COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-2192

Ames GRANT
IN-62-CR
158861
43 P.

Dynamic Resource Allocation in a Hierarchical Multiprocessor System
A Preliminary Study

Tin-Fook Ngai

Technical Report: CSL-TR-86-310

October 1986

The work described herein was supported by NASA Ames Research Center under contract NAG 2-248 and using facilities provided by NAGW 419.

Dynamic Resource Allocation in a Hierarchical Multiprocessor System
A Preliminary Study

by

Tin-Fook Ngai

Technical Report: CSL-TR-86-310

October 1986

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Abstract

In this report, an integrated system approach to dynamic resource allocation is proposed. Some of the problems in dynamic resource allocation and the relationship of these problems to system structures are examined. A general dynamic resource allocation scheme is presented. A hierarchical system architecture which dynamically maps between processor structure and programs at multiple levels of instantiations is described. Simulation experiments have been conducted to study dynamic resource allocation on the proposed system. Preliminary evaluation based on simple dynamic resource allocation algorithms indicates that with the proposed system approach, the complexity of dynamic resource management could be significantly reduced while achieving reasonably effective dynamic resource allocation.

Key Words and Phrases: Dynamic Resource Allocation, Hierarchical Program Graphs, Hierarchical Systems, Load Sharing, Mapping, Multiprocessor Systems, Program Behavior Models, Resource Reallocation, Run-time Resource Management.

RESOURCE ALLOCATION
MULTIPROCESSING COMPU[illegible]
SYSTEMS INTEGRATION
[illegible]

[illegible] PERFORM[illegible]
AL[illegible]
[illegible]
RE[illegible]

Acknowledgments

I would like to thank Professor Stephen Lundstrom for his interest in this work. His valuable comments and suggestions greatly improve the readability of this technical report. I would also like to acknowledge Microelectronics and Computer Technology Corporation (MCC) for permitting the use of CSIM.

Table of Contents

List of Figures

List of Tables

1. Introduction

One of the central problems in multiprocessor system design is how computing resources are allocated to a computation to achieve minimum execution time. This resource allocation problem has proven difficult both in theory and in practice. Conceptually, resource allocation realizes a mapping between program structures and the processor structure. Such a mapping can be established either before program execution (*static resource allocation*) or during run-time (*dynamic resource allocation*). It is well-known that static multiprocessor scheduling is *NP*-hard. Static resource allocation must rely on good heuristics *and* good estimation of dynamic characteristics of program behavior to achieve satisfactory mapping. But for computations whose dynamic behavior are highly data-dependent, good estimation of program behavior is very difficult, if not impossible. In this case, even with the best available heuristics, static resource allocation is unlikely to be satisfactory.

Dynamic resource allocation has the advantage of directly extracting run-time characteristics and is expected to achieve better resource allocation than static resource allocation. Dynamic resource allocation is also the only prospective solution for dynamic computations. However, progress in dynamic resource allocation is largely hampered by complexity of the problem. System behavior is more difficult to understand and to analyze in the presence of dynamic resource management. Dynamic resource management itself may be complex and inefficient, and affects overall system performance. In order to alleviate these difficulties, both hardware and software system supports are necessary.

Some examples of system support of dynamic resource allocation have been described in the literature.

- Dataflow machines [Denn80, GuKW85] support dynamic resource allocation explicitly. In the *dataflow model of computation*, tasks (or operations) which have all their operands available are scheduled for immediate execution on any available appropriate resource in a pool of resources.
- In the DDM-1 dataflow machine [Davi78], the process tree was mapped

dynamically onto the processor tree. A process, which was created in a processor, is either assigned to a subordinate processor or to the processor which created the process. But the rigid tree structure did not allow dynamic load balancing.

- The ZAPP system [BuSl81] employed the indirect r-n-cube network of processors to dynamically create a virtual tree of processors to match the process tree. This system proposed an approach to load balancing where an available processor might steal a pending process from its neighbors.
- The ALICE machine [DaRe81] supported an abstract architecture in which a common pool of processes was accessed by a collection of independent processing agents.
- The AMPS project [KeLP79] proposed a control tree structure for task distribution and load balancing. Tasks were invoked at the leaves and were distributed to other leaves via the control tree. Every node in the control tree also moved tasks between its subtrees to maintain load balancing.
- The Cedar project [GKLS83] developed a macro-dataflow architecture in which the global control unit scheduled and assigned compound functions (chunks of program) to processor clusters in dataflow fashion.
- The Rediflow project [KeLT84, LiKe86] developed a demand-driven load balancing scheme. A gradient plane was established by the demands of idle processors, and excessive tasks were transfered to the nearest idle processor via the gradient plane.
- In [BrOP86], an architecture which supported a two-level dynamic scheduling scheme was proposed for medium-grain computations. The first level assigned tasks to processing units on a completely random basis, and the second level corrected load imbalances.

Each of the above approaches supports only a single dynamic resource allocation either in one way or in the other. When a specific program execution model (e.g., the dataflow model) is adopted, the processor structure does not support decentralized resource management or dynamic load balancing; when a specific decentralized control structure is provided, resource allocation is restricted to some fixed granularity of computation; when a dynamic load balancing scheme is proposed, the program execution model only assigns new tasks to their creation sites regardless of current load situation. Due to the complexity of the dynamic resource allocation problem, most subproblems tend to be interrelated. When one difficulty is alleviated, another difficulty may be aggravated. Without balanced, thorough system supports, overall improvement in system performance by dynamic resource allocation will still be limited.

In this research, a new integrated system approach to dynamic resource allocation has been investigated. An extensible, hierarchical system architecture which provides both hardware and software supports to dynamic resource allocation has been defined. A preliminary study has been conducted to evaluate the proposed system architecture with respect to different dynamic resource management strategies. While a particular hierarchical multiprocessor system has been hypothesized, this study has been focused on the dynamic allocation of resources during execution and on how the various resource management components affect the overall system performance. The results of this preliminary study have shown that the proposed system architecture would accomplish effective dynamic resource allocation.

The following sections present a detailed account of this research. Section 2 examines general problems in dynamic resource allocation and their relationships to system structures. A general dynamic resource allocation scheme is introduced here. The discussion provides the rationale behind the proposed integrated system architecture. Section 3 describes the proposed hierarchical system architecture and the program execution model. The rest of the report gives details of the preliminary study. Section 4 describes the system and program models being used in the preliminary study. Section 5 gives the dynamic resource algorithms under study. Section 6 describes the simulation experiments and their results. Section 7 concludes the study and suggests possible future work.

2. Dynamic Resource Allocation and System Structures

The two important research questions in dynamic resource allocation are how to exploit maximum parallelism in both the processor structure and program and how to minimize run-time resource management overhead. The first problem is fundamental to resource allocation since it would allow significant improvement in program execution time. The second problem is also fundamental since run-time resource management may become complex and inefficient in an attempt to exploit maximum parallelism during run-time. The tradeoff between solutions to these two problems is a critical issue in dynamic resource allocation design. Minimization of the overhead in resource managment may result in considerable unexploited parallelism.

2.1. Exploitation of Parallelism

The basic principle of parallelism exploitation is to match *hardware parallelism* and *instantaneous program parallelism*. *Hardware parallelism* is the maximum number of concurrent computations supported by the processor structure. *Instantaneous program parallelism* is the number of concurrently executable units of computation at an instant (depending on the granularity of computation of interest). Effective resource management can be considered to be an instantaneous mapping problem - detect instantaneous program parallelism available and map it onto the available hardware parallelism. The key issues are:

- How are hardware parallelism and instantaneous program parallelism detected efficiently?
- What is the best resource allocation (mapping)?
- Can more parallelism be exploited if the hardware parallelism and instantaneous program parallelism are not matched?

Run-time information must be gathered to determine the current hardware parallelism and program parallelism. For large multiprocessor systems, this essential information gathering process may become inefficient and even ineffective (like collecting out-dated information). The resolution of the related complexity and inefficiency problems will be discussed later in this section. In order to support fast information gathering, the processor structure must provide the necessary interconnections and high communication bandwidth, and the program execution model should allow dynamic detection of instantaneous program parallelism. The dataflow model of computation [Denn80] supports such a program execution model.

When parallelism of both types are detected, the determination of the mapping between the needs of the program and the availability of resources must be made. This resource allocation is particularly difficult since the order of task assignments can affect the program behavior significantly. Task assigment based only on run-time information cannot guarantee best resource allocation [GPKK82]. Some kind of estimation on program behavior to guide the task assignment is useful.[1] Due to the nature of dynamic computation, unsatisfactory resource allocation is always a possibility. Resource reallocation could be considered to remedy such a situation. The processor structure and program structure should allow use of effective resource reallocation techniques.

The following dynamic resource allocation scheme has been proposed to address the problems discussed above. The basic ideas of this scheme are to use *a priori* information about program behavior to assist run-time scheduling and to allows remedial actions in case of poor allocations.

