing model. The difference in effective uniprocessors between one estimate and another based on a different benchmark program will often be much larger than the 3% error in the model.

The 4-point bound brackets the trace-driven simulation estimates quite well except near pessimistic saturation (N'). At N', the pessimistic model consumes 100% of the bus bandwidth because it is a deterministic model. But simulation and the queueing model will not consume 100% of the bus bandwidth because their requests are less orderly. So near N' the pessimistic estimate crosses the simulation. Empirically, however, we find that simulation approaches saturation at N', so it seems that the 4-point region remains a good bound. In the second appendix, the relationship between the pessimistic estimate and a lower bound is discussed in more detail.

Note that the simple queueing model's effective uniprocessors prediction is expressed entirely in terms of the *ratio* of $t_{transfer}$ to $t_{compute}$. The 4-point bound can be reorganized so that it too is expressed entirely in terms of this ratio. This agrees with Patel's study of multiprocessors with delta network or crossbar interconnects [Patel82]. In queueing theory, this ratio is usually referred to as ρ.

This ratio, $\frac{t_{transfer}}{t_{compute}}$, is perhaps the simplest performance indicator of all. Borrowing from the pessimistic estimate, suppose a computer design team wants to build a shared memory, single bus multiprocessor with the performance of at least K uniprocessors each with efficiency at least α (so that at most K/α processors are needed). They will succeed if they vary one or both of the memory system speed or the processor request rate for memory so that $\frac{t_{compute}}{t_{transfer}} \geq \frac{K/2-(1-\alpha/2)}{1-\alpha}$. For example, if they choose 80% efficiency, they must achieve $\frac{t_{compute}}{t_{transfer}} \geq 2.5K - 3$.

6. Additional Metrics Explored

In the preceding sections, our analysis of shared memory multiprocessor performance has focussed on the effective uniprocessors metric. But effective uniprocessors is not the whole story in multiprocessor performance. Because effective uniprocessors is a ratio between two speeds, both affected by design changes, it is often the case that slowing down the processor will straighten the effective uniprocessors curve. For example,

if the SPUR processor design team generates 100 nsec per cycle processors, then the curves in Figure 9 saturate more quickly, but this faster system will achieve a higher total MIPS than the 150 nsec cycle system unless both are saturated. Thus other measures of performance are needed to fully evaluate a design. Two other measures, bus utilization (BU) and average wait on memory (AW), were also evaluated in this experiment. This section presents the equivalent 4-point bound and simple queueing models for these two metrics.

6.1. Optimistic Estimates

The optimistic estimate for effective uniprocessors presented in Section 3.1 is recast for bus utilization and average wait in this section. Recalling Figure 2, in each request period the bus is busy for $Nt_{transfer}$ and the request period lasts for $t_{transfer}+t_{compute}$. So when the bus is not saturated,

$$BU = Nt_{transfer}/(t_{transfer}+t_{compute}) = N/N^*.$$

Because this model does not experience contention until the bus is saturated, the average wait is $t_{transfer}$ while $N \leq N^*$. Once saturated, each processor waits for the additional processors' service times, $(N-N^*)t_{transfer}$, before accessing memory. So the average wait is

$$AW = t_{transfer}+(N-N^*)t_{transfer} = Nt_{transfer}-t_{compute}.$$

The equations for BU and AW are

$$BU = \min\left[1.0, \frac{N}{N^*}\right] \quad \text{and} \quad AW = \max\left[t_{transfer}, Nt_{transfer}-t_{compute}\right].$$

6.2. Pessimistic Estimates

Recall the pessimistic estimate for effective uniprocessors given in Section 3.2. In Figure 4, request periods are of length $Nt_{transfer}+I$ where $I = t_{compute}-(N-1)t_{transfer}/2$. So while the bus is not saturated, $BU = Nt_{transfer}/(t_{compute}+(N+1)t_{transfer}/2)$. Notice that at $N = N'$, the bus saturates.

The wait for memory service depends on the order of service in this model, but after N requests each processor has waited for $t_{transfer}$, $2t_{transfer}$, $\cdots$ $Nt_{transfer}$. Averaging, as we did in Section 3.2, $AW = (N+1)t_{transfer}/2$.

Recall that the pessimistic model cannot be constructed when $N > N'$. In this region we define the pessimistic estimate to agree with the optimistic estimate. The equations for pessimistic BU and AW are

$$BU = \min\left[1.0, \frac{Nt_{transfer}}{t_{compute}+(N+1)t_{transfer}/2}\right] \quad \text{and} \quad AW = \max\left[\frac{(N+1)t_{transfer}}{2}, Nt_{transfer}-t_{compute}\right].$$

6.3. 4-Point Bounds

We can construct regions described by 4 points for the bus utilization and average wait on memory metrics analogous to the 4-point bound for effective uniprocessors given in Section 3.3. For bus utilization, the 4 points are:

Point 1: $(1, \frac{1}{N^*}) = (1, \frac{t_{transfer}}{t_{transfer}+t_{compute}})$

Point 2: $(N^*, 1.0) = (\frac{t_{transfer}+t_{compute}}{t_{transfer}}, 1.0)$

Point 3: $(N', 1.0) = (\frac{t_{transfer}+2t_{compute}}{t_{transfer}}, 1.0)$

Point 4: $(N^*, \frac{t_{transfer}+t_{compute}}{t_{transfer}+1.5t_{compute}}) = (\frac{t_{transfer}+t_{compute}}{t_{transfer}}, \frac{t_{transfer}+t_{compute}}{t_{transfer}+1.5t_{compute}})$

As we did in Section 3.3, the bus utilization at point 4 is always greater than or equal to 2/3, so we can simplify further with:

Point 4: $(N^*, 2/3)$

Examining the the bus utilization equation or its 4-point bound, we see that in these simple estimates, bus utilization is directly proportional to effective uniprocessors. That is, the graphs are identical and the ordinate axis is simply different by a factor of N^*.

For average wait for memory, the 4 points are:

Point 1: $(1, t_{transfer})$

Point 2: $(N^*, t_{transfer}) = (\frac{t_{transfer}+t_{compute}}{t_{transfer}}, t_{transfer})$

Point 3: $(N', t_{transfer}+t_{compute}) = (\frac{t_{transfer}+2t_{compute}}{t_{transfer}}, t_{transfer}+t_{compute})$

Point 4: $(N^*, t_{transfer}+.5t_{compute}) = (\frac{t_{transfer}+t_{compute}}{t_{transfer}}, t_{transfer}+.5t_{compute})$

Figure 10 shows the 4-point bounds on bus utilization and average wait for memory.

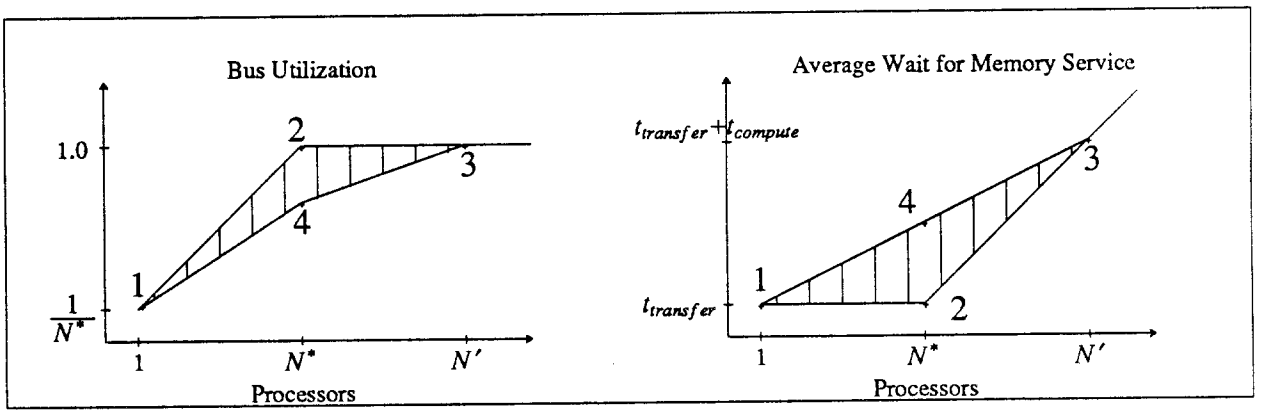

Figure 10

The region formed by joining the four points $(1,\frac{1}{N^*})$, $(N^*,1.0)$, $(N',1.0)$, $(N^*,\frac{t_{transfer}+t_{compute}}{t_{transfer}+1.5t_{compute}})$, *for bus utilization, and the 4 points* $(1,t_{transfer})$, $(N^*,t_{transfer})$, $(N',t_{transfer})$, $(N^*,t_{transfer}+.5t_{compute})$, *for average wait for memory, contains the curves for these two shared memory multiprocessor performance metrics.*

6.4. Simple Queueing Models

The simple queueing model given in Section 3.4 for estimating effective uniprocessors is also applicable to bus utilization and average wait for memory metrics. In Figure 7, code for the simple queueing model shows computation for these two metrics.

