# Design of Testbed and Emulation Tools

M. J. Flynn and S. F. Lundstrom

Technical Report CSL-TR-87-337

October 1987

This report constitutes the Final Report for NASA Contract NAG2-248 (S5) for November 1985–October 1987.

# Design of Test bed and Emulation Tools

by
M. J. Flynn and S. F. Lundstrom

Technical Report CSL-TR-87-337
October 1987

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
St anford, California 943054055

**Abstract**

In order to understand how to predict the performance of concurrent computing systems, an experimental environment is needed. The purpose of the research conducted under the grant was to investigate various aspects of this environment. A first performance prediction system was developed and evaluated (by comparison both with simulations and with actual systems). The creation of a second, complementary system is well underway.

**Key Words and Phrases:** Concurrent computing, testbed, performance prediction.

Copyright © 1987
by
M. J. Flynn and S. F. Lundstrom

# Contents

**Abstract** 1

**1 Task Level Performance Prediction** 2

1.1 Statement of the Problem . . . . . . . . . . 2

1.2 Research results . . . . . . . . . . 3

1.3 TSPP: Task System Performance Predictor . . . . . . . . . . 8

Description . . . . . . . . . . 8

Execution Instruction . . . . . . . . . . 8

Input Format . . . . . . . . . . 8

Output Format . . . . . . . . . . 11

1.4 Conclusions . . . . . . . . . . 15

**2 Program Source Level Performance Prediction** 15

2.1 Value Analysis . . . . . . . . . . 15

2.2 Stochastic Cache Simulator . . . . . . . . . . 16

Concept . . . . . . . . . . 16

Histories . . . . . . . . . . 17

Replacement policy . . . . . . . . . . 18

Results. . . . . . . . . . . 19

Problems . . . . . . . . . . 21

Advantages . . . . . . . . . . 21

2.3 Conclusions . . . . . . . . . . 22

**3 Resource management for concurrent systems: a heuristic approach** 22

3.1 Introduction . . . . . . . . . . 22

3.2 Axe experimental environment . . . . . . . . . . 23

3.3 Resource Management Studies on Axe . . . . . . . . . . 24

3.4 Current Status . . . . . . . . . . 25

3.5 Conclusions and Future Work . . . . . . . . . . 25

4 **Resource management for concurrent systems: a system architecture approach** 26

4.1 Evaluation of a New Multiprocessor System Architecture . . . . . . . . 26

4.2 The Hierarchical Multiprocessor System Architecture . . . . . . . . . . . 26

4.3 Evaluation and Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

4.4 Experience with Experimental Environments . . . . . . . . . . . . . . . . . 33

Effective use of abstraction models . . . . . . . . . . . . . . . . . . . . 34

Powerful and flexible simulation tools . . . . . . . . . . . . . . . . . . . 34

Versatile study aids . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

5 **Sponsored publications** 35

6 **Other references** 36

# List of Figures


1 Divide-And-Conquer problem. . . . . . . . . . . . . . . . . . . . . . . . . . 4
2 Shared memory multiprocessor to run problem. . . . . . . . . . . . . . . . 5
3 Service demands. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
4 Results of Divide-And-Conquer system. . . . . . . . . . . . . . . . . . . . 5
5 Master-Slave task system for validation experiment. . . . . . . . . . . . . 6
6 Program code of validation experiment. . . . . . . . . . . . . . . . . . . 6
7 Queueing network model of Sequent Balance 8000. . . . . . . . . . . . . . 6
8 Results of measurement experiment. . . . . . . . . . . . . . . . . . . . . 7
9 Example task system. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
10 Example queuing network model. . . . . . . . . . . . . . . . . . . . . . . 9
11 Example input file. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
12 Example output file. . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
13 Format of output file at dump mode. . . . . . . . . . . . . . . . . . . . 14
14 Global variables. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
15 Local variables. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
16 A three-level hierarchical multiprocessor system. . . . . . . . . . . . . 27
17 A machine program of three hierarchical levels. . . . . . . . . . . . . . 28
18 Program Behavior Model. . . . . . . . . . . . . . . . . . . . . . . . . . 30
19 Average execution time with respect to ideal execution. . . . . . . . . . 31
20 Actual execution time. . . . . . . . . . . . . . . . . . . . . . . . . . . 32

# Abstract

In order to understand how to predict the performance of concurrent computing systems, an experimental environment is needed. The purpose of the research conducted under the grant was to investigate various aspects of this environment. A first performance prediction system was developed and evaluated (by comparison both with simulations and with actual systems). The creation of a second, complementary system is well underway.

In addition to the research directly related to the development of performance projection tools, research into new methods for allocation of tasks onto the nodes of parallel processors was conducted. This line of research required the creation of new experimental environments suitable to the study of the resource allocation problems being studied. Lessons learned here are expected to contribute to the extension and generalization of the performance prediction methods which are the primary focus of this research.

This report first summarizes a performance projection method which has reasonable computational complexity and which is able to project the execution time of series-parallel reducible acyclic task systems on concurrent computing systems with an average error of only 1.7%.

The second section reports on progress toward performance projection starting with the source code of the program. Especially encouraging progress has been made in the development of a stochastic cache simulator for use in this system.

The third section discusses research into resource management in concurrent computing systems and the experimental environment that was created to support the research. This section reports on a heuristic approach applied between executions of a parallel program which has been shown experimentally to determine the optimum partitioning of tasks onto resources. The experimental environment created to study and evaluate this method will be described.

The fourth section summarizes preliminary research into a new multiprocessor system architecture which could efficiently support dynamic resource allocation using a combination of hardware and software techniques. This research was a particularly challenging problem. The simulation environment which was created and used on this study will be discussed.

The research conducted under this grant has been reported in five Stanford technical reports and three papers. One Ph.D. dissertation is nearing completion as this report is prepared. These are identified separately in the bibliography.

# 1 Task Level Performance Prediction

Investigator: Victor Mak

This section summarizes research which has resulted in a new method for predicting the performance of an application on highly concurrent computing systems. Section 1.1 states the problem addressed and the objective of this research. Section 1.2 summarizes the new method developed during this research.

The prediction method has been implemented in C and is now running under UNIX[1]. A user's manual of this program and sample input and output files are given in Section 1.3.

## 1.1 Statement of the Problem

The main objective of this research is to develop a computationally efficient and accurate method to predict performance of parallel computations running on concurrent systems. Performance measures that are of interest are the execution time of parallel computation and the utilization of physical resources in the concurrent system.

To determine the execution time of the parallel computation, it is important to consider the entire code rather than a part of it. Utilization of the physical resources in the concurrent computing system is also an important performance measure in order to understand how effectively the system resources are being used. This information can also help the system designer to detect and locate the system bottleneck.

Every performance prediction method has to reckon with a tradeoff between the accuracy of the estimates and the computational efficiency of the method. More accurate estimates often demand more detailed analysis and thus require more time to prepare the prediction. In the early stages of a design, concurrent system parameters are usually only estimated. The performance prediction method used should be computationally efficient to allow the system designer to experiment with a large number of design alteratives in order to pick a few promising ones for further studies. Accuracy of the estimates is of secondary importance in this case, as long as they are good for first order estimates. However, at the final design stage, very accurate system performance estimates will be required. In this case, it is more appropriate to perform a detailed analysis such as prototyping or detailed simulation.

The objective of this research is to develop a computationally efficient performance prediction method to be used in the early design stage. It should be a fast turnaround

---

**[1]UNIX is a trademark of AT&T Bell Laboratories**

tool to help system designers evaluate different design options and to help computer scientists compare the performance of different algorithm structures. The predictions should be accurate enough for at least first order estimates, and errors in the range of 10% to 20% are probably acceptable.

## 1.2 Research results

An accurate and computationally efficient method has been developed to predict performance for a class of parallel computations running on concurrent computing systems. Execution time of the parallel computation and utilization of physical resources in the concurrent system can be estimated using this method.

In this method, two models are used: one to represent the concurrent computing system and the other to represent the parallel computation. Physical resources of the concurrent system (e.g., CPU, memory module, and bus) are modeled as service centers in a queuing network model. Parallel computation is modeled as a task system with precedence relationships expressed as a series-parallel directed acyclic graph. This model includes a large class of computations written using block-structured parallel constructs like fork-join, cobegin-coend, and DOALL. A task corresponds to a sequence of instructions that can be executed independently on a single processor and is specified by its service demand on the system resources. A task can start execution as soon as all its precedence constraints have been satisfied. Tasks executing concurrently will compete with each other for system resources, thus causing contention in the system.