General Dynamic Resource Allocation Scheme

- At compilation time, do static program analysis to generate estimate measures (e.g., resource demands and number of dependent tasks) for every task.
- At program execution time, maintain simple system load measure(s). Dynamic resource allocation is done in three stages:
  1. dynamic scheduling based on estimate measures of tasks,
  2. dynamic task assignment based on the above scheduling and processor availability,

[1]The problem here bears a resemblance to static resource allocation. The major difference is that static information must be utilized with run-time information to decide the actual resource allocation.

3. dynamic resource reallocation based on system load measure(s).

When the hardware parallelism and the instantaneous program parallelism are not matched, action can still be taken to exploit additional parallelism. The action which is expected to be the most effective is a *dynamic grain size control* mechanism. If the program structure allows dynamic control of grain size, the scheduling and communication overhead should be able to be reduced. For example, if more hardware parallelism is available than the instantaneous program parallelism can utilize, the instantaneous program parallelism could be increased by breaking up executable units of computation into smaller grains. Similarly, if the hardware parallelism available is not as much as the instantaneous program parallelism needs, the executable units of computation could be grouped into large grain tasks.

2.2. Minimizing Overhead : Complexity and Efficiency Issues

Clearly, the overhead involved in resource management must be minimized in order that the computational resources be utilized for solving problems, rather than managing resources. Most resource management overhead comes from the dynamic collection and analysis of run-time information, and from the migration of tasks. The complexity and inefficiency of these management functions grow quickly with the size of processor and program structures. In order to minimize the overhead, the complexity of these management functions must be reduced and parallelism in resource management must be exploited.

An approximation may be used to reduce the complexity of the resource management problem. Resources are grouped into clusters, and program size is reduced by grouping units of computation into large grains. The resource management problem is then divided into simpler subproblems. Each subproblem handles a smaller number of resources (or clusters) and a smaller program. This "divide and conquer" approach also allows resource management to be operated in parallel to achieve better overall efficiency. In order to support this approach, the processor structure should directly support the concept of *resource clustering* and *distributed resource management*, and the program structure should provide the necessary representations for different grain sizes.

3. The Hierarchical System Architecture

In this section, a hierarchical system architecture which integrates considerations of processor structure and program structure to best support dynamic resource allocation is described. Both the processor structure and the program structure have a similar hierarchical structure. Mapping between the structures is done on level-to-level basis. This 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. Details of the system architecture and the program execution model are presented in the following paragraphs.

3.1. Processor Structure

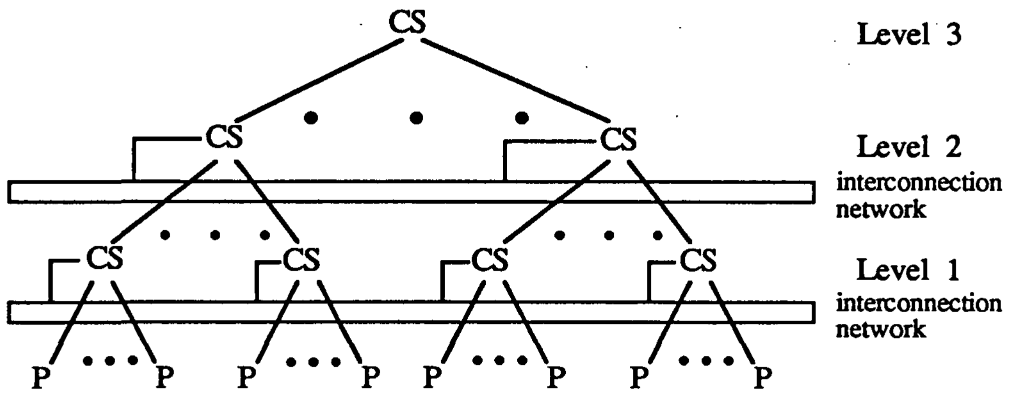


Figure 3-1: A three-level hierarchical multiprocessor system

Figure 3-1 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 (*PCU*) and data memories (*DM*). 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.

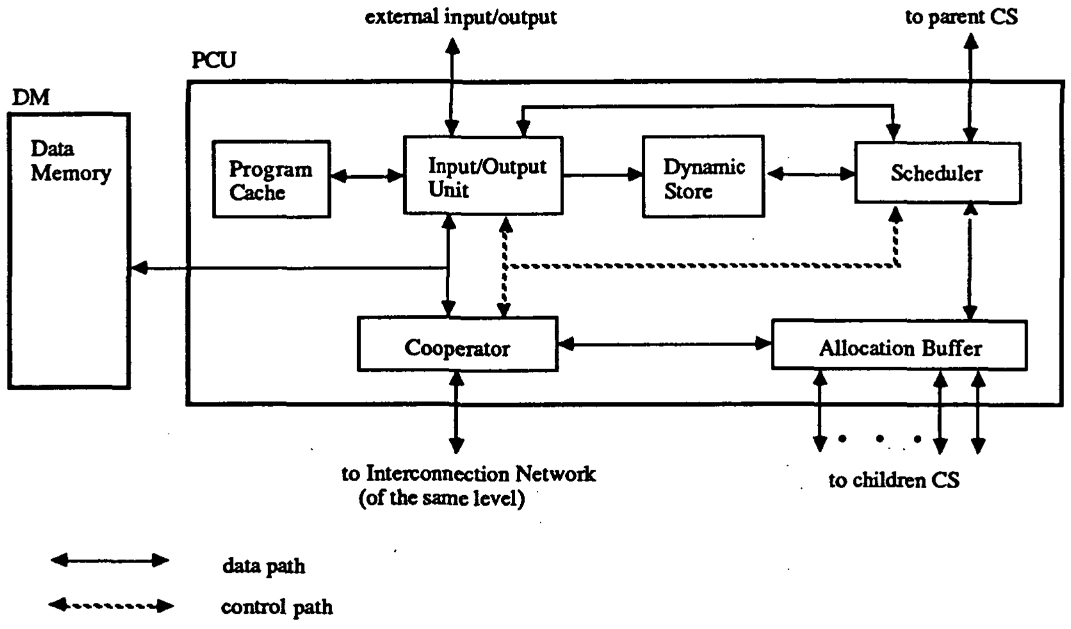


Figure 3-2: Functional blocks of a control subsystem

Figure 3-2 shows the functional blocks of a control subsystem. The data memory stores all local data structures. The remaining blocks constitute the processing control unit. The *scheduler* performs most of the dynamic resource management (dynamic scheduling and dynamic task assignment). The *cooperator* performs network communication, remote data access and dynamic resource reallocation. The *input/ouput unit* is responsible for all external input and ouput functions (e.g., disk accesses). The input/output unit is also capable of performing elementary data stream functions (e.g., filtering and merging). The *dynamic store* provides the memory working area to hold the executing programs, and the *allocation buffer* serves as the spool area for executable task packets. A *program cache* is provided for fast code retrieval.

3.2. Machine Program Structure

Figure 3-3 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* [DaKe82]. 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 next lower level or a primitive control function. Data objects are aggregates of data objects defined at the next lower level. The ground level dataflow graphs contain only elementary functions and primitive data object instances directly supported by the processors.

3.3. Program Execution Model

In the following program execution model, data-driven evaluation is assumed.[2] The general dynamic resource allocation scheme described in Section 2 is explicitly incorporated in the execution model. Figure 3-4 shows the typical flow of program execution inside a control subsystem. Figure 3-5 shows example activities in a four-processor control subsystem during a program execution. System activities are heavily overlapped in general.

Program execution proceeds as follows.

- As program execution begins, the scheduler of the control subsystem at the top level signals the input/output unit to load the top-level machine program into the dynamic store.
- As soon as the dataflow graph is loaded into the dynamic store, the scheduler uses the dataflow firing rule to determine any executable function nodes. Dynamic scheduling on these executable functions is then performed. Task packets which

[2]The proposed architecture and dynamic resource management is not restricted to data-driven evaluation. The described program execution model can slightly be modified to support demand-driven evaluation. Such detail is not pursued here.