The first, bus utilization is the proportion of time that the bus is active. This is 1 minus the proportion of time that there are no processors waiting for memory. In the terminology of Section 3.4, p_0 is the proportion of time that there are no processors waiting for memory. Then the bus utilization must be $1-p_0$.

To derive the average wait for memory service, we use Little's Law [Kleinrock75], $L_N = \lambda\omega$, where λ is the average rate of arrival of requests at the bus and ω is the average wait for memory service. To compute λ notice that while there are n processors waiting for memory, there are $(N-n)$ processors computing. The processors that are computing each generate requests at the rate of $1/t_{compute}$. Using the assumption that compute time is exponential, the rate of requests for memory is the sum of the individual rates of active processors, $(N-n)/t_{compute}$. So the average rate of arrival of requests at the bus, λ, is:

$$\lambda = \sum_{n=0}^{N} \frac{(N-n)}{t_{compute}} p_n = \frac{N}{t_{compute}} \sum_{n=0}^{N} p_n - \frac{1}{t_{compute}} \sum_{n=0}^{N} n\, p_n = \frac{N}{t_{compute}}(1) - \frac{1}{t_{compute}}(L_N) = \frac{(N-L_N)}{t_{compute}}$$

So by Little's Law, $AW = \omega = \frac{L_N}{\lambda} = \frac{L_N\, t_{compute}}{N-L_N}$.

6.5. Trace-Driven Simulation Results

The bus utilization and average wait for memory estimates from the simple queueing model were compared to trace-driven simulation in the experiment described in Section 3.5. Accuracy is measured by the maximum relative error and average relative error as defined in Section 4 and is shown in Table 7.

The results for bus utilization are very similar to those for effective uniprocessors; the average relative error is uniformly $\leq 4\%$ and the maximum relative error is uniformly $\leq 5\%$. For the average wait for memory service we see larger errors; the average relative error is as high as 10% and the maximum relative error is as high as 18%. This occurs because the average wait for memory metric is more sensitive to approximations of the interrequest and service distributions than effective uniprocessors or bus utilization (which are both utilization metrics) [Buzen77]. For this reason, the application to the SPUR multiprocessor in Section 7 derives estimates for SPUR MIPS from effective uniprocessors even though it may be more intuitive to use average wait for memory service.

7. Example: Application to the SPUR Multiprocessor Design

7.1. SPUR Multiprocessor Performance Estimation

SPUR is a shared memory multiprocessor workstation intended as a parallel processing testbed system [Hill86]. It uses a modified NuBus backplane (nominal bus cycle time of 100 nsecs) [Gibson85] and standard memory and peripherals. Its 4th generation Berkeley RISC processor has been designed and simulated at the register transfer level [Taylor86]. On top of this simulator, a LISP compiler and LISP system have been benchmarked (see Table 9). The simulator executes at the rate of 20,000 SPUR instructions (about 25,000 memory hierarchy references) per VAX 8650 CPU second, so benchmarking is an expensive investment. As shown in Table 9, a number of programs were examined (for reasons other than for this study), providing data that we can use to estimate potential multiprocessor performance.

The SPUR processor executes one instruction per cycle unless stalled. It is stalled for 1 cycle by a store operation, for 2 cycles whenever the on-chip instruction buffer does not contain a fetched instruction, for 1 cycle whenever a load or store operation contends with an instruction buffer miss at the second level cache, and if floating point overlap is turned off, during floating point operations. It is also stalled whenever the

Table 7: Accuracy in Bus Utilization Metric						
	Maximum Relative Error			Average Relative Error		
Trace.cache	A	B	C	A	B	C
OPSYS.1	2%	4%	5%	1%	1%	2%
OPSYS.2	1%	3%	4%	0%	1%	1%
OPSYS.3	3%	4%	2%	2%	2%	1%
OPSYS.4	4%	3%	4%	1%	1%	3%
OPSYS.5	4%	1%	4%	1%	0%	1%
OPSYS.6	4%	4%	4%	2%	2%	3%
OPSYS.7	4%	4%	3%	2%	1%	3%
OPSYS.8	2%	0%	4%	0%	0%	1%
DATABASE.1	1%	2%	0%	1%	1%	0%
DATABASE.2	2%	2%	2%	1%	1%	1%
DATABASE.3	1%	1%	1%	0%	1%	1%
DATABASE.4	4%	3%	3%	2%	1%	2%
DATABASE.5	3%	3%	4%	1%	1%	2%
DATABASE.6	4%	4%	5%	2%	2%	4%
DATABASE.7	4%	3%	4%	2%	2%	3%
DATABASE.8	3%	1%	4%	1%	0%	1%
LISPCOMP2.1	2%	2%	1%	1%	1%	1%
LISPCOMP2.2	2%	2%	2%	1%	1%	1%
LISPCOMP2.3	3%	2%	3%	2%	1%	3%
LISPCOMP2.4	3%	3%	4%	1%	1%	3%
LISPCOMP2.5	3%	1%	2%	1%	0%	1%
LISPCOMP2.6	4%	4%	4%	2%	2%	3%
LISPCOMP2.7	4%	3%	4%	2%	1%	3%
LISPCOMP2.8	2%	0%	4%	0%	0%	1%
Accuracy in Average Wait for Memory Metric						
OPSYS.1	18%	10%	9%	8%	2%	5%
OPSYS.2	11%	9%	5%	6%	3%	1%
OPSYS.3	16%	16%	14%	9%	7%	4%
OPSYS.4	9%	4%	12%	2%	1%	6%
OPSYS.5	4%	1%	5%	1%	0%	1%
OPSYS.6	9%	9%	13%	3%	3%	7%
OPSYS.7	10%	9%	12%	4%	2%	7%
OPSYS.8	2%	0%	3%	0%	0%	1%
DATABASE.1	16%	9%	11%	9%	4%	6%
DATABASE.2	11%	6%	15%	9%	3%	8%
DATABASE.3	13%	12%	4%	8%	9%	3%
DATABASE.4	12%	7%	16%	5%	2%	10%
DATABASE.5	9%	4%	13%	3%	1%	6%
DATABASE.6	5%	5%	8%	2%	2%	6%
DATABASE.7	9%	6%	14%	4%	2%	9%
DATABASE.8	5%	1%	7%	1%	0%	1%
LISPCOMP2.1	12%	5%	12%	8%	2%	7%
LISPCOMP2.2	7%	4%	11%	4%	1%	7%
LISPCOMP2.3	4%	2%	8%	2%	1%	5%
LISPCOMP2.4	6%	3%	10%	2%	1%	4%
LISPCOMP2.5	4%	1%	7%	1%	0%	2%
LISPCOMP2.6	2%	2%	3%	2%	2%	2%
LISPCOMP2.7	3%	2%	4%	2%	1%	2%
LISPCOMP2.8	2%	0%	4%	0%	0%	1%

Table 8: Glossary for SPUR Benchmark Characterization Variables % variables are shown as percentages in Table 9 but are intended as fractions elsewhere	
%load	fraction of instructions that load data into processor registers
%store	fraction of instructions that store data from processor registers
1st$mr	fraction of instructions not found in the on-chip instruction buffer
1&2$mr	fraction of instructions and data not found in either the on-chip instruction buffer or the off-chip mixed cache
%writeback	fraction of misses in the off-chip mixed cache that replace a modified block (requiring the modified block to be written back to memory in a separate memory request)
%float	fraction of instructions that are executed by the floating point coprocessor
avgfloat	average number of processor cycles that a floating point operation executes after the cycle it is issued