The queuing network and task system models are used as inputs to the prediction method which produces system performance estimates. The method uses an iterative algorithm which consists of two main steps in each iteration. In the first step, the residence time of each task is estimated by considering the amount of overlapping between tasks. In the second step, task residence times are used to reduce the precedence graph to obtain completion time of the task system as well as starting and ending time of each task. The starting and ending times are then used to obtain a new set of estimates of residence times. This iteration continues until both residence times and completion times converge to within some preset tolerance.

In the first step, task residence times are estimated using an algorithm based on approximate Mean Value Analysis but extended to handle programs with internal concurrency. The extension introduces a new quantity called the ***overlapping factor*** which accounts for the amount of time overlap between different task pairs in the computation. In the second step, the precedence graph is reduced by repeatedly applying series

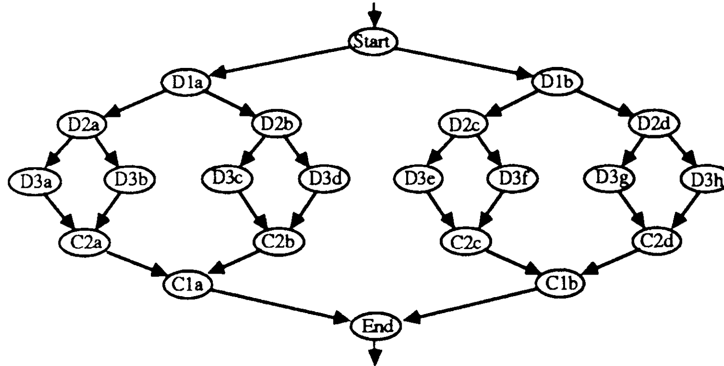


Figure 1: Divide-And-Conquer problem.

and parallel reductions to the nodes of the graph. A new approximate method is introduced to reduce two parallel dependent tasks which have queuing effect on each other's residence time.

This prediction method is very accurate and has been validated against both detailed simulation (Figures **1-4)** and measurements on a Sequent Balance 8000 multiprocessor (Figures 5-8). In the simulation study, one hundred test cases were used to compare the estimates from the prediction method to statistics collected during the simulation runs. Using the simulation statistics as reference, the average error of the predictions was 1.7% with a standard deviation of 1.5%. The maximum error was about 10%. A parallel program was also written on the Sequent Balance 8000 multiprocessor and its execution measured. The estimates of the prediction method also compare favorably with the measurements in this case.

The following figures are typical examples of the predictions from the new method compared to simulation results and to measurements taken on an actual multiprocessor.

The prediction method is very efficient computationally: it has polynomial complexity in both time and space. The time complexity for each iteration is $O(N^2K + N^3)$ and the space complexity is $O(N^2K)$, where N is the number of tasks in the parallel computation and $K$ is the number of physical resources in the concurrent system. Previously the best known methods all had exponential complexity.

The iterative algorithm has a high rate of convergence. For the hundred test cases used, the average number of iterations required to achieve convergence (for 0.1% tolerance) is five, and the maximum number of iterations required is twelve.

### *1.2 Research results*

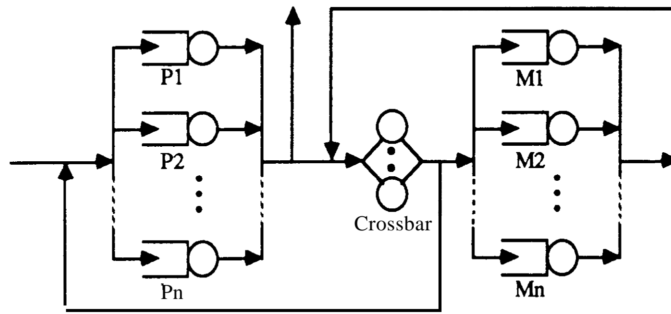


Figure 2: Shared memory multiprocessor to run problem.

| Tasks | Processor | Memory System | Crossbar |
|---|---|---|---|
| *Start, End* | 0.52 | 0.50 | 0.05 |
| *D1a–D3h* | 0.62 | 0.60 | 0.06 |
| *C1a–C2d* | 0.42 | 0.40 | 0.04 |

Figure 3: Service demands.

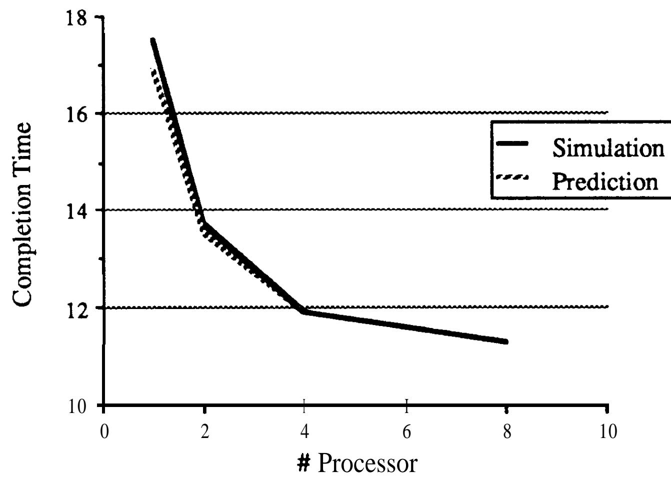


Figure 4: Results of Divide-And-Conquer system.

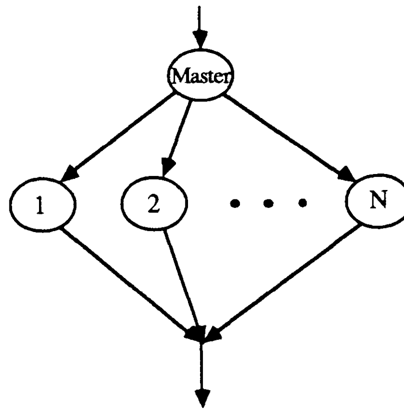


Figure 5: Master-Slave task system for validation experiment.

```
while (rand() > PROB) do
begin
        a ← b;        /* line 1 */
        b ← c;
        c ← d;
        d ← a;

        a ← b;
        b ← c;
        c ← d;
        d ← a;        /* line 60 */
end;
```

Figure 6: Program code of validation experiment.

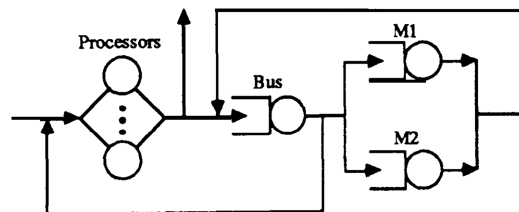


Figure 7: Queueing network model of Sequent Balance ***8000.***

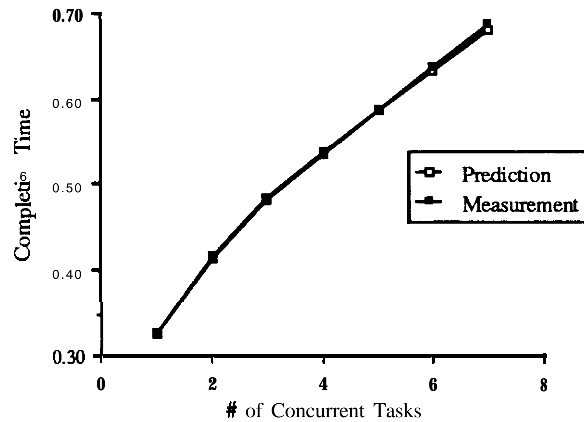


Figure 8: Results of measurement experiment.

The following prediction method developed during this research:

1. Can predict performance for a class of parallel computations which can be expressed as series-parallel task systems.

2. Is very accurate and has been validated against both detailed simulation and measurements on the Sequent Balance 8000 multiprocessor. The average error of the predictions when compared to the simulation statistics was 1.7% with a standard deviation of 1.5%.

3. Is also very efficient computationally. The method has polynomial complexity in both time and space. The time complexity for each iteration is $O(N^2K + N^3)$ and the space complexity is $O(N^2K)$.