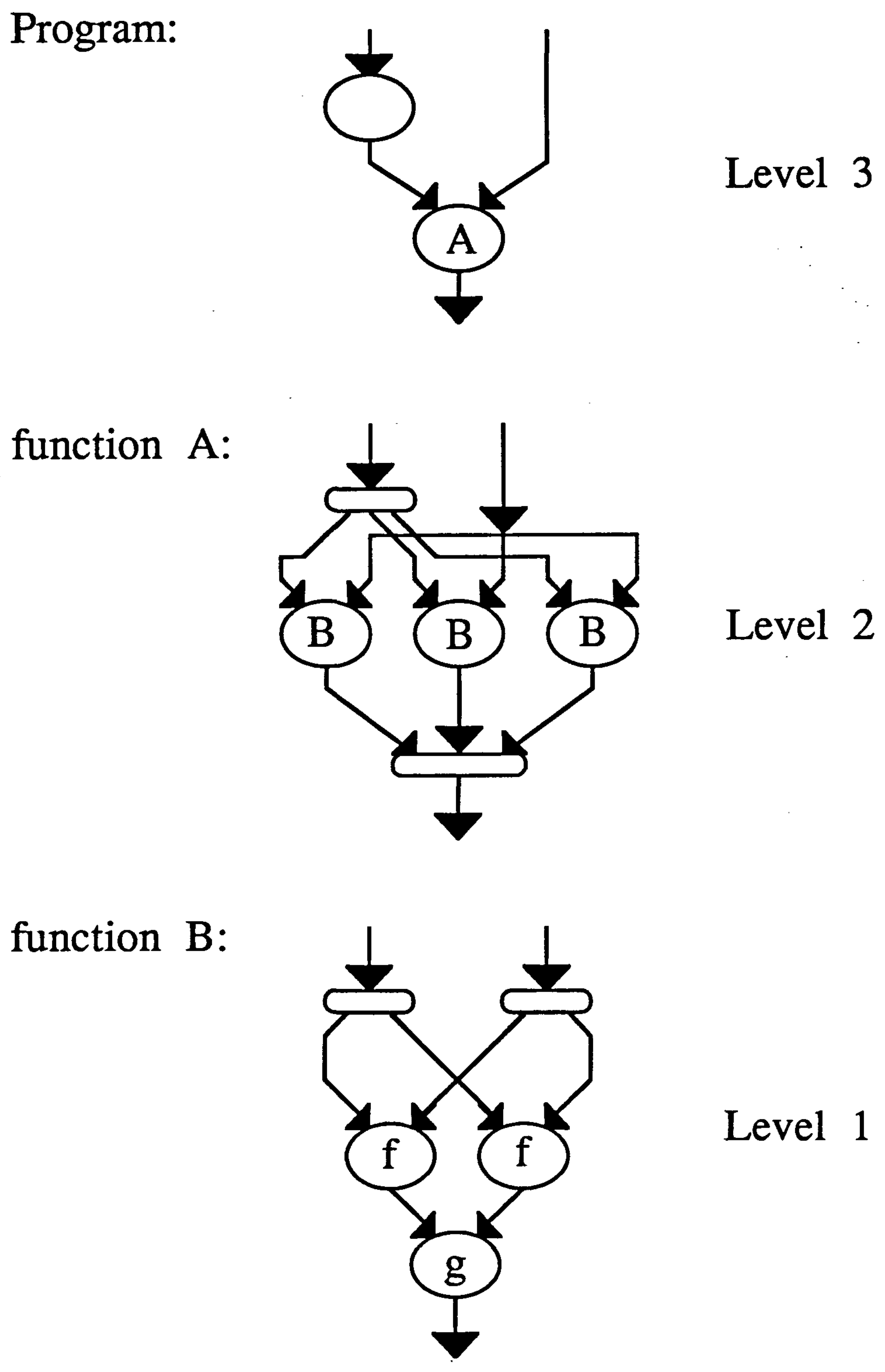
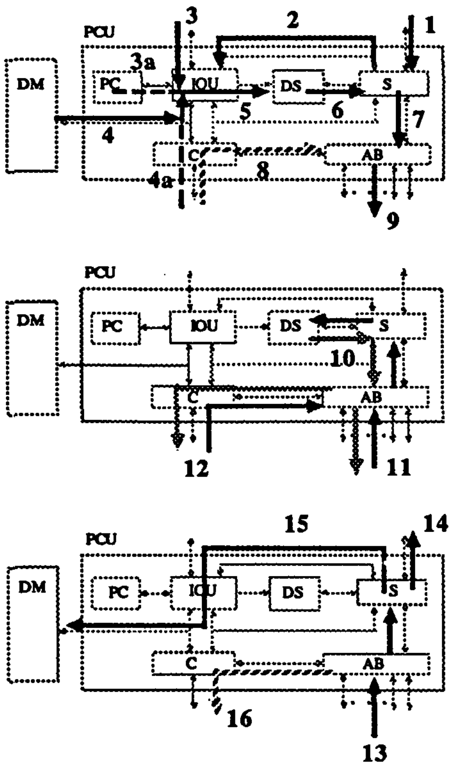


Figure 3-3: A machine program of three hierarchical levels



Task Initiation

1.get task packet from parent CS
 (not required in the top-level CS)
2. initiate progam loading
3. retrieve code from external I/O, or
 a. from program cache
4. retrieve data (if necessary) from DM, or
 a. from remote DM
5. program loading
6. initial dataflow-firing
7. dynamic scheduling
8. load sharing - get a task to execute
9. assign to a child CS for execution

Task Execution

10. return of result data and subsequent
 dataflow-firing
11. local execution by a child CS
12. load sharing - migrate to a sibling CS for
 remote execution

Task Completion

13. return of the last result data
14. return result data to parent CS
15. store data in DM (if necessary)
16. load sharing - return result to original CS

AB	Allocation Buffer	DM	Data Memory	PC	Program Cache
C	Cooperator	DS	Dynamic Store	PCU	Processing Control Unit
CS	Control Subsystem	IOU	Input/Output Unit	S	Scheduler

Figure 3-4: Typical flow of program execution

carry the function names together with the associated arguments and priorities are created. These task packets are put into the allocation buffer and are ready for assignment to a control subsystem at the next lower level.

• Schedulers in the control subsystems at the next lower level initiate the task assignment. When the control subsystem is able to take more tasks (i.e. current load is low), the scheduler accesses the allocation buffer in its parent control subsystem and grasps a task packet (if there is any).

• Once a task packet is assigned to a control subsystem, the scheduler passes the packet to the input/output unit and initiates program loading. The corresponding next lower-level function program is retrieved and is dynamically spliced with the

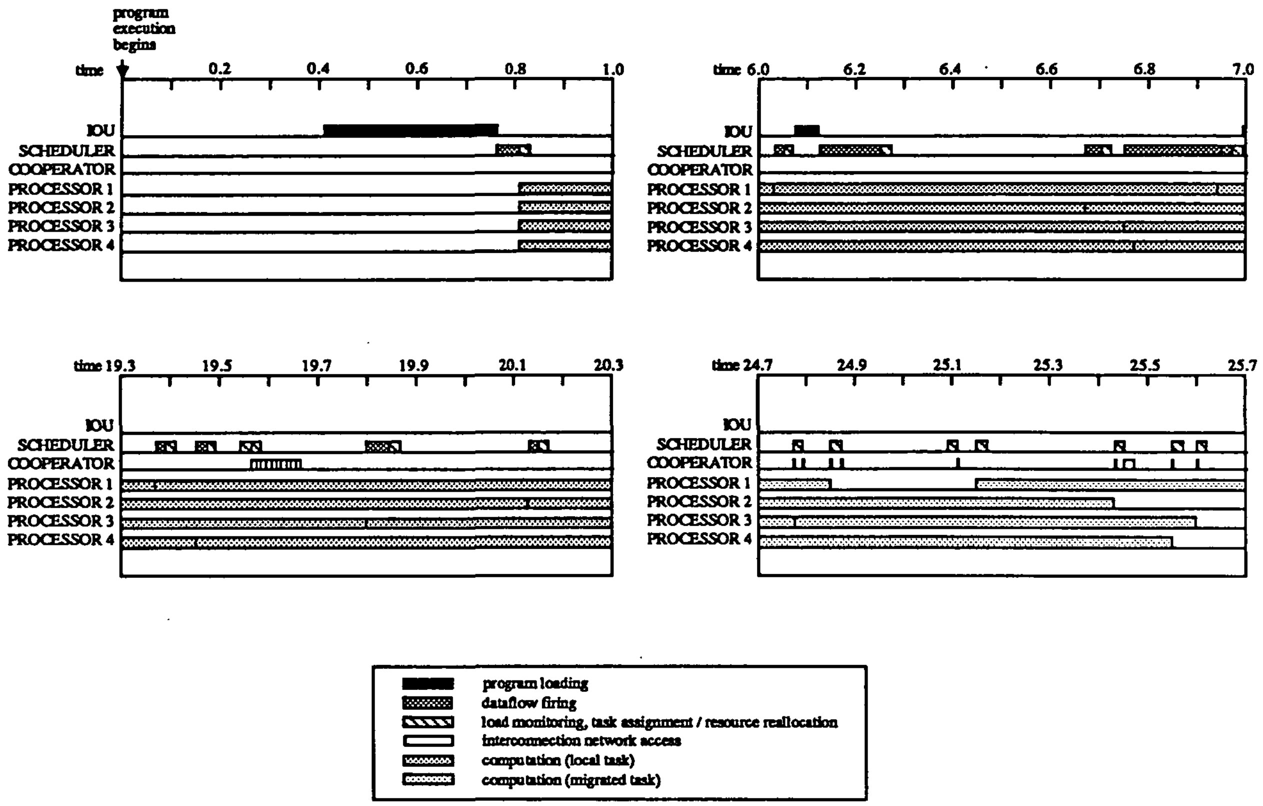


Figure 3-5: Example activities in a 4-processor control subsystem

input data instances to create a *static dataflow graph* [DeMi74].[3] The dataflow graph is then loaded into the dynamic store for execution.

- 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 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 data 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 allocation buffer of the parent control subsystem, the scheduler signals the cooperator to initiate resource reallocation. Task packets are migrated between

[3]Input data instances which are designated by the function arguments in the task packet can be either accessed in the local data memory, or remotely accessed via the cooperator.

allocation buffers of control subsystems at the same level to achieve load balancing.