Benchmark	#SPUR instrs	%load	%store	miss ratios		%write back	%float	avgfloat stall
				1st$	1&2$			
puzzle *	33.3 M	20.8	0.1	.065	.0001	6.3	9.2	3.8
LISPCOMP2	20.0 M	21.0	5.0	.217	.0193	21.7	0.6	5.0
weaver25	25.0 M	17.5	1.7	.199	.0063	10.7	11.6	4.7
weaver200	49.5 M	19.2	2.3	.183	.0121	15.4	10.3	4.7
rsim10	17.3 M	12.1	2.8	.245	.0049	42.0	1.85	4.4
rsim100	51.6 M	13.7	3.6	.247	.0129	27.2	1.5	5.2
boyer2	17.7 M	23.5	10.1	.202	.0078	65.9	0.0	5.1
slc.boyer	23.3 M	13.4	2.3	.177	.0096	18.3	0.6	5.1
fft	35.5 M	8.5	2.8	.228	.0066	66.5	3.3	3.7
slc.fft	8.9 M	21.3	4.6	.194	.0173	22.1	1.1	4.8
ttt	3.2 M	23.0	7.2	.245	.0128	29.9	2.8	4.7
average	25.2 M	17.3	4.2	.214	.0110	32.0	3.4	4.7

Table 9

Data available about the SPUR uniprocessor performance for a set of LISP programs. LISPCOMP2 is used in the analysis of our models and is the SPUR lisp compilation of the register allocation portion of itself. Weaver25 and weaver200 are different length executions of the OPS5 layout router program. Rsim10 and rsim100 are different length executions of the circuit simulation of a 10 bit counter. Boyer2 is a theorem proving program, fft is the execution of a FFT algorithm, ttt is the OPS5 tic tac toe game, and puzzle is a small puzzle solving program. Slc.boyer and slc.fft are the SPUR lisp compilation of boyer2 and fft respectively. The fraction of floating point instructions may be higher than the benchmarks should require because SPUR uses the floating point coprocessor to do integer multiply. Even so, these programs could not be called floating point intensive.

** The average values exclude puzzle because it is simply too optimistic.*

Program	$t_{compute}$	$t_{transfer}$	ρ	N^*	N'	N = 1		N = 6		N = 10	
						MIPS	BU	MIPS	BU	MIPS	BU
puzzle	11603	8	0.001	1451.3	2901.7	4.5	0.00	26.8	0.00	44.7	0.01
LISPCOMP2	61	8	0.131	8.6	16.2	3.2	0.12	17.4	0.63	24.6	0.90
weaver25	249	8	0.032	32.1	63.3	3.1	0.03	18.6	0.19	30.9	0.31
weaver200	122	8	0.066	16.2	31.4	3.0	0.06	17.8	0.36	28.6	0.58
rsim10	214	8	0.037	27.7	54.4	3.8	0.04	22.4	0.21	37.0	0.36
rsim100	95	8	0.084	12.9	24.8	3.4	0.08	19.4	0.45	30.4	0.70
boyer2	100	8	0.080	13.5	26.0	3.6	0.07	20.7	0.43	32.7	0.68
slc.boyer	118	8	0.068	15.8	30.6	4.0	0.06	23.5	0.37	37.8	0.60
fft	143	8	0.056	18.8	36.6	3.6	0.05	21.4	0.31	34.8	0.51
slc.fft	67	8	0.119	9.4	17.7	3.3	0.11	18.5	0.59	27.0	0.86
ttt	91	8	0.088	12.4	23.7	3.1	0.08	17.9	0.46	27.9	0.72
average	126	8	0.064	16.7	32.5	3.4	0.06	20.0	0.35	32.4	0.57
typical	104	8	0.077	14.0	27.0	3.4	0.07	19.6	0.41	31.1	0.66

Table 10

The model parameters, $t_{compute}$, $t_{transfer}$, (expressed in processor cycles) and their ratio $\rho = t_{transfer}/t_{compute}$ are shown here for each of the SPUR lisp benchmarks. From these we show the 4-point bound metrics N^*, the maximum number of effective uniprocessors, and N', the number of processors that ensure that the system is at saturation. Then we show the simple queueing model predictions of system MIPS and bus utilization for 1-, 6-, and 10-processor systems. System MIPS is computed as effective uniprocessors times uniprocessor MIPS. The typical benchmark corresponds to a hypothetical program with characteristics given by the averages in Table 9. Because the typical result is more pessimistic than the average result and because it seems more intuitive to use an average mix of characteristics instead of an average mix of benchmarks, the remainder of this section will use the typical result to represent overall performance. Notice that this table justifies our exclusion of puzzle because it is phenomenally optimistic. It will be included elsewhere to indicate the ideal.

second level cache does not have the data and must issue a memory request (with an average on-board overhead of 7.5 processor cycles and 2.5 bus cycles).

Using the data in Table 9, we calculate the compute time between memory requests as follows:

$$t_{compute} = CycleTime \left[MissOverhead + \frac{avgCycles/Instr(if\ no\ memory\ requests)}{avgRequests/Reference\ *\ avgReferences/Instr} \right]$$

$$t_{compute} = CycleTime \left[9.2 + \frac{1 + \%store + \%float\ *\ avgfloat + 2\ *\ 1st\$mr + 1st\$mr(\%load + \%store)}{1\&2\$mr(1+\%writeback)\ *\ (1 + \%load + \%store)} \right]$$

The blocks of the second level cache in SPUR are 8 words (32 bytes). We assume memory that takes 400 nsec to deliver the first word of a block and 100 nsec to deliver subsequent words and we assume a processor cycle time of 150 nsec. An application of our simple queueing model then estimates effective uniprocessors. We convert effective uniprocessors to SPUR MIPS by multiplying by uniprocessor SPUR MIPS, which is estimated as the average number of instructions executed between memory requests divided by the uniprocessor inter-request time ($t_{transfer}+t_{compute}$). Finally, the average number of instructions between requests is estimated as

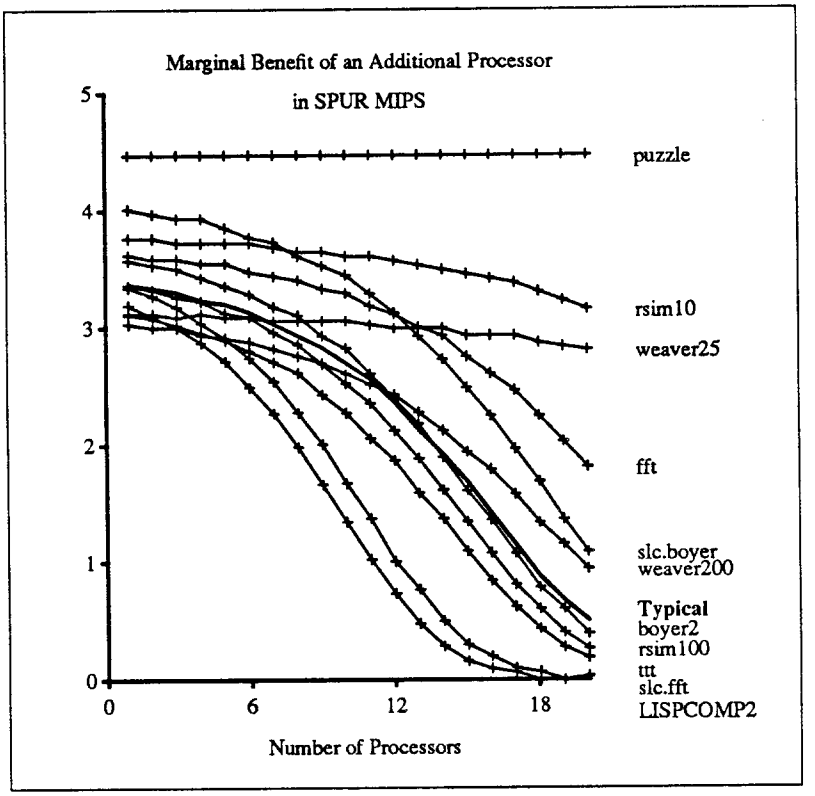

Figure 11

The marginal benefit of an additional processor is the increase in system MIPs resulting from the addition of an equally loaded processor. Notice that the typical benchmark gets about 2.5 SPUR MIPS from the 12th processor, so even the last processor in a SPUR system may be expected to add more than 70% of a uniprocessor to the system.

$$1/(1\&2\$mr(1+\%writeback)*(1+\%load+\%store)).$$

The resulting multiprocessor performance estimates for these programs on SPUR are graphed in Figure 9 and, in Table 10, we show the parameters, $t_{compute}$, $t_{transfer}$, their ratio $\rho = t_{transfer}/t_{compute}$, and performance estimates, N^*, N', and the total MIPS and bus utilization for the uni-, 6- and 10-processor systems. In Figure 11, the marginal benefit of an additional processor is shown for each benchmark. This is useful for determining when the addition of a processor is no longer cost effective.