4. Is based on an iterative algorithm that has a high rate of convergence. For the hundred test cases used, the average number of iterations required to achieve convergence (for 0.1% tolerance) is five.

5. Introduced new algorithms, namely the overlapping factor and the reduction of precedence graph, which extend Queuing Network Modeling to handle programs with internal concurrency.

## 1.3 TSPP: Task System Performance Predictor

### Description

The Task System Performance Predictor (TSPP) estimates the performance measures of a task system executing on a computing system represented by a queuing network model. The task system and the queuing network model are specified in the input file using the syntax described below. The output of the program consists of performance measures of both the task system as well as the queuing network model.

### Execution Instruction

On UNIX type the following command to the shell:

```
% tspp <infile >outfile
```

"infile" is the input file that specifies the task system and the queuing network model, "outfile" is the file storing the output of the program. There are two optional switches to this program:

```
tspp [-b] [-d]
```

Both switches control the amount and format of the output file, and will be described later.

### Input Format

The input to TSPP is a text file specifying the task system, resources in the queuing network model, and the service demand of each task on each resource. Consider the task system and the queuing network model in Figures 9 and 10, respectively. The input file specifying them is shown in Figure 11.

The input specification consists of three parts:

1. **Resource** definition: Declares the resources in the queuing network model. It starts with the keyword "resource", and is followed by a list of resource declarations. A resource declaration is of the form

```
<name> <- <resource type> [<exp>];
```

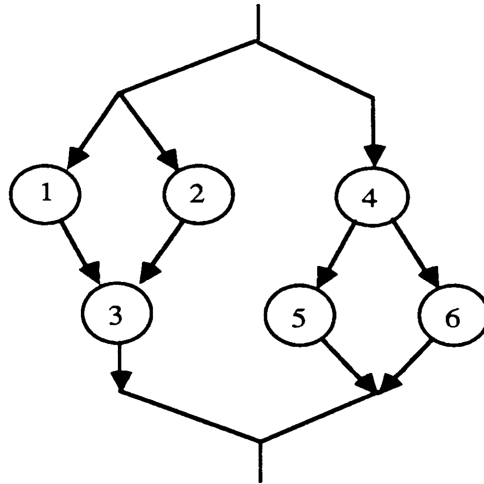


Figure 9: Example task system.

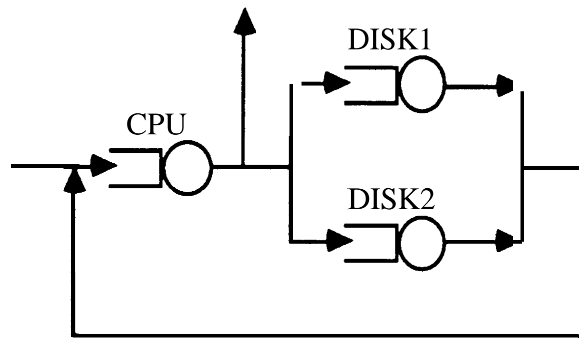


Figure 10: Example queuing network model.

```
% Resource definitions
resource
    cpu <- queuing;
    disk_1 <- queuing;
    disk-2 <- queuing;

% Task definitions
task
    task-1 <- {
        cpu: 0.42;
        disk-l: 0.4;
        disk-2: 0.4; }
    task-2 <- {
        cpu: 0.42;
        disk-l: 0.4;
        disk-2: 0.4; 3
    task-3 <- {
        cpu: 0.62;
        disk-l: 0.6;
        disk-2: 0.6; 3
    task-4 <- {
        cpu: 0.62;
        disk-l: 0.6;
        disk-2: 0.6; 3
    task-5 <- {
        cpu: 0.42;
        disk-l: 0.4;
        disk-2: 0.4; 3
    task-6 <- {
        cpu: 0.42;
        disk-l: 0.4;
        disk_2: 0.4; 3

% Structure definition
structure
    [ { C task-l;
            task-2 ;
        1
        task_3;
      3
      { task,4;
        [ task-5;
            task-b;
        1
      3
    1
```

Figure 11: Example input file.

The name of a resource can be any alphanumeric strings, including "_". Resource type can be either "queuing" or "delay". A "queuing" center is a resource with at least one server and a queue attached to it for waiting tasks. The number of servers is specificed by the value of the expression after <resource type>. If the expression is missing, then the default value of one is used. A "delay" center corresponds to an infinite server in which tasks can obtain service immediately without waiting. This is usually used to model resources that are strictly owned by the task (i.e. no sharing), or resources in which the contention is so low that it can be modeled as a "delay" center.

2. **Task** definition: Declares the tasks and their service demands on the resources of the system. It starts with the keyword "task" and is followed by a list of task declaration. A task declaration is of the form:

```
<task name> <- { <resource name>: <demand>;
                 <resource name>: <demand>;
                 <resource name>: <demand>; }
```

<task name> and <resource name> are any alphanumeric strings including "_". <demand> is the service demand of this task on the resource. It can either be a number or an arithmetic expression using operations +, -, *, and /. The unit of the service demand is arbitrary, it can be seconds, milliseconds, or any unit of time as long as it is used consistently.

3. **Structure** definition: Declares the precedence structure of the task system. It starts with the keyword "structure" and follows by the specification of the series-parallel task system. The syntax of the specification makes sure the task system is series-parallel. {. . .} is for the serial part, and [. . .] is for the parallel part. The task names are ended with a semicolon ";".

Comment lines start with "%". TSPP will ignore the rest of the line including the "%" sign. Errors in the input file will be detected and reported in the stderr. No output will be generated if there is an error in the input file.

### Output Format

When there is no switch to the contrary set in the command line, TSPP writes the following in a tabular format to the standard output:

1. Utilization: the utilization of each resource over the entire execution of the task system.

2. Queue Length: the mean queue length (including the one in service) of each resource over the entire execution of the task system.

3. Arrival Instant Queue Length: this is the average queue length as seen by the task when it arrives at the resource. If the resource is strictly owned by the task, i.e., no other task will compete with it for this resource, the arrival instant queue length will be zero. Otherwise, this length will correspond to the amount of contention the task experiences during its execution interval.

4. Queue Length of Task: the average queue length at the resource contributed by the task during the task's execution interval. This value corresponds to the fraction of time a task spends at this resource. The sum of queue lengths of this task at each resources must be equal to one. This average queue length of a task at a resource can be added to the arrival instant queue length as seen by this task to determine the average queue length of the resource during this task's execution interval.

5. Starting Time: the mean and standard deviation of the starting time of a task. A task can only start after all its ancestors have finished execution.

6. Residence Time: the mean and standard deviation of the residence time of a task. The residence time includes the actual service time at the resources and the queuing time spent waiting for service.

7. Ending Time: the mean and standard deviation of the ending time of a task. This time is equal to the sum of the task's starting time and residence time.

8. Completion Time: the mean and standard deviation of the completion time of the task system. This is the time when all tasks finish their execution.

9. Number of iterations: the number of iterations used by the algorithm to converge to this set of results.

An example output file is shown in Figure **12.**

The following switches control the amount and the format of the output:

**-b** Brief mode. Only the starting time, residence time, ending time, and completion time of the task system will be printed.