4. System and Program Models

A preliminary study has been conducted to evaluate the proposed hierarchical system architecture with respect to different dynamic resource management strategies. A software simulator which models the proposed hierarchical system has been implemented and serves as the basic vehicle of the study. Hierarchical systems with different system parameters were simulated. Synthetic programs generated from different program behavior models were executed on the simulator and performance statistics were collected. Evaluation is based on the amount of overhead introduced by dynamic resource management. The study has focused on dynamic resource allocation during program execution and on how various components of resource management affect the overall system performance.

This section describes the system and program models being used in the preliminary study. Details of the simulator and generation of synthetic programs are also described here. The dynamic resource allocation algorithms under study, details of the simulation experiments and results of the study will be given in the following sections.

4.1. System Model

A simple inductive system model can be used to evaluate the proposed hierarchical system architecture. The basic systems are the one-level and two-level systems. Any hierarchical system of more than two levels can be approximated as a two-level hierarchical system (see Figure 4-1): Hierarchical systems rooted at the third top level are treated as 'processors' and any additional interconnections between these 'processors' are ignored. When system parameters and task characteristics are appropriately adjusted for these aggregated 'processors', results obtained in two-level systems can then be used to project overall performance of the system. Since the aggregated 'processors' are themselves hierarchical systems of fewer levels, a similar approximation can be applied to determine the corresponding system parameters. Thus the performance of the proposed hierarchical system can be estimated inductively.

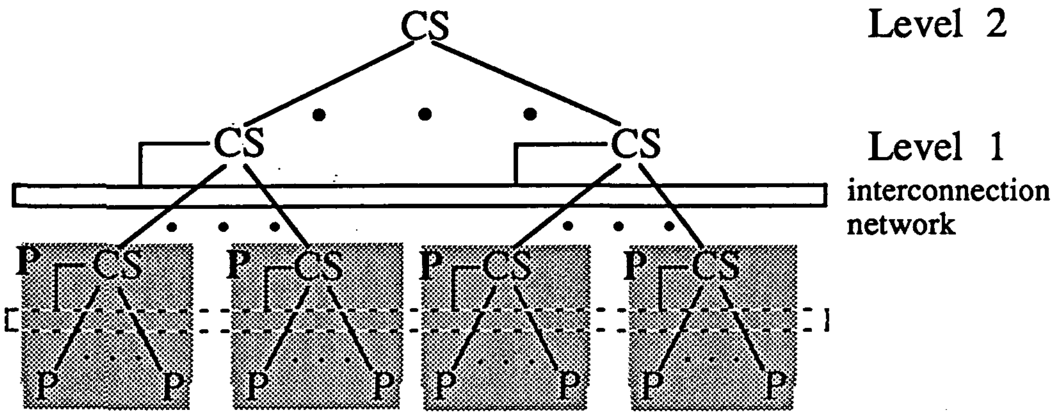


Figure 4-1: A three-level hierarchical system approximated as a two-level system

Table 4-1: System parameters

number of levels	2
system multiplicity	2, 4, 6, 8
number of processors	4, 16, 36, 64
data memory size	no limit
interconnection network	single bus

Based on the above observation, the preliminary study has concentrated on two-level hierarchical systems. To further simplify the system model under study, each control subsystem has the same number of control subsystems or processors at the next lower level attached to it. Hereafter this number is referred to as *the system multiplicity*. The interconnection network is modelled as a single shared resource (like a single bus). Table 4-1 gives system parameters of the system models under study.

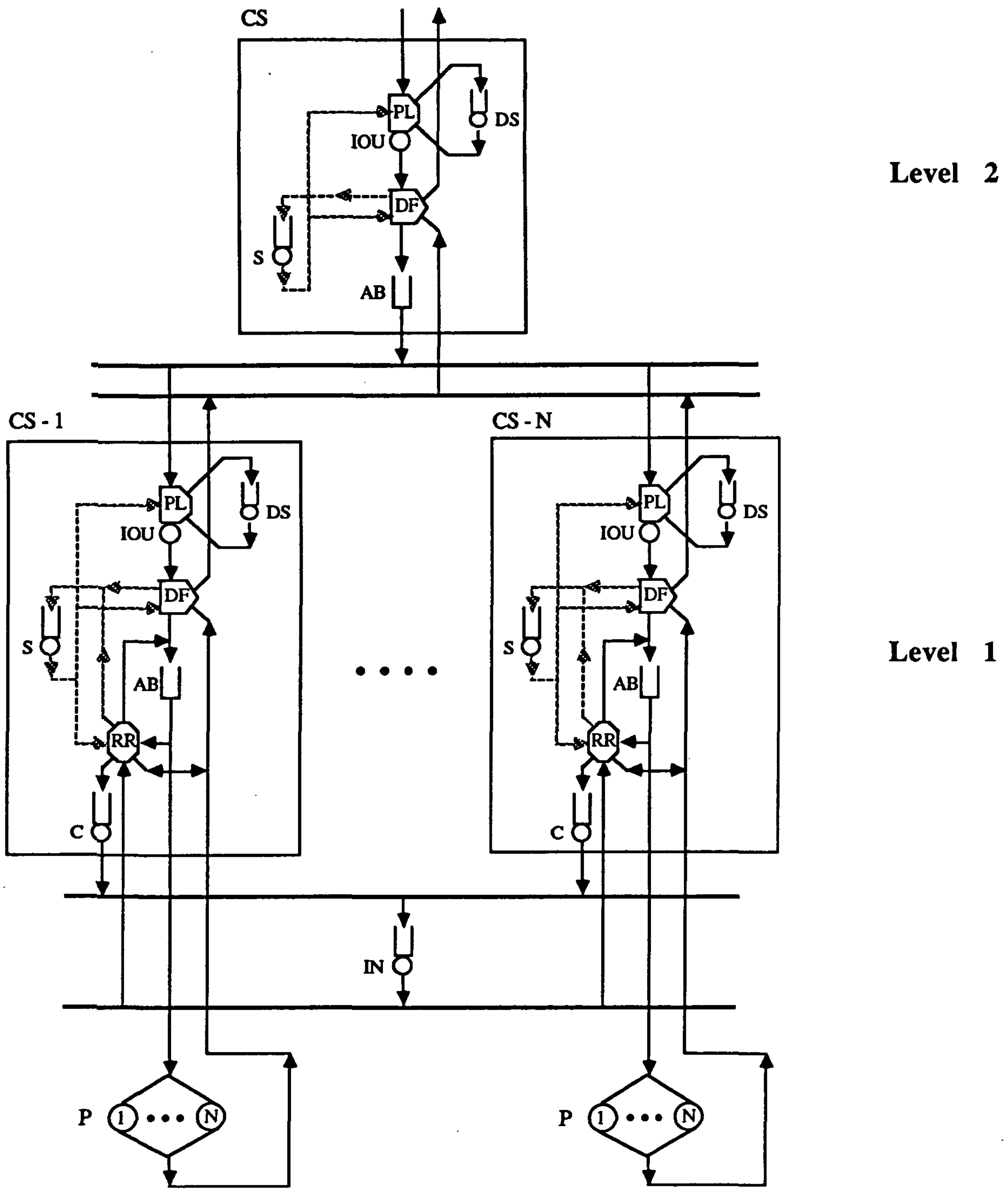
The hierarchical system simulator was written in an extended version[4] of CSIM [Schw86]. The dynamic resource management algorithms were built directly in the simulator. Extensive functions for statistics collection were provided. Figure 4-2 shows the simulation model of a two-level hierarchical system. Functional components in a control subsystem and the interconnection network are modelled as service centers in a queueing network. Using the program execution model described in Section 3, a task proceeds through the following stages of execution.

1. Program loading (*PL*) - initiated by the scheduler (*S*). After awaiting storage allocation in the dynamic store (*DS*), the input-output unit (*IOU*) loads in the task program.
2. Dataflow firing (*DF*) - initiated either by new program loading or by completion of a subtask execution. The scheduler is signaled. After the scheduler completes dataflow firing, a number of subtasks are allocated to the allocation buffer (*AB*). When the task completes execution, the result is returned to the parent control subsystem.
3. Subtask execution
    a. Local execution - initiated by an attached subsystem or processor.

    b. Remote execution - controlled by the resource reallocation function (*RR*). The scheduler performs load scheduling and initiates resource reallocation. Subtasks are removed from the allocation buffer, passed to the cooperator (*C*), transferred via the interconnection network (*IN*), and then allocated to the allocation buffer of another control subsystem. After remote execution, the result is returned to the original control subsystem. The resource reallocation function also handles the load sharing protocol between control subsystems of the same level.

Details of the scheduling and resource reallocation functions will be described in the next section. Table 4-2 gives the probability distribution functions of service times of the service

[4]Extensions, like dequeueing a process from a facility and accessing internal states of specific processes, are developed by the author to facilitate implementation of dynamic resource allocation algorithms.