7.2. Design Tradeoff Analysis in SPUR

The preceding section shows that the analysis of SPUR multiprocessor performance is highly dependent on processor stalls. It is educational to examine the potential performance improvements that arise from reductions in processor stalls. But reducing processor stalls requires architectural changes; thereby, requiring

Table 11: SPUR Design Changes Considered	
100nsSPUR	decreases the processor cycle time from 150 nsec to 100 nsec. This is the most difficult change to make because both cache access times, all processor and coprocessor component speeds and board logic speeds must all scale down by 2/3. We also expect this change to pay off with the largest performance gain.
noSTstall	eliminates stalls for store instructions. This might be done by delaying each written word in a buffer until the next write operation (VAX 8800).
fastFP	overlaps all floating point operations with integer operations. In SPUR users can request overlap where possible if they are prepared to deal with a limited imprecise interrupt problem.
1cycleIBmiss	reduces the instruction buffer miss stall to 1 cycle. This might be done in SPUR if the instruction buffer adds a forwarding path and the execution unit changes to accept late instructions. Unfortunately, this change will probably defeat instruction buffer prefetching; thereby, raising the instruction buffer miss ratio.
noIB	eliminates the instruction buffer and adds a 1 cycle stall to each load or store instruction to accommodate instruction fetch. Currently the instruction buffer delivers an instruction to the execution unit in well less than a cycle. Making this this change without stretching the cycle time or number of pipeline stages would be difficult.
20%improveIB	decreases the instruction buffer miss ratio by 20% while maintaining the same total number of board cache misses.
50%improveIB	decreases the instruction buffer miss ratio by 50% while maintaining the same total number of board cache misses. Perhaps the best way to get good increases in instruction buffer performance is to double its size and double the number of words brought in from the board cache on each fetch or prefetch. Of course, this calls for more chip space and pins.
20%improveB$	decreases the board cache miss ratio by 20% while maintaining the same total number of blocks written back.
50%improveB$	decreases the board cache miss ratio by 50% while maintaining the same total number of blocks written back. Two ways to achieve substantial decreases in the board cache miss ratio are to double the cache size or make it two way set associative. The first costs board space and the second may increase the processor cycle time.
64bitBus	doubles the width of memory and backplane transfer words.

modifications to simulation software and re-simulation of portions of the system. To reduce the cost of this design evaluation process, we employ our simple queueing model.

Design Change	$t_{compute}$	ρ	N^*	N'	MIPS						
					N = 1	N = 2	N = 3	N = 6	N = 10	N = 20	N => N'
SPUR as is	104	.077	14	27	3.4	6.7	10.0	19.6	31.1	46.3	47.2
100nsSPUR	105	.114	10	18	4.8	9.6	14.2	27.0	39.9	47.2	47.2
noSTstall	102	.079	14	26	3.4	6.8	10.2	20.0	31.6	46.4	47.1
fastFP	95	.084	13	25	3.7	7.3	10.8	21.1	33.1	46.7	47.1
1cycleIBmiss	92	.087	12	24	3.8	7.5	11.2	21.7	33.9	46.9	47.2
noIB	89	.090	12	23	3.9	7.7	11.5	22.3	34.6	47.0	47.3
20%improveIB	99	.081	13	26	3.5	7.0	10.5	20.5	32.3	46.6	47.2
50%improveIB	91	.088	12	24	3.8	7.6	11.3	22.0	34.3	46.9	47.2
20%improveB$	121	.066	16	31	3.5	6.9	10.3	20.2	32.6	53.0	55.6
50%improveB$	162	.049	21	41	3.6	7.2	10.7	21.2	34.8	63.5	76.0
64bitBus	104	.051	21	40	3.5	6.9	10.3	20.4	33.4	60.4	70.7

Table 12

This table reports performance estimates for 10 design changes to the SPUR multiprocessor. The parameter $t_{transfer}$ is 12 bus cycles except in the widebus case where it is 8 bus cycles. Total system MIPS are reported from simple queueing estimates of effective uniprocessors. The maximum system MIPS occurs when the bus is saturated (as it will certainly by with more than N' processors) and is reported in the last column.

In Table 11 we describe 10 SPUR design changes, some of which should be thought of as goals for the redesign of processor components.[5] Each of the benchmarks in Table 9, except for the unrealistically optimistic puzzle program, was re-evaluated for each design change. The simple queueing model used 7.4 SUN 3 CPU ***minutes*** to compute new performance estimates. The equivalent simulation time would have been over 7.2 SUN 3 CPU ***weeks.*** Table 12 shows the average parameter values and performance estimates for each design and Figure 12 shows the percent improvement in SPUR MIPS of the average performance for each design change.

From the 4-point bound we know that the maximum speed up in a system will be N^*, so the maximum system MIPS is

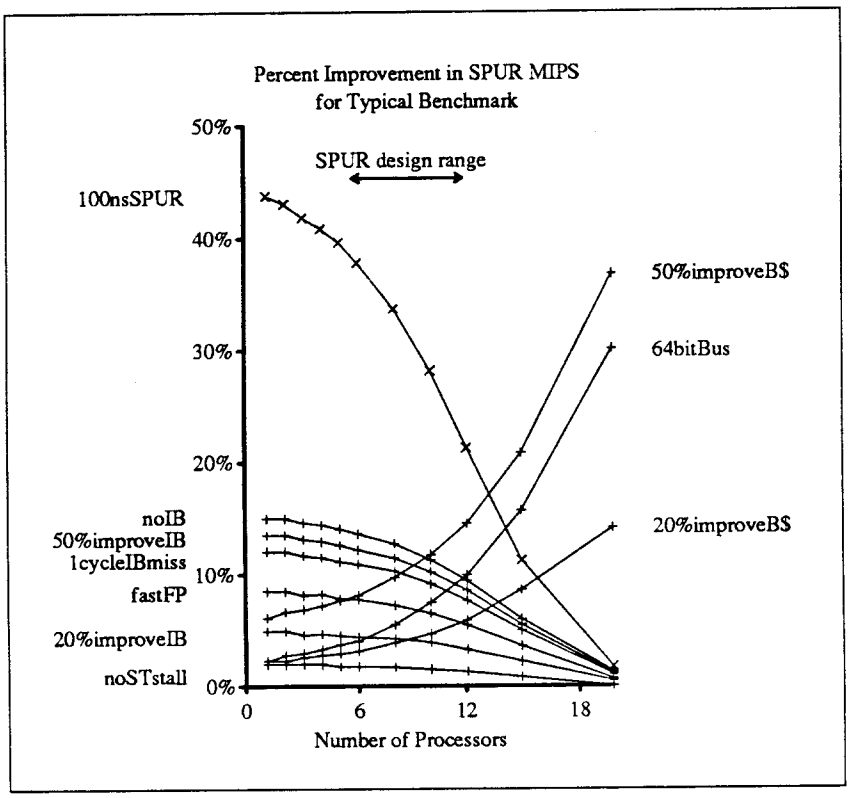

Figure 12

Each of the 10 design changes considered has its percent improvement over SPUR graphed here. If a 10% improvement across the design range is required before a change is implemented, then only a reduction in the cycle time is a clear candidate.

[5] There are many other design changes that the SPUR design team has discussed. These are merely one selection that is easily explored with the data provided about the lisp benchmarks.

$$N^* \; UniMIPS = \frac{t_{compute} + t_{transfer}}{t_{transfer}} \; \frac{10^{-6}}{(t_{compute} + t_{transfer}) \; 1\&2\$mr \; (1 + \%writeback) \; (1 + \%load + \%store)}$$

and this is independent of the processor cycle time (if processor performance is unaffected by decreases in the cycle time) and many of our design changes as is shown in Table 12. This means that a faster uniprocessor will not allow the system to achieve a higher overall MIPS rate, but that it will achieve the maximum with fewer processors. The practical caveat is that bus cycle times are dependent on bus length and this is increased when large number of processors are interconnected.

In Figure 13 the marginal benefit of an additional processor is shown for the typical benchmark in the basic SPUR design and under each design change.