-d Dump mode. With this switch on, TSPP will dump the results in a format easily read by other programs for display and analysis. The results are output as text

```
                              Results
                              -------


Resource        Utilization     Queue Length
--------        -----------     ------------

cpu             0.487           0.675
disk-1          0.467           0.637
disk-2          0.467           0.637


                        Arrival Instant Queue Length
                        ----------------------------
task_1          0.513   0.486   0.486
task-2          0.513   0.486   0.486
task-3          0.209   0.197   0.197
task_4          0.366   0.346   0.346
task-5          0.408   0.386   0.386
task-6          0.408   0.386   0.386


                        Queue Length of Task
                          -----------
task-1          0.348   0.326   0.326
task_2          0.348   0.326   0.326
task-3          0.343   0.329   0.329
task-4          0.344   0.328   0.328
task_5          0.348   0.326   0.326
task-6          0.348   0.326   0.326


Task            start           Residence       End
----            -----           ---------       ---

task-1          0.000 (0.000)   1.824 (1.824)   1.824 (1.824)
task-2          0.000 (0.000)   1.824 (1.824)   1.824 (1.824)
task-3          2.635 (1.955)   2.186 (2.186)   4.822 (2.933)
task-4          0.000 (0.000)   2.462 (2.462)   2.462 (2.462)
task-5          2.462 (2.462)   1.700 (1.700)   4.161 (2.991)
task-6          2.462 (2.462)   1.700 (1.700)   4.161 (2.991)

Completion time = 5.999 (2.907)
Number of iterations = 4
```

Figure 12: Example output file.

```
<number of tasks, N>
<number of resource, K>
<utilization of resource 1>


<utilization of resource K>
<queue length of resource 1>


<queue length of resource K>
<arrival instant queue length of task 1 at resource 1>


<arrival instant queue length of task 1 at resource K>


<arrival instant queue length of task N at resource K-1>
Car-rival instant queue length of task N at resource K>
<queue length of task 1 at resource 1>


<queue length of task 1 at resource K>


<queue length of task N at resource K>
<mean starting time of task 1>
<standard deviation of starting time of task 1>
<mean residence time of task 1>
<standard deviation of residence time of task 1>
<mean ending time of task 1>
<standard deviation of ending time of task 1>


<mean starting time of task N>
<standard deviation of starting time of task N>
<mean residence time of task N>
<standard deviation of residence time of task N>
<mean ending time of task N>
<standard deviation of ending time of task N>
<mean completion time>
<standard deviation of completion time>
<number of iterations used>
```

Figure 13: Format of output file at dump mode.

with each value separated by a newline character as in Figure 13. With the **-b** switch on, only the task number, starting time, residence time, ending time, and the completion time will be output.

### 1.4 Conclusions

The prediction method described is very accurate and computationally efficient. It has been validated against both detailed simulation and measurement on a multiprocessor. Average error found in the simulation study was 1.7%, with a standard deviation of 1.5%, and the maximum error found was 10%. The prediction method also has a very high rate of convergence. 90% of the test cases used can achieve convergence within seven iterations.

This material is based on a draft version of the investigator's Ph.D. dissertation, which will be available in December 1987.

## 2 Program Source Level Performance Prediction

Investigator: William ***Lynch***

The overall goal of this project is to predict the performance of concurrent computing systems. The software to be executed determines performance. ***Value analysis*** quantifies this performance effect by analyzing the flow of values in the software prior to actual execution. This section describes on a major component of this flow analysis, the ***stochastic cache simulator.***

### 2.1 Value Analysis

Value analysis models all programs as value transformers. This model may encompass many different programs and programming languages. The model consists of the input values, the desired output values, and the program which specifies a transformation between these input and output values.

This value-transformation model has excellent characteristics for performance prediction. The two most important factors in machine speed are memory accesses and the ALU operations, corresponding exactly with the value theory model, where the values represent memory accesses and the transformations represent ALU operations. The analysis of values and transformations allows performance estimation with the parameters of space and time, both in memory and logic units.

Value analysis examines test programs within the semantics of the language, independent of implementation (including hardware). The performance results are derived without program execution, and are expressed in terms of basic operations.

## 2.2 Stochastic Cache Simulator

Value analysis, which statically analyzes the source code of the program, cannot drive trace-driven cache simulations, because there is no inherent execution of the program, and therefore no address trace can be produced. The execution of the program on some arbitrary machine could produce an address trace for cache simulations, but the choice of an arbitrary machine could affect the performance prediction, particularly with respect to instruction tracing (where the difference between the source and the target instruction set could have an effect) and parallel architectures (where execution order could be different). In addition, experience with the tools of the ***architect's workbench*** shows that trace-driven cache simulations may take hours before results are available.

Thus, value analysis requires an adjunct method to provide fast cache-performance estimates without lengthy simulations or address traces. The stochastic cache simulator satisfies these requirements.

### Concept

A memory access hits in the cache if the location accessed exists in the cache. Whether or not the accessed location exists in the cache depends on two things: the history of the program execution, and the specific parameters of the cache, particularly replacement policy. A conventional cache simulator tracks the contents of a cache over a trace of the memory accesses, essentially a bookkeeping function where every access in the dynamic trace is evaluated. The contents of the cache are known for each access because the entire **exact** history until that dynamic access is known. Rather than examine dynamic accesses, the stochastic cache simulator examines ***static*** accesses, that is, an access as seen in a static analysis of the user code. The stochastic cache simulator examines each possible history of execution, and determines the hit rate for the access by probabilistically summing each histories' hit rate. The probability of this access hitting is given by

$$p_{\text{hit}} = \sum_{h}^{\text{histories}} p_{\text{hit},h}\, p_h$$

where $p_{\text{hit},h}$ is the probability that history $h$ puts the reference in the cache, and $p_h$ is the probability of history $h$ occurring.

Because the stochastic cache simulation is based on static references, rather than dynamic references, results are available much faster than with trace-driven simulations. In addition, the use of static references satisfies one of the constraints placed by value analysis, which cannot generate dynamic-reference traces.

### Histories

Unfortunately, independent evaluation of all possible histories results in exponential simulation time. To make the simulation time more reasonable, the stochastic cache simulator probabilistically combines histories as it traverses the static reference trace. The history for each basic block is determined at the beginning of each basic block through probabilistic addition of the histories of each possible entrance to the basic block, giving

$$H_{\text{block}} = \sum_{e}^{\text{entrances}} H_e p_e$$

where the contents of ***H***, the history, will be discussed in the following paragraphs. This history-combining allows the simulator to pass quickly over the graph of the basic blocks of the program, without deep searches of many histories.

Combining histories is difficult, however, when there is a dependency between the histories to be combined, and the resulting history, as in a loop or recursion. The simulator currently solves this problem iteratively, by simulating the dependent block assuming that the dependent entrance does not exist, then returning to the dependent block with the results from the first simulation. This model relates to what happens in real life, with an initial entrance followed by the rest of the entrances, due to iteration or looping.

Unfortunately, this model also introduces error into the calculation, as demonstrated by the following example.

```
for i = 1 to 100
        if a > b
        then a = i          (block 1)
        else b = i          (block 2)
```

The histories for the loopback combine from block 1 and block 2. For the remaining simulations of the loop, the histories will contain a with some probability, and **b** with some probability (say 0.5 each). Thus, the simulation would show that, based on the initial execution, a would miss half the time, and **b** would miss half the time. However, in a real cache, ***both*** a and b would be in the cache after the first few iterations.

Therefore, a simple, single-iterative method cannot accurately model some program constructs. Currently, this problem is not resolved and is receiving the highest-priority attention in the development of the stochastic cache simulator. Methods under consideration are multiple-iteration (easy, but probably slow), and the introduction of a combinatorial algebra which would allow the solution of the set of linear probabilistic equations describing the histories.

The histories contain at least one thing: the probability of any given value being in the cache. Currently, all values are kept in each history, although future improvements will probably pare this down to only those which could be in the cache. The hit-probability becomes one after the value is accessed, and changes through history-combination and replacement. The simulator combines block-entrance histories by

$$P_V = \sum_{h}^{\text{histories}} p_{v,h}\, p_h$$

for all values $v$.

### Replacement policy

The replacement policy of the cache can add significant complexity to the history, as well as to the execution of the simulation.

A random replacement policy adds nothing, as the replacement is implemented simply by reducing the probability that all other values are in the cache by the probability of each value not being replaced times the probability of a miss,

$$P_V = P_V \frac{l-1}{l}(1 - p_x)$$

for all values v except x, where x is the value being referenced (therefore $p_x = 1$), and $l$ is the number of lines in the cache.

A least-recently-used (LRU) replacement policy adds complexity to the histories, as some notion of the age of each value must be retained. The most straightforward and accurate method uses a ***stack model*** of the cache, where the values in the cache are lined up in order of age. For stochastic caches, a fully-represented stack comprises a table of the probability versus age. Unfortunately, this table can become quite large, and each entry in the table must be updated on each access, slowing the stochastic simulation considerably.