AB	Allocation Buffer	DS	Dynamic Store	PL	Program Loading
C	Cooperator	IN	Interconnection Network	RR	Resource Reallocation
CS	Control Subsystem	IOU	Input/Output Unit	S	Scheduler
DF	Dataflow-Firing	P	Processor		

Figure 4-2: Simulation model of a two-level hierarchical system

Table 4-2: Probability distribution functions of service times

Service Center	Service	Probability Distribution
Input / Output Unit (IOU)	program loading (PL)	exponential
Scheduler (S)	dataflow-firing (DF)	exponential
Scheduler (S)	scheduling overhead	delta *
Interconnection Network (IN)	network transfer	delta *

*Service time is a constant.

centers. The mean values of these service times will be specified in Section 6.

4.2. Program Models

As discussed in Section 2, the performance of dynamic resource allocation depends largely on the behavior of the executing program. In the preliminary study, evaluation of the proposed architecture was based on different kinds of program behavior rather than on specific programs. This approach to evaluation not only gives more general results but also allows better understanding of the impact of program behavior. In order to obtain a large number of test programs with similar characteristics, different kinds of program behavior were modelled and synthetic programs were generated from these models.

As far as resource allocation is concerned, the most important characteristics of a program are the resource demands of elementary tasks (such as the amount of computation work and memory requirement), the dependency relationship among the tasks, and the number of times a task within a loop is executed. Other details of a program, for example what function is computed, are irrelevant in the program behavior model. For a preliminary study like this one, construction of a large number of machine programs and/or characterization of their dynamic behavior is not worthwhile. A simple stochastic model was used instead. The number of executable tasks during program execution is modelled as a stochastic process. This model avoids the details of task dependencies and loop executions, but offers appropriate execution profiles of some classes of parallel programs. Resource demands of an elementary task and size of a task are modelled as random variables. To further simplify the model, tasks at different program levels have similar stochastic structures. The program behavior model is described below:

Program behavior model

Parameters : Probability distribution functions of

- N_{size} - size of a task (any program level)
- N_{df} - number of executable subtasks per task per dataflow-firing
- T_{et} - execution time of an elementary task

Program behavior :

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

Synthetic programs can easily be generated according to the above program behavior model. Since the program model does not require program structures to be predefined, it is more efficient to generate synthetic programs during simulation. In this study, the program generation procedure is directly integrated into the simulator.

Table 4-3 gives four program models that were used in the study. For convenient reference in later discussions, the models are named by two letters followed by a number. The first letter denotes the probability distribution function of execution time of an elementary task; the second letter denotes the probability distribution function of the number of executable subtasks per dataflow-firing and task size; and the number denotes the mean size of a task. For example, in the program model XP11, execution time of elementary tasks is exponentially distributed, while

Table 4-3: Program Models

Parameters		XP11	XU11	XU20	UU20
		Program Model			
N_{size}	Mean	11	11	11	11
	Probability Distribution	exponential	uniform over [1, 21]	uniform over [15, 25]	uniform over [15, 25]
N_{df}	Mean	10	10	10	10
	Probability Distribution	Poisson	uniform over [0, 20]	uniform over [0, 20]	uniform over [0, 20]
T_{et}	Mean (time unit)	1	1	1	1
	Probability Distribution	exponential	exponential	exponential	uniform over [0.8, 1.2]

the number of executable subtasks per dataflow-firing and task size are Poisson distributed. (The other letter U denotes uniform distribution.) Every task has a mean size of 11 subtasks.

These program models were chosen to cover a variety of program behaviors. Elementary tasks may have different variances in execution time : exponentially distributed in XP11, XU11 and XU20, or uniformly distributed within a small range in UU20. The number of subtasks to be fired each time also varies: Poisson distributed in XP11, or uniformly distributed in other models. Two different task sizes were tested : 11 and 20. Task size is either Poisson distributed or uniformly distributed. It should note that in UU20 and XU20, task size varies little, i.e., uniformly between 15 and 25. The total program size are 121 elementary tasks in XP11 and XU11, and 400 elementary tasks in UU20 and XU20. These variations of program behavior contribute directly to the amount of inherent parallelism in the program. The more uniform the tasks, the more parallelism the program has. As more subtasks become executable per dataflow-firing, more parallelism can be exploited. Thus UU20 has the largest amount of parallelism, and XU11 and XP11 have the least. XU20 falls in between.

5. Dynamic Resource Allocation Algorithms

The dynamic resource allocation algorithms used in this preliminary study are simple and instructive. Instead of devising sophisticated algorithms, an understanding of how various components of resource management may affect system performance has been emphasized. No *a priori* information or estimation of the program behavior is assumed. Executable tasks are scheduled under a first-come(ready)-first-served(scheduled) discipline. In the following description of the task assignment and resource reallocation algorithms, *load level* of a control subsystem is defined to be the number of task packets in the allocation buffer. A scheduler always gives task assignment and resource reallocation a lower priority than dataflow-firing.

5.1. Task Assignment Algorithms

Task assignment in a control subsystem is initiated by schedulers in the children control subsystems. When load level of a control subsystem is below a threshold, th_{load}, the scheduler attempts to get a task packet from the allocation buffer of its parent control subsystem. The number of concurrently executing tasks assigned to a control subsystem is limited by another parameter, *max_task*.

In this study, th_{load} was chosen as two. Depending on whether multitasking is allowed, two different values of *max_task* were used. For non-multitasking task assignment, *max_task* is one. The corresponding algorithm is referred as Algorithm TA1. For multitasking task assignment, *max_task* was chosen as four and the algorithm used is referred as Algorithm TA4.

5.2. Resource Reallocation Algorithm

A simple load sharing algorithm, referred as Algorithm RR, was used. Control subsystems under the same parent control subsystem are assumed to be named in a cyclic order. That is, every control subsystem has a unique predecessor and a unique successor within the same level. The basic idea of the resource reallocation algorithm is to share load evenly between every pair of consecutive control subsystems. Load sharing is receiver-initiated [EaLZ85] (i.e., by control subsystems with resources available). The control protocol for load sharing is described below.

1. When the load level of a control subsystem is below th_{load} and no further tasks can be assigned to the control subsystem, a request packet is sent to the successor. The request packet carries current load information of the control subsystem.

2. When the successor control subsystem receives a request packet, its current load level is checked against that contained in the request packet. If its current load level is higher than its predecessor, task packets in the allocation buffer that account for half of the extra load migrate to the predecessor for remote execution.[1] Otherwise, the load request is pending.

3. In case of pending load request, the successor continuously monitors changes in its load level. Whenever the current load level allows load sharing (i.e. higher than that requested), an acknowledgement packet is sent to the predecessor. When the predecessor receives the acknowledgement packet, it checks its current load. If the current load is below th_{load}, a new request packet is sent to the successor to re-initiate load sharing.

4. When a control subsystem sends a request packet to its successor and is waiting for any response (either task packets or acknowledgement packet), no new load request is made unless current load level is lower than that requested.

Besides the above protocol, the following migration and execution disciplines are used: The last scheduled task always migrates first; and the first scheduled (local or migrated) task always be executed first. For migrated tasks, their original scheduling orders are recorded in the task packets and are used as if local scheduling orders.

6. Experiments and Results

The preliminary study is divided into two parts. In the first part of the study, five sets of simulations were run to study how individual resource management component affects system performance. In the second part, the hierarchical system was evaluated with respect to a typical set of system parameters.

[1]In case the extra load is an odd number, only its smaller half is shared.

The performance measure used throughout this study is *average execution time w.r.t. ideal execution.* Ideal execution here refers to program execution on an ideal multiprocessor system that has the same number of processors but does not incur any overhead due to resource management. An executable task is assigned to any available processor under a first-ready-first-scheduled discipline.[2] The average execution time w.r.t. ideal execution is defined as the ratio of average execution time to average ideal execution time. This performance measure indicates how well the dynamic resource allocation approximates the ideal execution. The closer to unity the ratio, the less the run-time resource management overhead impacts performance (so the more effective the dynamic resource allocation is).

For each simulation configuration (i.e., system configuration, set of system parameters, resource allocation algorithms and program model used), the average execution time is determined from simulation runs of 100 synthetic programs. Since this study has focused on resource management during program execution, program execution time reported here does not include initialization time (i.e., from the time the top level control subsystem initiates the first program loading to the time a ground-level processor starts the first computation).[3] For all simulation experiments reported here, results on average execution time have a 95% confidence interval within ± 5% range. (Average ideal execution time is determined from simulation runs of 1000 synthetic programs and has a 95% confidence interval within ± 2% range.)

[2]Under the assumption that no prior/exact information about program tasks is available, the above scheduling gives the best possible scheduling. When compared with optimal (deterministic) scheduling (which is strongly *NP*-complete), the scheduling is one of the best approximations known [LaLR82]: Program execution time resulted from such scheduling is less than two times that resulted from optimal scheduling.

[3]Average initialization time is independent of the program model used. For two-level hierarchical systems, it is equal to two times the sum of average program loading time and average dataflow-firing time.