From these results we might conclude that:

- decreasing the processor cycle time is, as expected, the best way to increase the performance of the system in the SPUR design range,

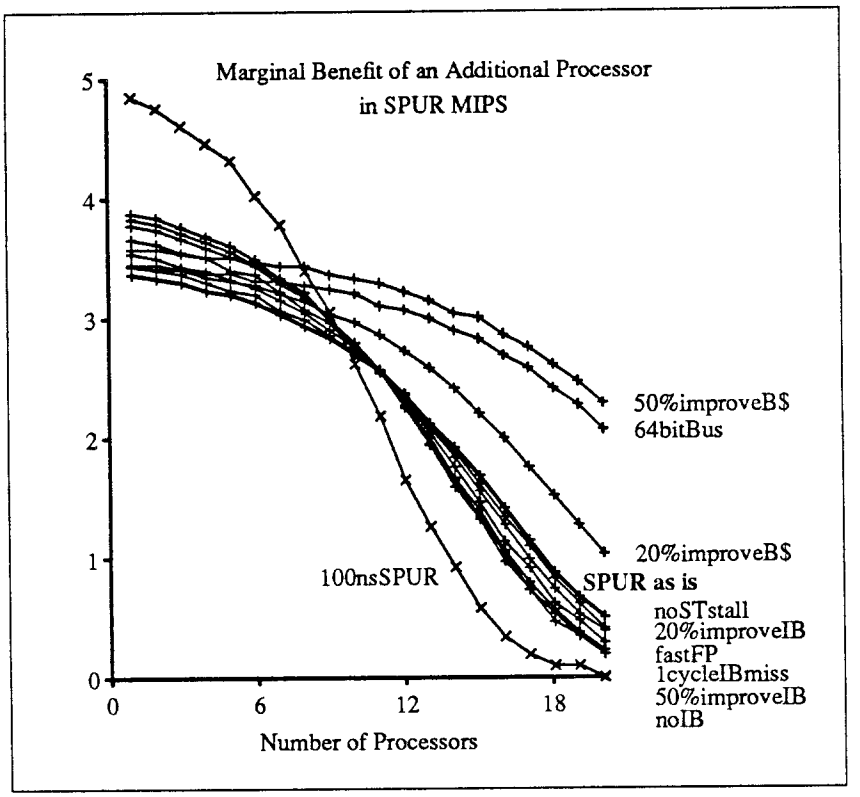

Figure 13

This graph presents the marginal benefit of an additional processor for each of the 10 design changes and the original SPUR design. As we would expect, in a shared bus environment faster processors become cost inefficient more quickly.

- improvements to the instruction buffer that do not stretch the cycle time are a good way to speed up a uniprocessor SPUR,
- improvements to the board cache miss ratio or the memory bandwidth are the best way to speed up a many processor SPUR,
- removing the 1 cycle stall on store instructions is not worth much,
- and decreasing processor demands on external memory or increasing memory bandwidth are two good ways to make additional processors cost effective in large systems.

8. Future Work

Our study uses independent processor workloads to evaluate potential system performance. Unfortunately, a multiprocessor used for parallel processing may suffer data sharing penalties from two sources: locks and cache consistency.

Contention for locks may cause processors to context switch or spin polling the data across the backplane, introducing higher memory traffic rates in both cases, or spin in place waiting for some form of IPC signal, possibly lowering memory traffic rates. It is not yet clear how to estimate the effect of locks without information on the mechanism and distribution of lock contention.

Cache consistency solutions sometimes require extra backplane transfers [Katz85]. To assess the effect of cache consistency solutions techniques, information is needed on the distribution of consistency related transfers.

When cache blocks are small, a significant portion of the bus bandwidth is lost while memory is accessing the first word of a block. By packet switching both the memory request and response [Fielland84, Frank84] in a multiple memory unit system, some of this bandwidth can be regained. Since the bus is not held while the memory is active, the simple queueing model may not be appropriate in this case.

9. Summary

A rule of thumb, the 4-point bound, is a very simple tool for eliminating large numbers of design choices. When greater accuracy is needed a simple queuing model, the M/M/1//N queue in queueing theory terminology, provides estimations of effective uniprocessors and bus utilization metrics within 4% of trace-driven simulation using orders of magnitude less CPU time and disk space. Both of these models are based on the ratio of the mean memory service time and the mean processor compute time between memory requests,

$\frac{t_{transfer}}{t_{compute}}$, rather than on the individual values of these means or their corresponding distributions.

10. Acknowledgments

Thanks to Kris Anderson, Mark Hill, Wen-Mei Hwu, Brent Welch, and David Wood for their valuable comments on a draft of this paper. Special thanks to Mark and David for the many discussions on this material, particularly those that led to the paranoid bound in the second appendix. Special thanks are also due to my advisors in this work, Dave Patterson and Alan Smith.

The OPSYS trace was provided by Bill Harding and the Amdahl Corporation. The DATABASE trace was provided by Joe Hull, Mark Francis, Rollie Schmidt and the Synapse Computer Corporation.

We would like to thank Richard Newton and DEC for the use of their VAX 8650 computer time. And for keeping the SPUR simulator running hour after hour, we would like to thank George Taylor and Ben Zorn.

Principal funding this work is provided by a Natural Sciences and Engineering Research Council of Canada Postgraduate Scholarship, by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269 and by computer resources provided under DARPA contract N00039-84-C-0089.

11. Bibliography

[Allen78] A.O. Allen, *Probability, statistics, and queueing theory with computer science applications*, Academic Press, New York, 1978.

[Allen80] A.O. Allen, "Queueing models of computer systems," *IEEE Computer*, April 1980, pp 13-24.

[Baskett76] F. Baskett, A.J. Smith, "Interference in multiprocessor computer systems with interleaved memory", *CACM*, vol 19, no 6, June 1976, pp 327-334.

[Baskett86] F. Baskett, J.L. Hennessy, "Small shared-memory multiprocessors," *Science*, vol 231, February 1986, pp 963-967.

[Bell85] C.G. Bell, "Multis: a new class of multiprocessor computers," *Science*, vol 228, April 1985, pp 462-467.

[Bentley84] J. Bentley, "The back of the envelope," *CACM*, vol 27, no 3, March 1984, pp 180-183.

[Buzen77] J.P. Buzen, D. Potier, "Accuracy of exponential assumptions in closed queueing models," *Proc. 1977 SIGMETRICS/CMG Int. Conf. Comput. Perf. Modeling, Measurement*, Washington DC, November 1977, pp 53-64.

[Chandy78] K.M. Chandy, C.M. Sauer, "Approximate methods for analyzing queuing network models of computing systems," *Computing Surveys*, vol 10, no 3, September 1978, pp 281-317.

[Clark83] D.W. Clark, "Cache performance in the VAX-11/780," *ACM Transactions on Computer Systems*, vol 1, no 1, February 1983.

[Clark85] D.W. Clark, J.S. Emer, "Performance of the VAX-11/780 translation buffer," *ACM Transactions on Computer Systems*, vol 3, no 1, February 1985.

[Deminent82] J. Deminent, "Experience with multiprocessor algorithms," *IEEE Transactions on Computers*,

vol C-31, no 4, April 1982.

[Denning78] P. Denning, J. Buzen, "The operational analysis of queueing network models", *Computing Surveys*, vol 10, no 3, September 1978, pp 225-261.

[Dubois85] M. Dubois, "A cache-based multiprocessor with high efficiency," *IEEE Transactions on Computers*, vol C-34, no 10, October 1985.

[Gibson85] G. Gibson, "SpurBus specification," *Proceedings of CS292I: Implementation of VLSI Systems*, ed. R.H. Katz, University of California, Berkeley, September 1985. Also Computer Science Division technical report UCB/CSD 86/259.

[Fielland84] G. Fielland, D. Rogers, "32-bit computer system shares load equally among up to 12 processors," *Electronic Design*, September 1984, pp 153-168.

[Frank84] S.J. Frank, "Tightly coupled multiprocessor system speeds memory-access times," *Electronics*, vol 57, no 1, January 1984, pp 164-169.

[Fuller76] S.H. Fuller, "Price/performance comparison of C.mmp and the PDP-10," *Proceedings of the 3rd Annual Symposium on Computer Architecture*, Pittsburgh, Penn., January 1976, pp 195-202.

[Hill83] M.D. Hill, "Evaluation of on-chip cache memories," Unpublished Master's Report, University of California, Berkeley, December 1983.

[Hill86] M.D. Hill, S.J Eggers, J. Larus, G. Taylor, et al, "Design decisions in SPUR," *IEEE Computer*, vol C-19, no 11, Nov 1986.