To overcome the speed and size limitations of a fully-represented stack,, the current implementation of the stochastic cache simulator uses an ***expected*** *value* method for LRU

replacement, where the expected value of distance from the top of the stack, given that the value is in the cache, is kept for each value, and denoted $E[d|P]$. These expected distance values are combined at basic block entry with the equation

$$E_v[d|P] = \frac{\sum_h^{\text{histories}} E[d_{v,h} | P_{v,h}] p_{v,h} p_h}{\sum_h^{\text{histories}} p_{v,h} p_h} = \frac{\sum_h^{\text{histories}} E[d_{v,h} | P_{v,h}] p_{v,h} p_h}{p_v}$$

The simulation of LRU replacement sets $p_v = 0$ whenever $E[d_v|P_v]$ is greater than the cache size, which corresponds to replacement in a real cache. The simulation sets $E[d_v|P_v]$ to 0 when v is accessed, which corresponds to the movement to the top of the LRU stack in a real cache. The age $(E[d|P])$ of all values except v is increased on an access of v by following these two rules:

- A hit by a value takes the value to the top of the stack. All values which were lower on the stack get older. Already higher (older) values are not affected.
- A miss by a value ages all values on the stack.

Thus, the formula for the new $E[d|P]$'s for all values v except the accessed value x is given by

$$\begin{aligned} E[d_v|P_v] &= E[d_v|P_v] + p_x + (1 - p_x) = E[d_v|P_v] + 1, && \text{for } E[d_v|P_v] <= E[d_x|P_x] \\ &= E[d_v|P_v] + (1 - p_x), && \text{otherwise .} \end{aligned}$$

The expected-value method sacrifices some accuracy over the full-table method in that the ages of values in the cache are only averages of the histories, rather than complete representations. This compression of information results in some incorrect replacement and non-replacement of values during the simulation. The expected value method thus greatly reduces simulation time and size, with some imprecision. The effects of the imprecision have not been fully investigated, but seem to be small.

Currently, no replacement policies other than LRU and random are implemented.

### Results

The stochastic cache simulator has been implemented and compared to actual cache simulations running on the same host and benchmarks. Figures 14 and 15 show the miss rate results from one benchmark for copy-back, LRU caches.

Clearly, the results from the stochastic simulation show the same trends as the full simulation. However, for some cache sizes, particularly in the transition range, there

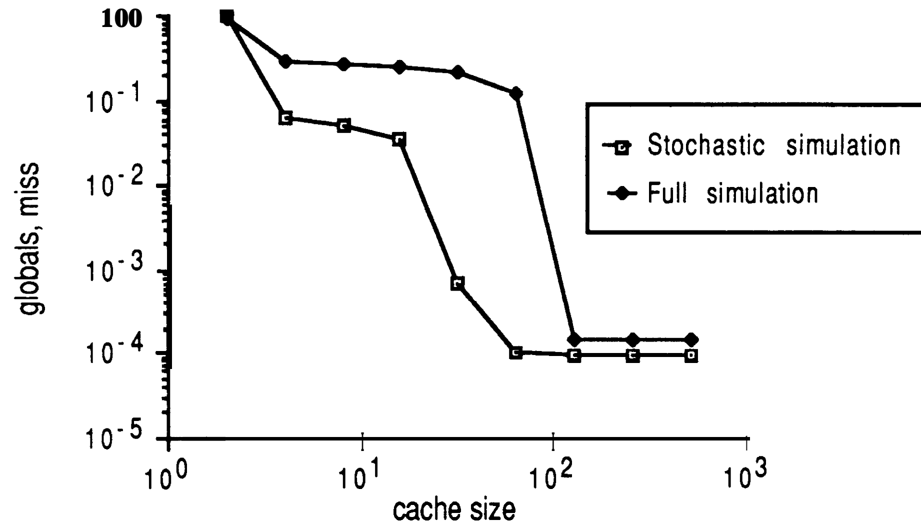


Figure 14: Global variables.

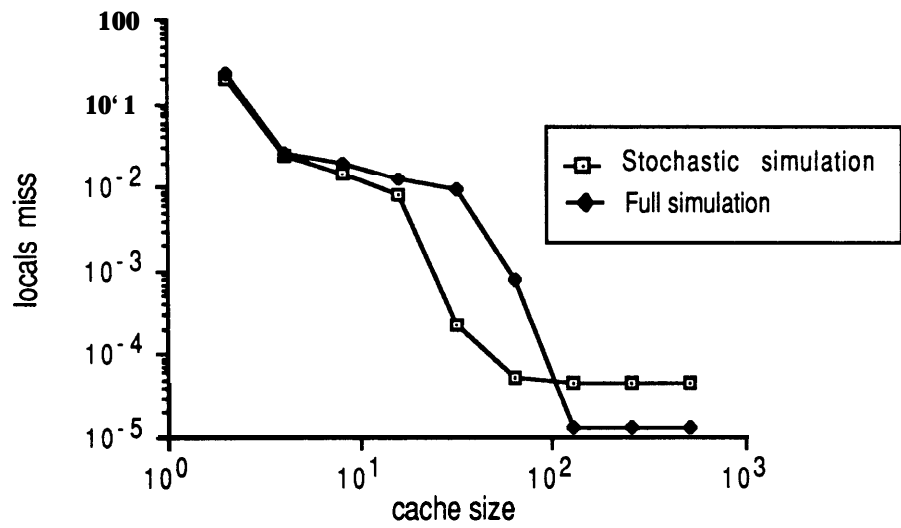


Figure 15: Local variables.

can be significant differences in the results. These differences appear to arise from the current implementation's problem with loops and recursion, and may disappear in future versions.

The stochastic cache simulator executes about 100 time faster than the full simulation. This comparison is, however, difficult to make because the full simulator uses the ***inclusion*** principle of LRU replacement to simulate 10 caches at once. For other replacement strategies, the speedup of the stochastic cache simulator would be even more dramatic.

### Problems

Several problems plague the current implementation of the stochastic cache simulator. The most serious of these is the looping/recursion problem, which is believed to cause the disparity between actual and stochastic simulations in figures 14 and 15. The solution to this problem is currently under investigation.

The generation of the history-combining probabilities poses another, although less serious, problem (essentially, these are branch probabilities). Value analysis may be able to generate most of the probabilities accurately without further assistance. Partial executions of the program under study could generate all of the necessary data.

Dynamic references such as pointers pose a more serious problem. Because the value analysis/stochastic cache simulator combination does not execute the test program, the actual value accessed by dereferencing a pointer is not known. For the moment, the stochastic cache simulator has a parameter which merely sets the probability of a dynamic reference missing to some fixed value. For the test programs simulated thus far, the value of this parameter has had virtually no effect on the miss-rate results.

Simulating caches with line sizes larger than one value presents another, though small, problem. The stochastic simulation method still works, but on a Line basis, rather than a value basis. Either value analysis or some external program must place the values in lines for the simulation. This should not be a difficult problem to overcome.

### Advantages

Despite the preceding problems, stochastic cache simulations have several advantages. First, they are very fast, allowing a large number of potential caches to be evaluated quickly. Secondly, they are in general much smaller than full cache simulations, although the size depends on the static test-program size. Thirdly, they do not necessarily require test-program execution.

Another potential advantage is that the simulation should be easily vectorizable, consisting of mostly floating-point operations, which would allow stochastic cache simulations to run very fast on vector supercomputers.

### 2.3 Conclusions

Stochastic cache simulations show promise for the fast testing of a large number of cacheing strategies for programs. Some limitations exist, such as problems with dynamic variables, which restrict the range of programs to which the method may be applied. However, because of its speed, size, and input requirements, the stochastic cache simulator is ideal for use with value analysis.

## 3 Resource management for concurrent systems: a heuristic approach

**Investigator: Jerry Yan**

### 3.1 Introduction

While distributed architectures promise to deliver orders of magnitude speed-up, the effective use of such systems still depends critically on its resource management system to properly trade-off communication loss and concurrency gain, to adapt to behavioral variations among and within programs, and to take advantage of specific hardware characteristics [NYM 87]. Research is being conducted to determine how distributed computations can be mapped onto multiprocessors so as to minimize execution time. The class of programs being considered here, falls under a subset of the Actor programming paradigm [Agha 85]. A computation is expressed as a collection of autonomous computing agents known as ***players*** [Yan 86b]. They interact with one another via message passing. When a player receives a message, it may perform user-programmed computations; send/wait for specific messages; or create new players. The multiprocessors being considered consist of homogeneous processing elements (or sites) connected via physical communication links. The responsibility of the resource management system in this context can be formulated as a placement problem: "Where should each player be placed during program execution so that execution time is minimized?". This problem is NP-hard-even for well structured programs.