6.1. Effect of individual resource management component

Five sets of simulations were conducted to study the effect of individual resource management components on system performance. In order to gain an insight into any performance limits due to individual management components, relatively fine-grain computations and slow communications were assumed (e.g., mean execution time of elementary tasks is in the order of 20 μs and 10-byte packets are transferred over a 10 MB/s bus). Table 6-1 shows the simulation configurations of the five sets of simulations. (Note that the mean execution time of elementary tasks is taken as one simulation time unit.) Table 6-2 shows how the simulation results of these experiments were compared in the study. Figures 6-1 to 6-5 gives the simulation results.

Table 6-1: Simulation configurations for study of individual resource management component

Configuration		Simulation Set A	B	C	D	E
Program Model		XP11	XP11	XP11	XU11	UU20
Service Time (time units)	Program Loading (PL)	0	1	0	0	0
	Dataflow-firing (DF)	0	0	0.1, 0.2	0	0
	Scheduling Overhead	0.05	0.05	0.05	0.05	0.05
	Network Transfer	0.05	0.05	0.05	0.05, 0.1	0.02, 0.05
Resource Allocation Algorithm		TA1 (only), TA4 (only), TA1 + RR, TA4 + RR				

Table 6-2: The study and the simulation sets

Study	Simulation Sets
Resource Allocation Algorithms	all
Impact of System Multiplicity	all
Effect of Program Loading Time	A and B
Effect of Dataflow-Firing Time	A and C
Effect of Network Transfer Time	D (for Program Model XU11)
	E (for Program Model UU20)

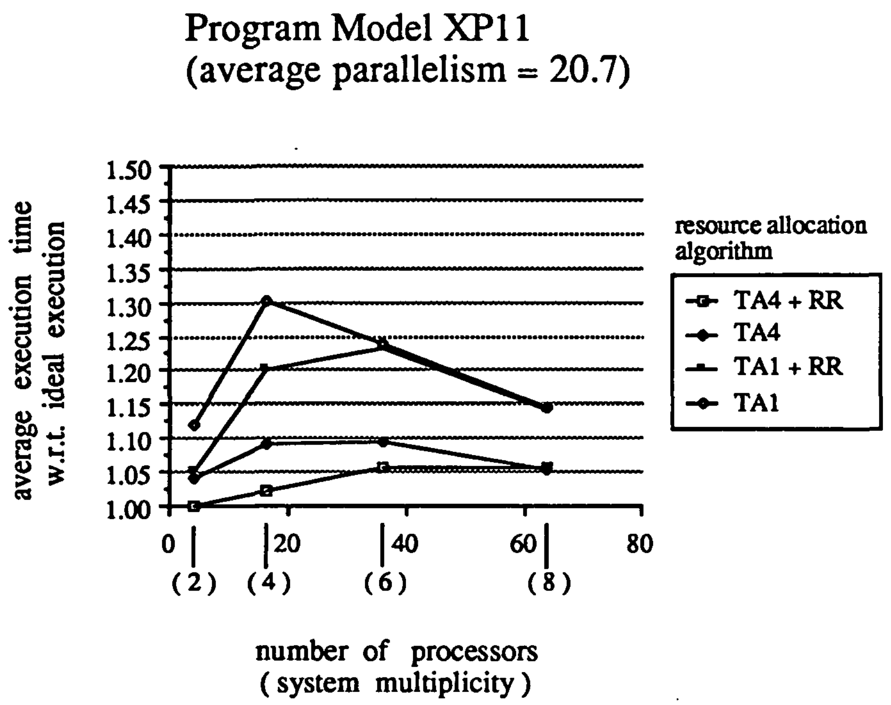


Figure 6-1: Simulation results (Set A)

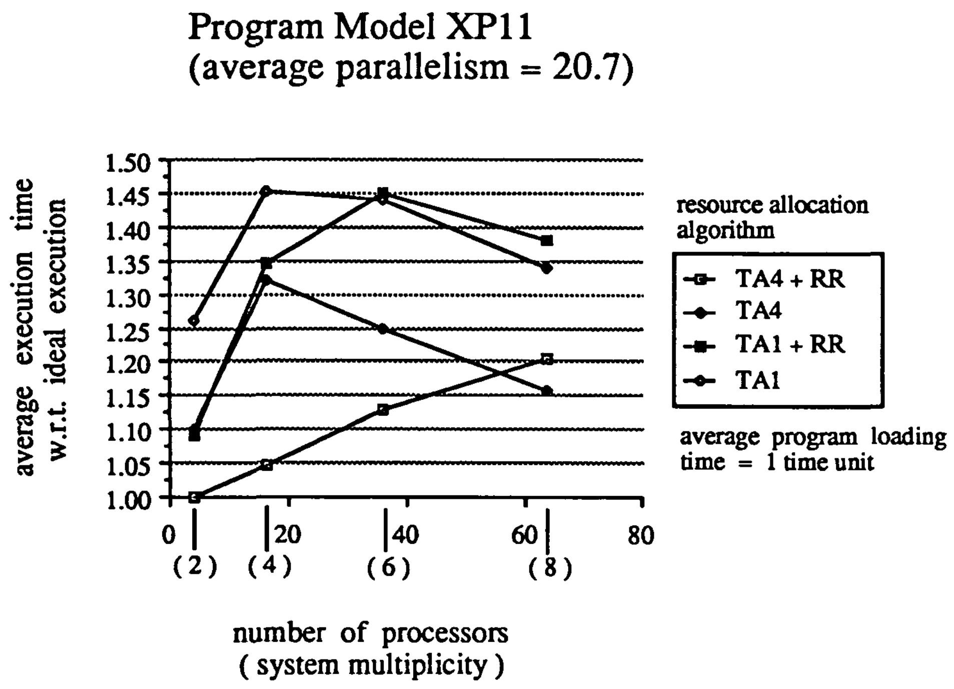
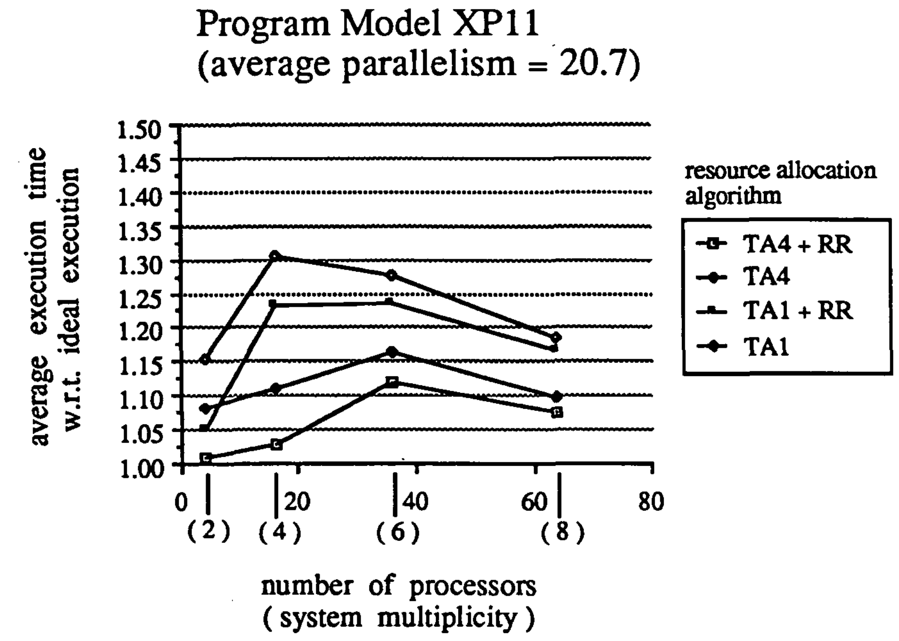
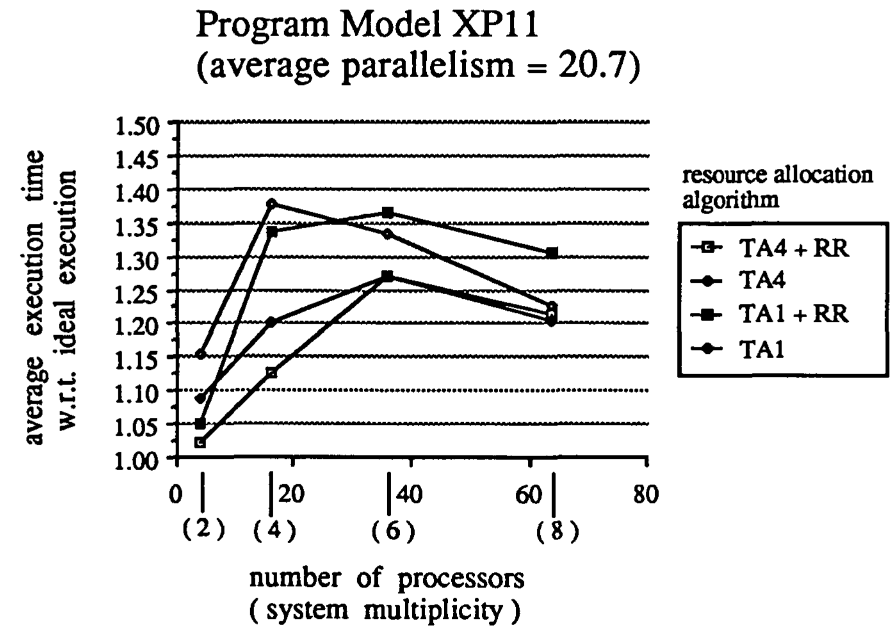