[Katz85] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, R.G. Sheldon, "Implementing a cache consistency protocol," *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, Mass., June 1985, pp 276-283.

[Kinney78] L.L. Kinney, R.G. Arnold, "Analysis of a multiprocessor system with a shared bus," *Proceedings of the 5rd Annual Symposium on Computer Architecture*, January 1978, pp 89-95.

[Kleinrock75] L. Kleinrock, *Queuing systems*, vol 1, Wiley, New York, 1975.

[Marsan82] M.A. Marsan, G. Balbo, G. Conte, "Comparative performance analysis of single bus multiprocessor architectures," *IEEE Transactions on Computers*, vol C-31, no 12, December 1982, pp 1179-1191.

[Marsan83] M.A. Marsan, G. Balbo, G. Conte, F. Gregoretti, "Modeling bus contention and memory interference in a multiprocessor system," *IEEE Transactions on Computers*, vol C-32, no 1, January 1983, pp 60-72.

[Marshall68] K.T. Marshall, "Some Inequalities in Queuing," *Operations Research*, vol 16, no 3. 1968, pp 651-665.

[Patel82] J.H. Patel, "Analysis of multiprocessors with private cache memories," *IEEE Transactions on Computers*, vol C-31, no 4, April 1982, pp 296-304.

[Reyling74] G. Reyling Jr., "Performance and control of multiple microprocessors systems," *Computer Design*, March 1974, pp 81-86.

[Saaty61] T.L. Saaty, *Elements of Queuing Theory*, McGraw-Hill, 1961, pp 323-329.

[Smith85] A.J. Smith, "Cache evaluation and the impact of workload choice," *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, Mass., June 1985, pp 64-73.

[Taylor86] G.S. Taylor, P.N. Hilfinger, J.R. Larus, D.A. Patterson, B.G. Zorn, "Evaluation of the SPUR lisp architecture," *Proceedings of the 13th International Symposium on Computer Architecture*, Tokyo, Japan, June 1986, pp 444-452.

[Towsley86] D. Towsley, "Approximate models of multiple bus multiprocessor systems," *IEEE Transactions on Computers*, vol C-35, no 3, March 1986.

[Wood86] D.A. Wood, S.J Eggers, G. Gibson, M.D. Hill, J.M. Pendleton, S.A. Ritchie, G.S. Taylor, R.H. Katz, D.A. Patterson, "An in-cache address translation mechanism," *Proceedings of the 13th*

International Symposium on Computer Architecture, Tokyo, Japan, June 1986, pp 444-452.

12. Appendix A: Raw Data

Some of the bus simulation results are presented in Tables 13 and 14. The values found in these tables are not important to the accuracy of the models, but may interest readers. The SPUR project uses cache configuration 4. Its processor and memory speeds are not firm at the moment, but should fall between configurations A, B and C (see Section 7). These tables list the value of effective uniprocessors, bus utilization and average wait on memory in nanoseconds for the 6-processor and 10-processor systems for each request trace and speed configuration. We also report the trace $\rho = t_{transfer} / t_{compute}$, the uniprocessor execution time in seconds and the uniprocessor bus utilization. Notice that there is insufficient information in these tables to compare cache designs.

Config A (100ns per ref, 400ns 1st word, 100ns 2nd- word)									
Trace.Cache	ρ	Uni Sim Secs	Uni Bus Util	At 6 Processors EU	Bus Util	Avg Wait	At 10 Processors EU	Bus Util	Avg Wait
OPSYS.1	0.130	0.11	0.12	5.5	0.62	6719	8.3	0.91	10731
OPSYS.2	0.124	0.11	0.11	5.5	0.61	2148	8.2	0.88	3656
OPSYS.3	0.138	0.11	0.12	5.5	0.66	1046	8.0	0.94	1926
OPSYS.4	0.265	0.13	0.21	4.4	0.92	3320	4.7	1.00	7561
OPSYS.5	0.968	0.20	0.50	2.0	1.00	17973	2.0	1.00	32369
OPSYS.6	0.213	0.12	0.18	4.9	0.87	1156	5.6	1.00	2716
OPSYS.7	0.178	0.12	0.16	5.1	0.78	1294	6.5	0.99	2722
OPSYS.8	3.452	0.46	0.78	1.3	1.00	38924	1.3	1.00	66124
DATABASE.1	0.049	1.32	0.05	5.9	0.28	4296	9.8	0.46	5263
DATABASE.2	0.063	1.34	0.06	5.9	0.35	1513	9.6	0.57	2027
DATABASE.3	0.054	1.33	0.05	5.9	0.31	737	9.7	0.50	954
DATABASE.4	0.157	1.46	0.14	5.4	0.74	2239	7.0	0.96	4935
DATABASE.5	0.330	1.68	0.25	3.9	0.97	11485	4.0	1.00	25193
DATABASE.6	0.204	1.53	0.18	4.8	0.84	1208	5.7	0.99	2685
DATABASE.7	0.160	1.47	0.14	5.3	0.75	1167	6.9	0.97	2554
DATABASE.8	1.015	2.56	0.51	2.0	1.00	34200	2.0	1.00	61399
LISPCOMP2.1	0.081	2.16	0.08	5.8	0.44	4997	9.3	0.70	7290
LISPCOMP2.2	0.105	2.21	0.10	5.7	0.55	1882	8.6	0.82	3256
LISPCOMP2.3	0.145	2.30	0.13	5.4	0.69	1151	7.3	0.94	2299
LISPCOMP2.4	0.284	2.58	0.23	4.2	0.94	3540	4.4	1.00	7886
LISPCOMP2.5	0.713	3.46	0.42	2.4	1.00	16670	2.4	1.00	31051
LISPCOMP2.6	0.189	2.39	0.16	4.9	0.81	1157	6.0	0.98	2537
LISPCOMP2.7	0.238	2.50	0.20	4.5	0.89	1625	5.0	1.00	3604
LISPCOMP2.8	2.345	6.86	0.71	1.4	1.00	38004	1.4	1.00	65203

Table 13: Raw Data for Configuration A

Configuration A sets the mean time between processor references into its caches at 100 nsec, the memory latency on the first word of a block to 400 nsec and the memory latency on subsequent words in a block to 100 nsec.