Many approaches proposed were based on abstract machine/program models (e.g., graphical representations). In many cases, "artificial metrics" (such as execution costs

[Lo 81] or mapping cardinality [Bokhari 77]) were the only means to evaluate the performance of these approaches. This reaffirms the difficulty of the problem and the need for a fast turn-around experimental environment to represent the operation of concurrent systems. In fact, one of the most difficult issues to be addressed in this context involves choosing an appropriate level of abstraction-resorting to neither lengthy instruction-set level simulations nor general stochastic models.

The ***Axe*** experimentation environment was designed to facilitate such investigations at the operating system level using discrete-time simulation [YL 86a]. Instead of relying solely on analysis of program text, Axe directly executes the program model, and collects run-time statistics which are used to characterize individual players and their interaction with one another. Resource utilization bottlenecks are also reported, enabling resource management adjustments to be carried out.

Axe is fast when compared with instruction-level simulation. At the same time, the model preserves the key characteristics of program behavior. These characteristics are important to evaluating dynamic partitioning strategies.

## 3.2 Axe experimental environment

The ***Axe*** experimentation environment consists of a set of software tools that enables the following tasks to be performed in an integrated environment:

- ***Computation model specification.*** By analyzing program text, the researcher expresses the program using a behavior description language, or ***BDL*** [Yan 86b]. BDL enables the construction of program models at many levels of abstraction. These models preserve the pattern of communication/synchronization between players and their computational requirements while at the same time, allow efficient simulation to be carried out. BDL models, as opposed to actual program text, are simulated.

- ***Execution environment specification.*** Various parameters that describes the multiprocessor as well as operating system algorithms and overhead (such as the number of sites, connection topology, routing algorithms, processor speed, band-width of the communication links, context switch overhead, etc.) can be modified.

- ***Simulation.*** Data is collected automatically for performance evaluation. The user sits back to observe improvements being made as Axe "learns" about program behavior and "adapts" to it.

- ***Experimentation.*** The researcher is able to study various issues in parallel processing: problem formulation, hardware architectural issues, matching machines and programs, and operating system level algorithms.

## 3.3 Resource Management Studies on Axe

The approach proposed here involves an iterative refinement framework which ***improves*** the execution speed for a given application on a given machine by proposing perturbations to a given mapping:

1. start with some initial configuration.
2. execute (or simulate the execution of) the program on the target machine.
3. gather data during program execution (or simulation).
4. analyze the data and propose a new mapping (configuration).
5. go to step 2 until no new mapping can be generated.

Because resource management considerations are carried out after (or, in fact, in between) program executions, this technique is known as ***post-game analysis.***

At the heart of the post-game analysis (i.e., step 4) is a rule-based system which examines the performance of a given mapping and opportunistically proposes changes to shorten execution time [Yan 87c]. Its operation is divided into two phases:

1. ***Multiple-perspective performance evaluation and prediction.*** The performance of a given mapping is analyzed based on the data collected during program execution. Heuristics which represent independent lines-of-reasoning concerned with particular aspects of the execution environment (such as communication relationships between players or resource contention at certain sites) are applied. All the perturbations proposed and their potential impact on execution time are gathered at the end of this phase.
2. ***Constraint satisfaction and propagation.*** These proposals are then prioritized and combined (or eliminated) based on their projected impact on execution time as well as the heuristic which proposed them. A new mapping is then generated, ready to be used next time the program is executed.

## 3.4 Current Status

- The effectiveness of simple heuristics was tested against simple pipeline-like computations on ensemble architectures. The results are promising-attaining 95% optimal speed-up in some cases [Yan 87a].
- A strategy which involves multiple heuristics was tested against two kinds of parallelism: pipe-line vs. divide-and-conquer types [Yan 87b]—attaining 96% optimal speed-up in many cases, taking only a few iterations. Detailed measurements also revealed that the parameters used by the heuristics correlated well the execution time.

  Experiments on a larger scales (with 200+ players) have been carried out. They, too, have demonstrated a similar level of success [Yan 87c].

## 3.5 Conclusions and Future Work

The study of resource management strategies for concurrent systems requires the use of significant test cases. The development of a detailed model of a complete application seems unreasonable. In the case reported here, an abstraction of the application program is created using BDL models. This abstraction is then "executed" on an operating-system level model of the multiprocessor. The Axe simulation environment enables studies in applications formulation, multiprocessor organization/architecture, and instrumentation/representation of concurrent systems to be carried out.

Axe was able to help verify that resource management for concurrent systems involves many (possibly conflicting) goals. The resource management strategy proposed here ("post-game" analysis) has demonstrated both the ability to trade-off these goals and a level of success which is very close to theoretical optimum and outperforms approaches based on random search. Axe will be used to carry out experiments with a wider range of test cases to study the capabilities and limitations of this approach.

# 4 Resource management for concurrent systems: a system architecture approach

Investigator: **Tin Fook** *Ngai*

## 4.1 Evaluation of a New Multiprocessor System Architecture

In the exploration of high-performance multiprocessor systems, dynamic resource allocation is an important, but very difficult, issue. Dynamic resource allocation has the advantage of directly extracting run-time characteristics and is expected to achieve better resource allocation than static resource allocation (especially for computations whose run-time behavior is highly data-dependent and thus less predictable). The major difficulties of dynamic resource allocation lie in the exploitation of run-time parallelism and in the minimization of run-time resource management overhead. A new multiprocessor system architecture which integrates considerations of processor structure and program structure has been proposed to alleviate these difficulties [Ngai86, NYM87]. In the proposed system, mapping between the processor structure and the executing program is established dynamically at multiple levels of instantiations. Due to the intricacy of run-time interactions between program execution and resource management, performance evaluation of such system is more complicated than evaluation of systems that employ static resource allocation.

In this part of the study, simulation experiments have been conducted to evaluate the proposed system. In order to obtain fast but fairly reliable evaluation over a wide range of program characteristics, the simulated programs are synthetic programs generated from different stochastic program behavior models. The proposed system is evaluated with respect to different system configurations and different program behavior models. Simple but instructive dynamic resource allocation algorithms are used throughout these experiments. The results of this preliminary study indicate that with the proposed system architecture, the complexity of dynamic resource allocation could be significantly reduced while achieving reasonable effective resource allocation. The following subsections describe the proposed system architecture, the evaluation method and the results of the study. Further details of the proposed system and the preliminary study can be found in [Ngai86].

## 4.2 The Hierarchical Multiprocessor System Architecture

In the proposed hierarchical system architecture, both the processor structure and the program structure have a similar hierarchical structure. Mapping between the struc-

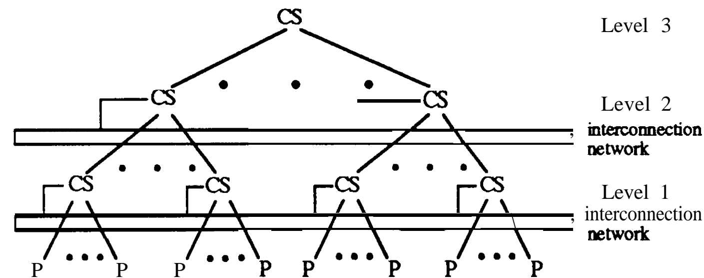

Figure 16: A three-level hierarchical multiprocessor system.

tures is done on a level-to-level basis. This approach reduces the complexity of dynamic resource management. Hardware control units are coupled with data memories and are distributed over the hierarchical processor structure. This allows distributed and efficient resource management. The system architecture and the program execution model are sketched below. Detail descriptions of the system architecture and its operation are given in [Ngai86].

Figure 16 shows the processor structure of the proposed hierarchical system. There are two kinds of local ensembles: processors (P) and control subsystems (CS). Processors may have private memories or caches. Control subsystems contain processing control units and data memories. The processing control units are responsible for dynamic resource management, network communication and input/output. These control subsystems are distributed over the structure in hierarchical levels. Each control subsystem (except those at the lowest level) is directly connected to a specified number of control subsystems at the next lower level. Every control subsystem at the lowest level is connected to a number of processors. In order to support efficient resource sharing, control subsystems at each hierarchical level are interconnected by a separate interconnection network.