Figure 6-2: Simulation results (Set B)



(a) Mean dataflow-firing time = 0.1 time unit



(b) Mean dataflow-firing time = 0.2 time unit

Figure 6-3: Simulation results (Set C)

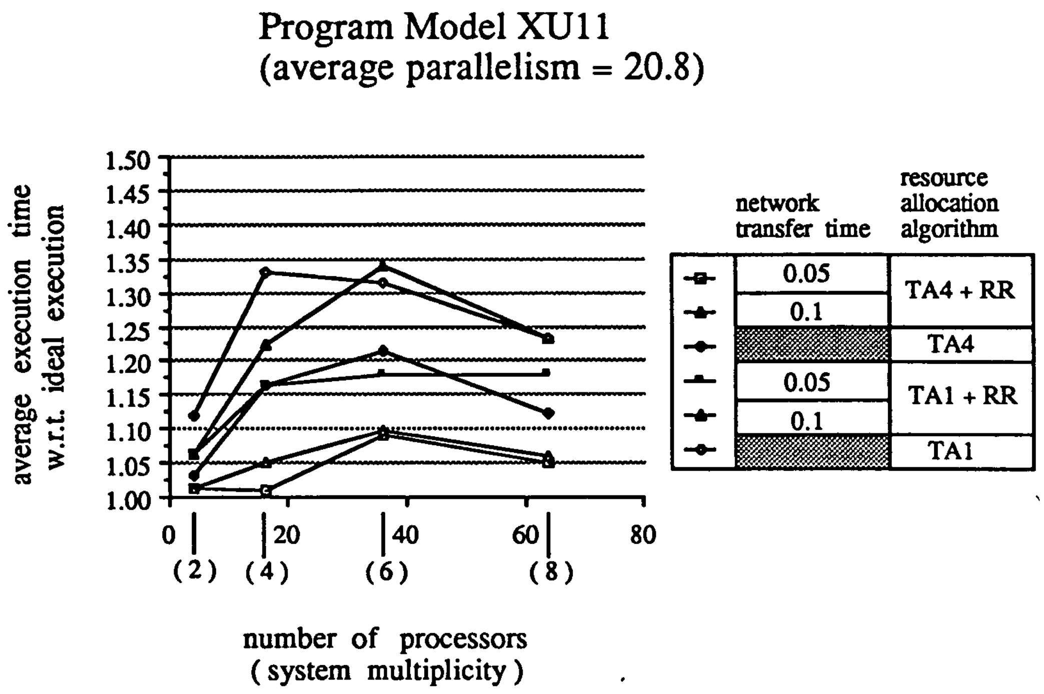


Figure 6-4: Simulation results (Set D)

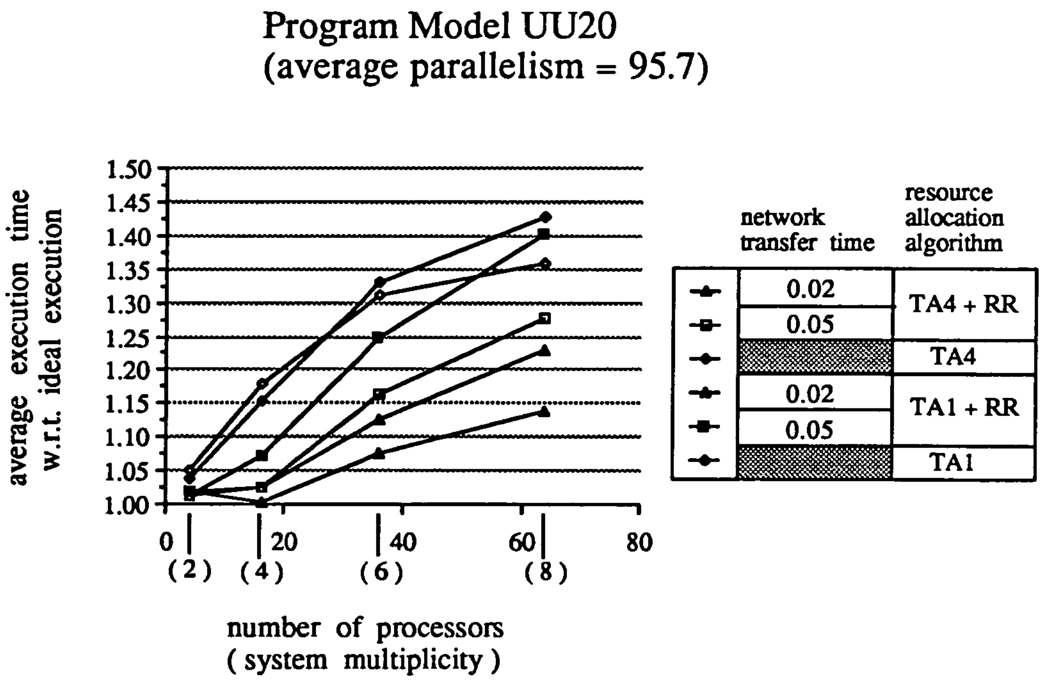


Figure 6-5: Simulation results (Set E)

Observations

- *On resource allocation algorithms*

  In general, algorithms with multitasking task assignment (TA4) are better than those with non-multitasking task assignment (TA1); algorithms with resource reallocation (RR) are better than those without. Resource allocation with both multitasking and load sharing (TA4 and RR) has the best performance.

- *On the general variation in performance with system multiplicity*

  In general, the average execution time w.r.t. ideal execution exhibits a bell shape variation with system multiplicity (i.e., the number of immediate processors attached to a control subsystem). When the system is overloaded (i.e., the number of processors is too small w.r.t. the *average program parallelism*[4]) or underloaded (i.e., the number of processor is too large), the resource management overhead tends to be small. Maximum overhead (w.r.t. ideal execution) occurs when the number of processors and the average program parallelism become comparable in magnitude. This observation agrees with the intuition that resource allocation is most critical when the instantaneous program parallelism fluctuates around the available hardware parallelism. This observation also shows that the resource allocation algorithms (especially the load sharing protocol described in Section 5.2) adapt well to global load situations.

- *On the effect of program loading*

  Long program loading time can affect system performance significantly. Figure 6-2 shows the performance when the average program loading time is as large as the grain-size of computation. When compared with the case that the program loading time is negligible (Figure 6-1), performance can be degraded by as much as 22%. Two important observations are:

  1. Resource allocation with multitasking and load sharing (TA4 and RR) is the least affected by program loading time. For system multiplicity up to six (i.e., with 36 processors), performance degradation is less than 8% and the total overhead is well within 15%.

  2. As system multiplicity increases beyond six, resource allocation with load

[4]Average program parallelism is determined by the time-average number of concurrently executable elementary tasks when executed on an ideal machine with infinite number of processors.

sharing (RR) has worse performance than that without. This phenomenon can be explained as follows: In the program model XP11, an average task at level 1 has a total computation work of 11 time units. When the system multiplicity is eight or above, subtasks can be executed concurrently by eight or more local processors. The average execution time of a task becomes comparable to the program loading time of 1 time unit. Since effective load sharing reduces task execution time, the bottleneck of execution shifts from subtask execution to program loading and becomes aggravated.

- *On the effect of dataflow-firing*

There exists a critical value of average dataflow-firing time beyond which system performance degrades notably. When the average dataflow-firing time is 0.1 time unit, system performance at small system multiplicity is similar to that with negligible dataflow-firing time (compare Figure 6-3a with Figure 6-1). At larger system multiplicities, system performance degrades by less than 6%. When the average dataflow-firing time increases to 0.2 time unit, system performance degrades by as much as 20% (see Figure 6-3b). When system multiplicity is larger than four, resource allocation with load sharing gives poorer performance than that without. A similar "bottleneck aggravation" phenomenon as in the program loading case accounts for such behavior: Consider the case of system multiplicity of six. Since six processors are attached to a control subsystem at level 1, the maximum throughput is one completed subtask per 0.17 time unit. Dataflow-firing with a mean time of 0.2 time unit obviously becomes the bottleneck. Because load sharing improves average turnaround time of a subtask, the bottleneck is aggravated.

- *On the effect of network transfer time*

Depending on characteristics of the program, network transfer time can affect system performance significantly.

  - Program model XU11 (Figure 6-4):

  When the average transfer time increases from 0.05 time unit to 0.1 time unit, the system performance of resource allocation with multitasking task assignment (TA4) varies less than 4%; but system performance of resource allocation with non-multitasking task assignment (TA1) degrades by as much as 14%. This observation shows that non-multitasking task assignment is more sensitive to long network transfer time.

- Program model UU20 (Figure 6-5):

  This program model has more uniform elementary tasks and more parallelism than XU11. When average network transfer time increases from 0.02 time unit to 0.05 time unit, the system performance of multitasking resource allocation degrades by as much as 14%, and the performance of non-multitasking resource allocation degrades by 18%.