Config B (100ns per ref, 400ns 1st word, 400ns 2nd- word)									
Trace.Cache	ρ	Uni Sim Secs	Uni Bus Util	At 6 Processors EU	Bus Util	Avg Wait	At 10 Processors EU	Bus Util	Avg Wait
OPSYS.1	0.468	0.15	0.32	3.2	1.00	49338	3.2	1.00	101236
OPSYS.2	0.342	0.13	0.26	3.9	0.97	10400	3.9	1.00	23421
OPSYS.3	0.207	0.12	0.17	5.1	0.85	1918	5.9	1.00	4614
OPSYS.4	0.728	0.17	0.43	2.4	1.00	15372	2.3	1.00	28565
OPSYS.5	3.468	0.46	0.78	1.3	1.00	73770	1.3	1.00	125367
OPSYS.6	0.213	0.12	0.18	4.9	0.87	1156	5.6	1.00	2716
OPSYS.7	0.267	0.13	0.22	4.3	0.92	2550	4.7	1.00	5707
OPSYS.8	13.046	1.47	0.93	1.1	1.00	152322	1.1	1.00	255122
DATABASE.1	0.176	1.48	0.15	5.2	0.77	26541	6.5	0.98	58733
DATABASE.2	0.172	1.48	0.15	5.2	0.76	6866	6.6	0.97	14983
DATABASE.3	0.082	1.36	0.08	5.8	0.44	1253	9.3	0.69	1854
DATABASE.4	0.433	1.81	0.30	3.3	0.99	12471	3.3	1.00	25463
DATABASE.5	1.183	2.76	0.54	1.8	1.00	66593	1.8	1.00	118190
DATABASE.6	0.204	1.53	0.18	4.8	0.84	1208	5.7	0.99	2685
DATABASE.7	0.241	1.57	0.20	4.5	0.90	2378	5.0	1.00	5388
DATABASE.8	3.836	6.16	0.80	1.3	1.00	147600	1.3	1.00	250399
LISPCOMP2.1	0.290	2.58	0.23	4.2	0.94	37735	4.4	1.00	84664
LISPCOMP2.2	0.289	2.58	0.23	4.1	0.94	9834	4.4	1.00	21719
LISPCOMP2.3	0.217	2.45	0.18	4.7	0.86	2253	5.5	0.99	4999
LISPCOMP2.4	0.782	3.60	0.44	2.2	1.00	15690	2.2	1.00	28881
LISPCOMP2.5	2.554	7.22	0.72	1.4	1.00	72451	1.4	1.00	124051
LISPCOMP2.6	0.189	2.39	0.16	4.9	0.81	1157	6.0	0.98	2537
LISPCOMP2.7	0.357	2.74	0.27	3.6	0.97	3148	3.7	1.00	6584
LISPCOMP2.8	8.862	20.39	0.90	1.1	1.00	151404	1.1	1.00	254203
Config C (200ns per ref, 400ns 1st word, 100ns 2nd- word)									
OPSYS.1	0.065	0.21	0.06	5.8	0.35	5405	9.4	0.56	7626
OPSYS.2	0.062	0.21	0.06	5.9	0.34	1708	9.5	0.55	2230
OPSYS.3	0.069	0.21	0.07	5.8	0.38	860	9.4	0.60	1200
OPSYS.4	0.132	0.23	0.12	5.6	0.66	2034	7.9	0.93	3938
OPSYS.5	0.484	0.30	0.33	3.0	0.99	14458	3.0	1.00	28743
OPSYS.6	0.106	0.22	0.10	5.7	0.56	796	8.7	0.84	1279
OPSYS.7	0.089	0.22	0.08	5.8	0.48	867	9.0	0.75	1380
OPSYS.8	1.726	0.56	0.64	1.6	1.00	37049	1.6	1.00	64247
DATABASE.1	0.025	2.58	0.02	6.0	0.14	3916	10.0	0.24	4216
DATABASE.2	0.031	2.60	0.03	6.0	0.18	1329	9.9	0.30	1474
DATABASE.3	0.027	2.59	0.03	6.0	0.16	665	9.9	0.26	753
DATABASE.4	0.079	2.72	0.07	5.9	0.43	1585	9.4	0.69	2301
DATABASE.5	0.165	2.94	0.14	5.3	0.76	6856	6.8	0.98	15275
DATABASE.6	0.102	2.78	0.10	5.7	0.55	767	8.6	0.82	1330
DATABASE.7	0.080	2.72	0.08	5.9	0.45	801	9.3	0.71	1190
DATABASE.8	0.507	3.81	0.34	2.9	1.00	27755	2.9	1.00	54801
LISPCOMP2.1	0.041	4.16	0.04	6.0	0.23	4190	9.9	0.38	4879
LISPCOMP2.2	0.053	4.21	0.05	5.9	0.30	1478	9.7	0.49	1858
LISPCOMP2.3	0.073	4.30	0.07	5.8	0.40	824	9.4	0.65	1161
LISPCOMP2.4	0.142	4.58	0.13	5.5	0.69	2155	7.4	0.94	4454
LISPCOMP2.5	0.356	5.46	0.27	3.7	0.97	12252	3.8	1.00	26103
LISPCOMP2.6	0.094	4.39	0.09	5.7	0.51	774	8.8	0.78	1269
LISPCOMP2.7	0.119	4.50	0.11	5.6	0.61	1028	8.0	0.89	1928
LISPCOMP2.8	1.172	8.86	0.55	1.8	1.00	35208	1.8	1.00	62408

Table 14: Raw Data for Configuration B and C

Configuration B sets the mean time between processor references into its caches at 100 nsec, the memory latency on the first word of a block to 400 nsec and the memory latency on subsequent words in a block to 400 nsec. Configuration C sets these values to 200, 400 and 100, respectively.

13. Appendix B: More on Pessimistic Estimations

As noted in Section 5, the pessimistic estimate described in Section 3.2 does not give a strict lower bound on effective uniprocessors. This appendix presents a model that does give such a lower bound and relates it to the pessimistic estimate.

13.1. A Lower Bound: The Paranoid Bound

A lower bound on multiprocessor throughput performance (both MIPS and effective uniprocessors) occurs when the execution time of the N workloads, one per processor, is maximized. The execution time can be broken down into the time the memory is busy and the time it is idle. If each workload contains R memory requests then the portion of run time that the memory is busy is $NRt_{transfer}$ units long. Since all processors are computing when the memory is idle and since the total computation time per processor is $Rt_{compute}$, the maximum time that the memory is idle must be $\leq Rt_{compute}$. So the maximum multiprocessor execution time is $NRt_{transfer} + Rt_{compute}$.

A lower bound on effective uniprocessors is then

$$EU \geq \frac{N(Uniprocessor\ Run\ Time)}{Multiprocessor\ Run\ Time} = \frac{N(t_{transfer}+t_{compute})}{Nt_{transfer}+t_{compute}}$$

since the uniprocessor execution time is fixed at $R(t_{transfer}+t_{compute})$ per workload. This maximizes the memory idle time, so it also gives rise to a lower bound on bus utilization

$$BU \geq \frac{Nt_{transfer}}{Nt_{transfer}+t_{compute}}$$

and we see that the bus is never saturated. To bound average wait for memory service, notice that if arbitration is fair, the most a processor could wait for service is a transfer time for each of the other N-1 processors plus its own. So $AW \leq Nt_{transfer}$.

We name the implicit model giving rise to these lower bounds, shown in Figure 14, the *paranoid bound.* It is interesting to evaluate the paranoid bound at N^* and N'.

$$EU(N^*) = \frac{N^*}{2-\frac{1}{N^*}}$$

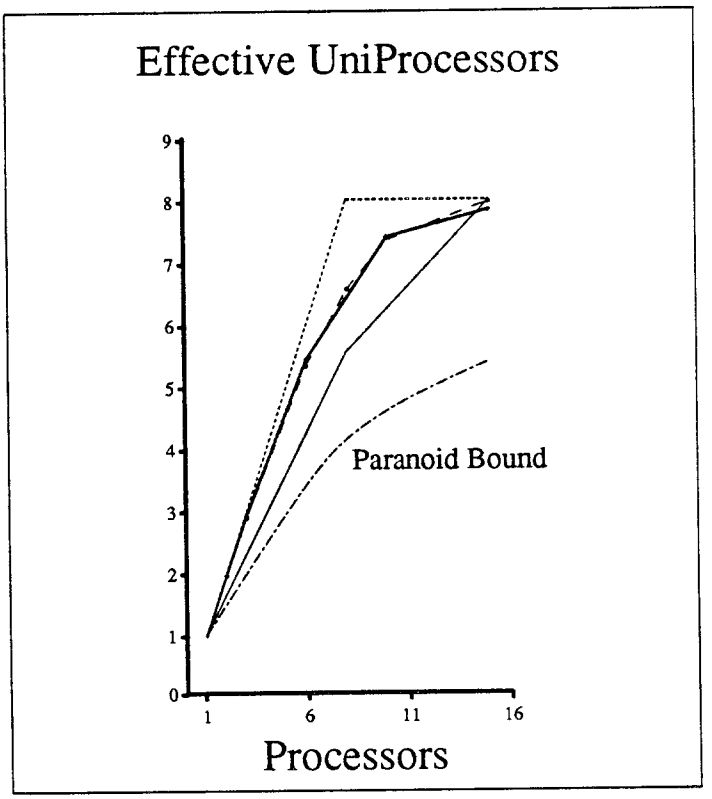

Figure 14

A sample of the paranoid bound's estimation of effective uniprocessors is shown in this figure with the corresponding trace-driven simulation, 4-point bound and simple queueing model estimations. Although the paranoid bound is a true lower bound, this example demonstrates its weaknesses in practice. In this example, the ratio, $t_{transfer}/t_{compute}$, *is .142.*

$$EU(N') = \frac{N^*}{2 - \frac{N^*}{N'}} = \frac{N^*}{\frac{3 + \frac{t_{transfer}}{t_{compute}}}{2 + \frac{t_{transfer}}{t_{compute}}}}$$

So the paranoid bound exceeds $\frac{1}{2}N^*$ at N^* and exceeds $\frac{2}{3}N^*$ at N', since $0 < \frac{t_{transfer}}{t_{compute}} < \infty$.

13.2. A Family of Pessimistic Models

The pessimistic estimate of Section 3.2 and the paranoid bound are related; there is a family of models that have the pessimistic estimate as their most optimistic member and the paranoid bound as their limiting pessimistic model.

We construct our family of pessimistic models by extending the pessimistic estimate. Let requests be organized into groups, r requests to a group, such that at the beginning of each request period all processors

simultaneously initiate r back-to-back (no intervening computation) requests. Then processors compute (with duration $rt_{compute}$) until the next request period. Referring to Figure 4, this means that the pessimistic estimate has r = 1.