Figure 17 illustrates the general structure of the machine programs. Machine programs are represented as hierarchies of labeled dataflow graphs. A labeled dataflow graph is a directed graph in which every node is labeled by a function name and every edge is labeled by a data object instance. (Control signals are treated uniformly as data object instances.) A function node is said to be executable when corresponding data object instances are available on all its input edges-the dataflow firing rule. The number of hierarchical levels of the machine program is made the same as that of the processor structure at compilation time. At each level except the lowest (ground level), a function node represents either a labeled dataflow graph at the lower level or a primitive control function. Data objects are aggregates of data objects defined at the next

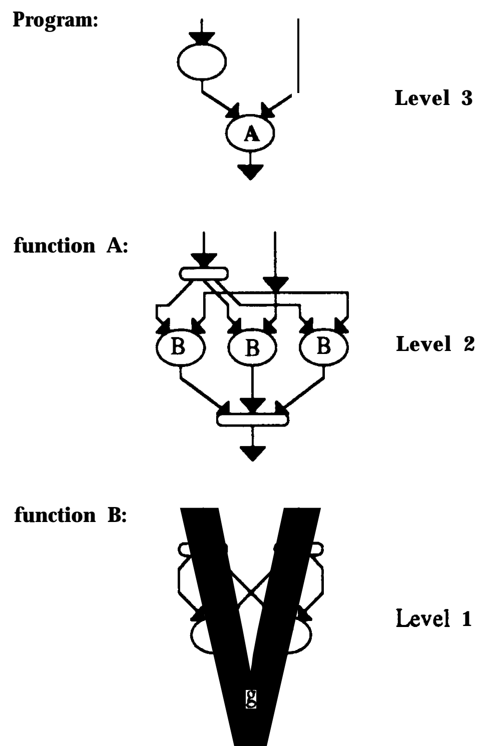


Figure 17: A machine program of three hierarchical levels.

lower level. The ground level dataflow graphs contain only elementary functions and primitive data object instances directly supported by the processors.

Program execution on the hierarchical multiprocessor system proceeds as follows.

- As program execution begins, the control subsystem at the top level loads in the top-level machine program.
- As soon as the dataflow graph is loaded into the control subsystem, the dataflow firing rule is used to determine any executable function nodes. For each executable function node, a task packet is generated and made ready for assignment to a control subsystem at the next level.
- When a control subsystem at the next lower level is able to take more tasks (i.e., current load is low), it grasps a task packet (if there is any) from its parent control subsystem.
- Once a task packet is assigned to a control subsystem, the control subsystem loads in the corresponding next lower-level function program.
- This process continues through all levels of the system. At the ground level, functions are directly executed by the attached processors.
- Upon completion of a function execution in a control subsystem, the resulting data is passed back to the parent control subsystem and the subsequent firing (in the parent control subsystem) is initiated. In the case when the resulting data is defined at the current program level, the data is stored in the local memory and a global reference is returned.
- When the current load in a control subsystem is low and there are no task packets in the parent control subsystem, resource reallocation is initiated. Task packets migrate between control subsystems at the same level via the interconnection network to achieve load balancing.

## 4.3 Evaluation and Results

Since performance of the proposed system directly depends on the run-time interactions between the executing program and the dynamic resource allocation system, system evaluation is based on simulations at the level of detail where these run-time interactions become visible. A software simulator which explicitly implements the above program execution model and the dynamic resource allocation algorithms was constructed to serve as the basic vehicle of the preliminary study.

***Parameters:*** Probability distribution functions of

- $Nsize$ – size of a function dataflow graph (any program level)
- $N_{df}$ – number of executable tasks per function graph per dataflow-firing
- $T_{et}$ – execution time of an elementary function

***Program behavior:***

- When an elementary function is executed, it takes $T_{et}$ time units.
- When a function begins execution, it has $N_{size}$ tasks to be executed. It also occupies $N_{size}$ space units in the control subsystem.
- At each dataflow-firing, $N_{df}$ tasks become executable. When the number of tasks to be fired is less than $N_{df}$, all of these tasks become executable.
- When all tasks of a function graph are executed, the function completes execution.

Figure 18: Program Behavior Model.

The proposed hierarchical system architecture exhibits an interesting property: Any system with more than two levels can be approximated as a two-level system. Thus results on two-level systems can be used to project performance of higher level systems. Based on this observation, this preliminary study has concentrated on two-level hierarchical systems.

In order to have fast and fairly reliable evaluation of the proposed system over a wide range of program characteristics, stochastic program behavior models are used to regenerate appropriate execution profiles for different classes of parallel programs. Synthetic programs are generated from each program behavior model and are executed on the simulator to give performance data. Figure 18 gives a general description of the stochastic program behavior model.

Performance of the proposed system is evaluated with respect to different system sizes and different program models. System parameters (like scheduling time and network transfer time) are estimated from typical hardware appropriate for a high performance multi-microprocessor systems. The granularity of computation is between medium and fine.

The performance measure used in the evaluation is average execution time with

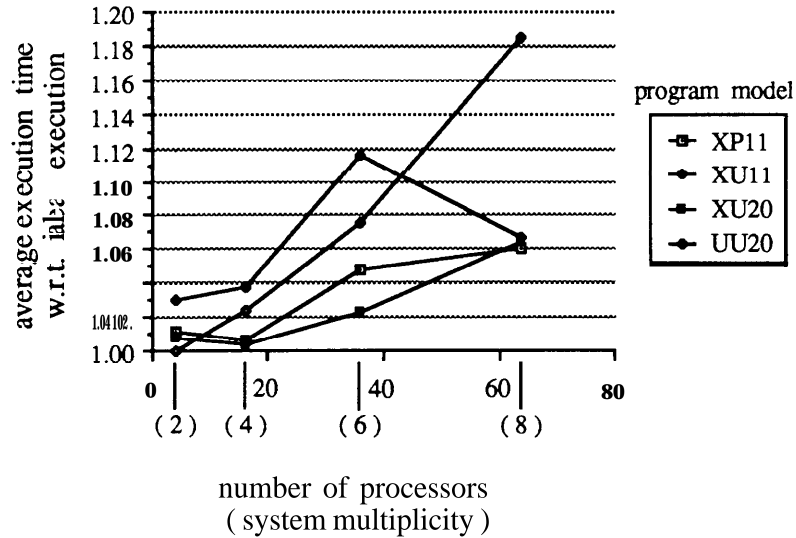


Figure 19: Average execution time with respect to ideal execution.

respect to ideal execution. Ideal execution refers to program execution on an ideal multiprocessor system that has the same number of processors and does not incur any overhead due to resource management. Executable tasks are assigned to any available processor under a first-ready-first-scheduled discipline.

Figures 19 and 20 summarize the simulation results. The four program models used cover a variety of program characteristics: from a large amount of parallelism (UU20) to a small amount (XU11), from non-uniform elementary functions (XU20) to uniform ones (UU20), and from totally random task dependency pattern (XU11) to less random pattern (XP11). How various program characteristics and resource management components affect the system performance are discussed in detail in [Ngai86]. The simulation results show clearly that the proposed hierarchical system architecture supports low-overhead and effective dynamic resource allocation. By using resource allocation algorithms as simple as those used in this preliminary study, nearly ideal dynamic resource allocation can be achieved for systems with small system multiplicity (e.g., fan-out of four) or for programs that have non-uniform elementary tasks and less random task dependency pattern (such as XP11 and XU20). For systems with large system multiplicity and programs that have either uniform elementary tasks or totally