Using the arguments of Section 3.2, we get the following equations[6]:

$$N\ (saturation\ knee) = 1 + 2r\frac{t_{compute}}{t_{transfer}}$$

$$EU\ (not\ saturated) = \frac{N\ (t_{compute} + t_{transfer})}{t_{compute} + Nt_{transfer} - \frac{N-1}{2r}t_{transfer}}$$

$$BU\ (not\ saturated) = \frac{Nt_{transfer}}{t_{compute} + Nt_{transfer} - \frac{N-1}{2r}t_{transfer}}$$

$$AW\ (not\ saturated) = \frac{r-1}{r}Nt_{transfer} + \frac{N+1}{2r}t_{transfer}$$

As we have noted, the pessimistic estimate has r = 1. If we let r go to ∞ then these equations give rise to the paranoid bound. Figure 13 shows a sample of the effective uniprocessors estimates of these models relative to the corresponding optimistic estimate.

13.3. What Does This Mean?

Table 15 shows the (interpolated) number of processors that cause the pessimistic estimate to exceed trace-driven simulation, for the simulation runs that had poor enough performance to show an estimate for some $N \geq N'$. Table 15 also shows the relative error (positive implies the model exceeds simulation) of the pessimistic estimate (r = 1), r = 2 model and the paranoid bound (r = ∞) at N'.

Empirically, the pessimistic estimate of Section 3.2 (r = 1) is a closer estimate of effective uniprocessors at N' than any of the other pessimistic models unless the system can't support multiple processors ($N' \leq 2\ or\ 3$). If performance near N' is particularly important then perhaps the r = 2 estimate at N' should be joined with the r = 1 estimate at N^* to give a more pessimistic bound (notice that near N' each processor may be achieving as little as 50% utilization).

The paranoid bound gives weak but strict lower bounds on effective uniprocessors at N^* and N' of $\frac{1}{2}N^*$ and $\frac{2}{3}N^*$, respectively.

[6] Notice that when r = 1/2, these equations give the same estimates as the optimistic estimate. It is not clear to us why one request every two request periods should model the best possible performance for arbitrary N.

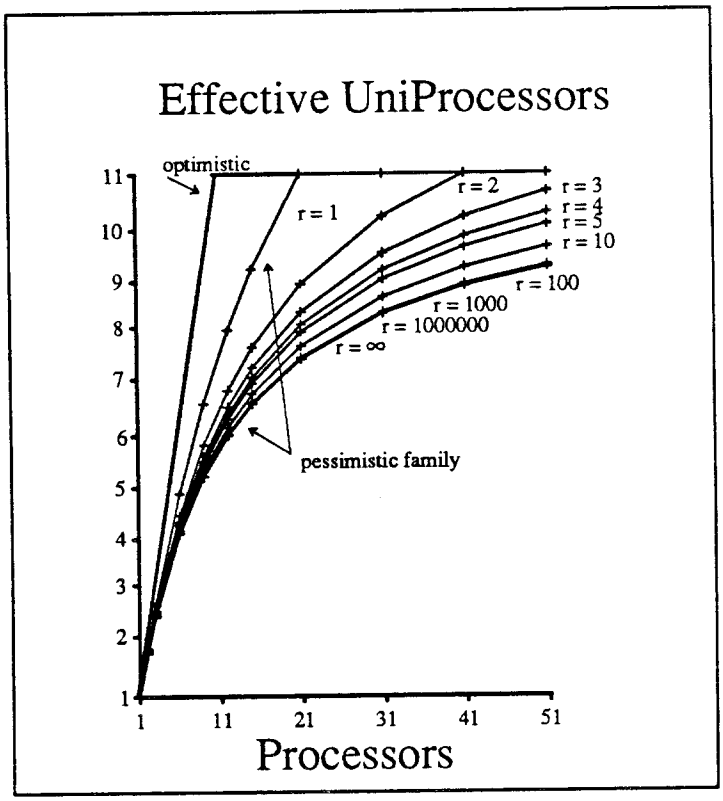

Figure 15

This graph shows effective uniprocessor estimates for the optimistic estimate against the corresponding family of pessimistic models. Notice that for r > 100, all the models give nearly the same estimate for effective uniprocessors, all approximately the paranoid bound's estimate. Also notice how much the pessimistic model estimates for r = 1 and r = 2 differ relative to the difference between the optimistic estimate and the pessimistic estimate (r = 1). In this example, the ratio, $t_{transfer}/t_{compute}$, is .1.

Trace.CacheConfig	N'	Cross Over	Error at N'		
			r = 1	r = 2	r = ∞
OPSYS.1B	5.3	4.9	3%	-14%	-26%
OPSYS.2B	6.8	7.1	0%	-18%	-30%
OPSYS.3B	10.7	11.0	-1%	-19%	-32%
OPSYS.4A	8.6	8.0	4%	-15%	-28%
OPSYS.4B	3.7	3.2	8%	-9%	-21%
OPSYS.5A	3.1	2.3	6%	-9%	-21%
OPSYS.5B	1.6	1.0	12%	3%	-5%
OPSYS.5C	5.1	4.1	8%	-10%	-23%
OPSYS.6A	10.4	10.0	2%	-17%	-30%
OPSYS.6B	10.4	10.0	2%	-17%	-30%
OPSYS.7A	12.2	12.1	2%	-17%	-30%
OPSYS.7B	8.5	7.9	4%	-14%	-28%
OPSYS.8A	1.6	1.0	12%	3%	-5%
OPSYS.8B	1.2	1.0	7%	3%	0%
OPSYS.8C	2.2	1.0	7%	-5%	-15%
LISPCOMP2.1B	7.9	7.6	3%	-15%	-28%
LISPCOMP2.2B	7.9	7.4	4%	-15%	-28%
LISPCOMP2.3A	14.8	14.1	2%	-17%	-30%
LISPCOMP2.3B	10.2	9.6	2%	-17%	-30%
LISPCOMP2.4A	8.0	7.4	5%	-14%	-27%
LISPCOMP2.4B	3.6	2.9	8%	-8%	-20%
LISPCOMP2.5A	3.8	3.2	9%	-8%	-21%
LISPCOMP2.5B	1.8	1.0	10%	0%	-9%
LISPCOMP2.5C	6.6	6.2	4%	-15%	-27%
LISPCOMP2.6A	11.6	10.7	4%	-15%	-28%
LISPCOMP2.6B	11.6	10.7	4%	-15%	-28%
LISPCOMP2.7A	9.4	8.2	5%	-14%	-27%
LISPCOMP2.7B	6.6	5.5	6%	-13%	-26%
LISPCOMP2.8A	1.9	1.0	9%	-2%	-11%
LISPCOMP2.8B	1.2	1.0	9%	4%	-1%
LISPCOMP2.8C	2.7	2.1	8%	-7%	-18%
DATABASE.1B	12.4	12.4	1%	-18%	-31%
DATABASE.2B	12.6	12.5	2%	-17%	-30%
DATABASE.4A	13.7	13.3	2%	-17%	-30%
DATABASE.4B	5.6	4.7	5%	-13%	-26%
DATABASE.5A	7.1	6.8	3%	-15%	-28%
DATABASE.5B	2.7	2.1	7%	-8%	-19%
DATABASE.5C	13.1	12.9	2%	-17%	-30%
DATABASE.6A	10.8	10.0	4%	-15%	-29%
DATABASE.6B	10.8	10.0	4%	-15%	-29%
DATABASE.7A	13.5	12.8	3%	-16%	-29%
DATABASE.7B	9.3	8.5	4%	-15%	-28%
DATABASE.8A	3.0	2.3	5%	-10%	-21%
DATABASE.8B	1.5	1.0	12%	3%	-4%
DATABASE.8C	4.9	4.1	8%	-10%	-23%

Table 15: Various Pessimistic Estimates Accuracy at N'

This table shows the trace-driven simulations that have N' less than 15 (the maximum number of processors simulated). For each simulation, the value of N at which the pessimistic estimate exceeded the simulation estimate of effective uniprocessors is shown. Then the accuracy of the r=1, r=2 and r=∞ pessimistic estimates at N' are given (positive exceeds the simulated value). Although the r=1 estimates in general exceed simulation estimates at N', they are also generally closer than the r=2 estimates unless performance is especially poor ($N' \leq 2$ or 3).