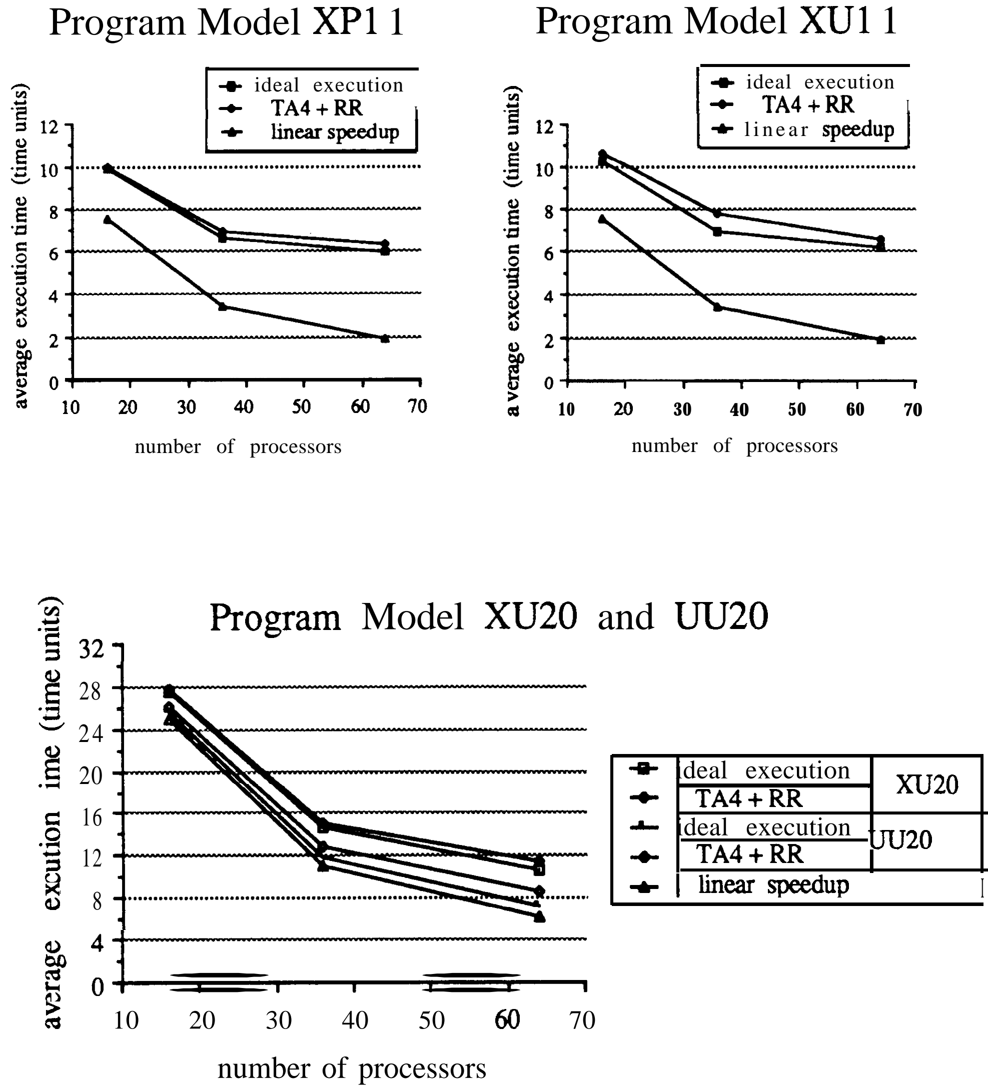


Figure 20: Actual execution time.

random dependency pattern, the gain in exploiting parallelism considerably outweighs the resource management overhead incurred.

## 4.4 Experience with Experimental Environments

When creating experiments used to study dynamic systems (like the one being studied here), the major problems found are:

- how to manage the complexity of dynamic interactions between various hardware and software system components,

  how to define important dynamic instances that are necessary in understanding the system behavior,

- how to record and present all these instances, and

- how to support easy and quick modification to some system components.

A friendly, versatile, and quick-to-build experimental environment is found highly desirable in solving these problems. The provision of such an environment not only significantly reduces the amount of effort spent in creating experiments, but also allows better solutions.

The following paragraphs summarize lessons learned in this study and suggest a few desirable features that an experimental environment should have for studies of dynamic systems.

### Effective use of abstraction models

The study of dynamic systems usually relies on simulations due to the dynamic nature of the system and the lack of satisfactory analytical tools. However, full-scale simulations are very expensive and time consuming. In this study, the use of abstraction models (both system and programs) are found to be very effective in reducing experiment time while important events can still be observed. In particular, when there are few (or no) real programs available, the use of abstract program models avoids substantial work in developing and characterizing test programs, and allows quick and thorough evaluations of new systems proposed. Problems with abstract models are the validity of the model and whether the abstractions are appropriate for the study.

### Powerful and flexible simulation tools

Dynamic components (such as control schemes or communication protocols) are always objects of interest in the study of dynamic systems. These components are usually observed closely in experiments and are subject to frequent modifications (for comparison and evaluation purposes). Simulation tools need to support easy modification of these components and close observation of their dynamic behavior. Validation of the correctness of the implementation of these dynamic components is always a major concern in setting up the experiments. Parallel program debugging is subtle and difficult. Powerful, interactive debugging tools are highly desirable. The debugging tools should be integrated into the simulation environment to allow quick and effective debugging.

### Versatile study aids

Dynamic system behavior is difficult to analyze. The study of dynamic systems usually goes through the following cycle: observation-interpretation and hypothesis-validation. Since observables of a dynamic system are numerous and complex, the task to draw intuitive interpretation or hypothesis from the observation is difficult without any study aids. In this study, activity traces are recorded during simulation and are used to identify situations where performance is poor (for example, when system load distribution is poor). However, trace interpretation is still a demanding task. Interactive, visual aids are highly desirable. Useful study aids can be: pictorial representation of observables, walk-through of the change of the system at any selected time frame, replay of past events, and alternative statistics generation from raw data. A friendly interface to these study aids is necessary.

# 5 Sponsored publications

Papers and reports about work sponsored in whole or part by this contract:

[FMM87] Michael J. Flynn, Chad L. Mitchell, and Johannes M. Mulder, "And Now a Case for More Complex Instruction Sets," IEEE Computer 20(9):71-83, September 1987.

[Mak86] Victor W. K. Mak, "Queueing Network Models for Parallel Processing of Task Systems: An Operational Approach," Computer Systems Laboratory Technical Report CSL-TR-86-306, Stanford University, September 1986.

[Mak86a] Victor W. K. Mak, "A Survey of Concurrent Architectures" Computer Systems Laboratory Technical Report No. CSL-TR-86-307, Stanford University, September 1986.

[Mak87] Victor W. K. Mak, "Performance Prediction of Concurrent Systems," Ph.D. dissertation, Stanford University, December 1987 (expected).

[Ngai86] Tin-Fook Ngai, "Dynamic Resource Allocation in a Hierarchical Multiprocessor System-A Preliminary Study," Computer Systems Laboratory Technical Report No. CSL-TR-86-310, Stanford University, October 1986.

[NYM87] Tin-Fook Ngai, Jerry C. Yan, Victor Mak, Stephen F. Lundstrom and Michael J. Flynn, "Mapping between Parallel Processor Structures and Programs," Proc. Twentieth Annual Hawaii International Conference on System Sciences, January 1987, pp. 172-181.

[Yan86a] Jerry C. Yan and S. F. Lundstrom, "AXE: A simulation environment for actor-like computations on ensemble architectures," Proc. 1986 Winter Simulation Conference, pp. 424-429, Washington DC.

[Yan87b] Jerry C. Yan, "Managing and Measuring Two Parallel Programs on a Multiprocessor," Computer Systems Laboratory Technical Report CSL-TR-87-333, Stanford University, June 1987.

[Yan 87c] Jerry C. Yan, "Multi-Perspective Performance Evaluation and Prediction of Concurrent Execution Environments," Computer Systems Laboratory Technical Report (in preparation).

# 6 Other references

[Agha 85] Gul A. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems," PhD Thesis, Massachusetts Institute of Technology, AI Lab., TR 844.

[Bokhari 77] S. H. Bokhari, "Dual Processor Scheduling with Dynamic Reassignment ," IEEE ***Transactions on Software Engineering*** Vol. SE5, No. 4, pp. 341-349, July, 1979.

[Lo 81] Virginia Mary Lo, Jane W. S. Liu, "Task Assignment in Distributed Multiprocessor Systems," IEEE ***Pmceedings of the 1981 International Conference on Parallel Processing*** pp. 358-360, August 1981.

[Yan86b] Jerry C. Yan, "Parallel Program Behavior-Specification & Abstraction Using BDL," Computer Systems Laboratory, Technical Report No. CSL-TR-86298, Stanford University, August 1986.

[Yan87a] Jerry C. Yan, "Post-Game Analysis- A n Initial Experiment for Heuristic-based Resource Management in Concurrent Systems," Computer Systems Laboratory Technical Report No. CSL-TR-87-314, Stanford University, February 1987.