The resource management overhead due to network transfer time is large for highly parallel programs. Consider the best resource allocation algorithm among the four algorithms studied (i.e., TA4 + RR) and the average network transfer time of 0.05 time unit. The resource management overhead is within 10% for program model XU11, but becomes as large as 28% for program model UU20. This large overhead is due to the large amount of resource reallocation made to exploit instantaneous program parallelism.

- *On the effectiveness of resource reallocation*

The simple resource reallocation algorithm (RR) can effectively balance system load to give better resource allocation. Consider the case with program model UU20 (Figure 6-5). At system multiplicity of eight, the multitasking task assignment causes more imbalance in system load than non-multitasking one, resulting a performance 7% poorer (compare algorithm TA4 with algorithm TA1). However, the load imbalance caused by multitasking task assignment provides a better opportunity to expolit parallelism. With resource reallocation (RR) at the average network transfer time of 0.02 time unit, the performance of multitasking resource allocation improves by 30% while that of non-multitasking resource allocation improves only by 13%. As a net result, multitasking resource allocation has a performance 10% better than non-multitasking resource allocation.

6.2. Evaluation of the hierarchical system architecture

In this part of the study, evaluation of the hierarchical system architecture was based on a typical set of system parameters. Table 6-3 shows the simulation configuration. Typical hardware appropriate for a high performance multi-microprocessor system (e.g., 16MHz microprocessors, memories with 100ns access time and 20MB/s busses) is assumed. The mean execution time of elementary tasks is in the order of 50 μs and all task packets and control packets are 10 bytes long. Resource allocation uses both algorithms TA4 and RR. Four different program models (XP11, XU11, XU20 and UU20) were used to evaluate the system architecture. Figure 6-6 gives the average execution time w.r.t. ideal execution. Figure 6-7 shows the actual average execution time (in simulation time units). In addition to execution time, network utilization and details of the overall network traffic are also recorded and are shown in Figures 6-8 and 6-9 respectively.

Table 6-3: Simulation configuration for evaluation of the hierarchical system architecture

Program Model		XP11, XU11, XU20, UU20
Service Time (time units)	Program Loading (PL)	0.2
	Dataflow-firing (DF)	0.05
	Scheduling Overhead	0.02
	Network Transfer	0.01
Resource Allocation Algorithm		TA4 + RR

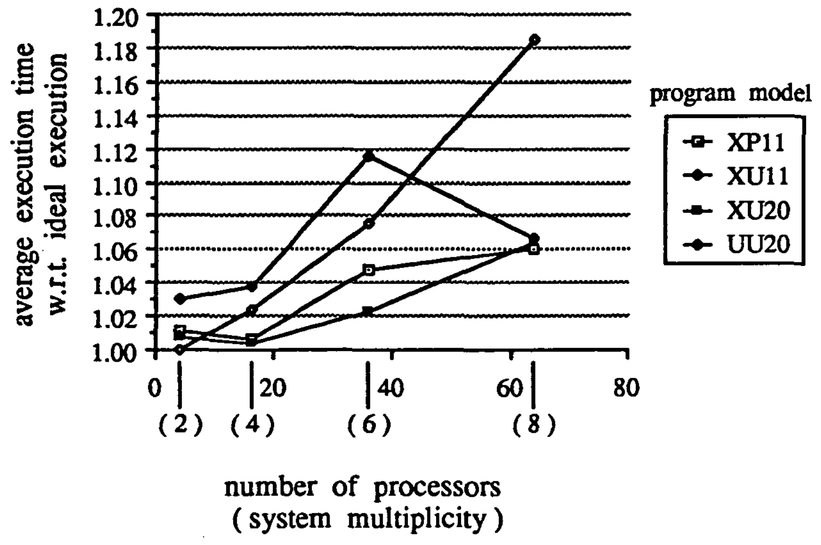


Figure 6-6: Average execution time w.r.t. ideal execution

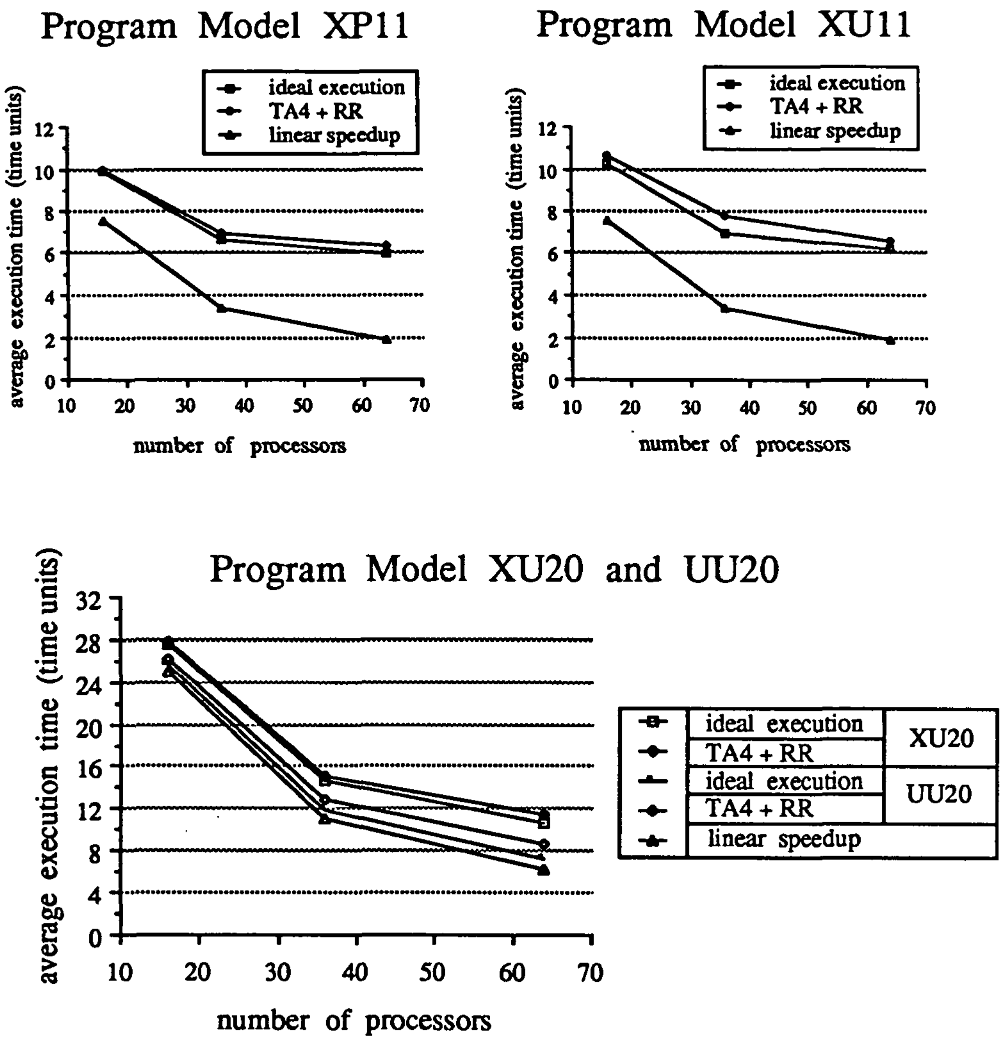


Figure 6-7: Actual execution time

Observations

- For the four program models tested, dynamic resource allocation incurs reasonably low overhead (see Figure 6-6). When the number of processors is 16 or less (i.e., system multiplicity is four or less), run-time resource management overhead accounts for less than 4% of the corresponding ideal execution. For larger systems (up to 64 processors), the resource management overhead is less than 20% (less than 8% in most test configurations).
- Run-time resource management overhead is most affected by task dependency

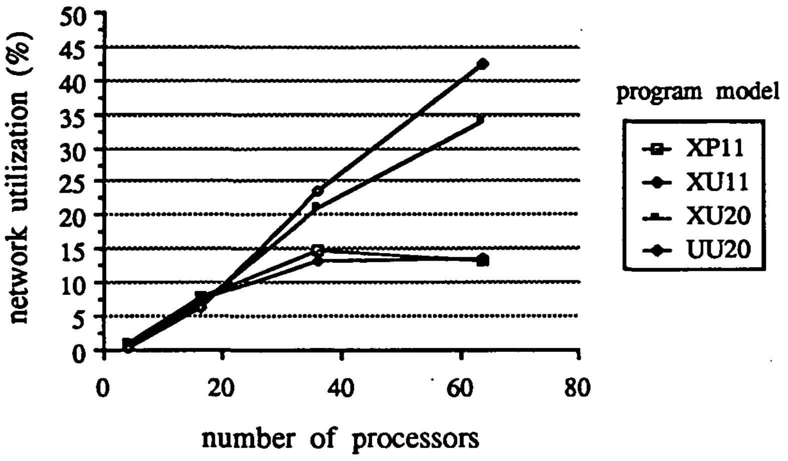


Figure 6-8: Utilization of the interconnection network

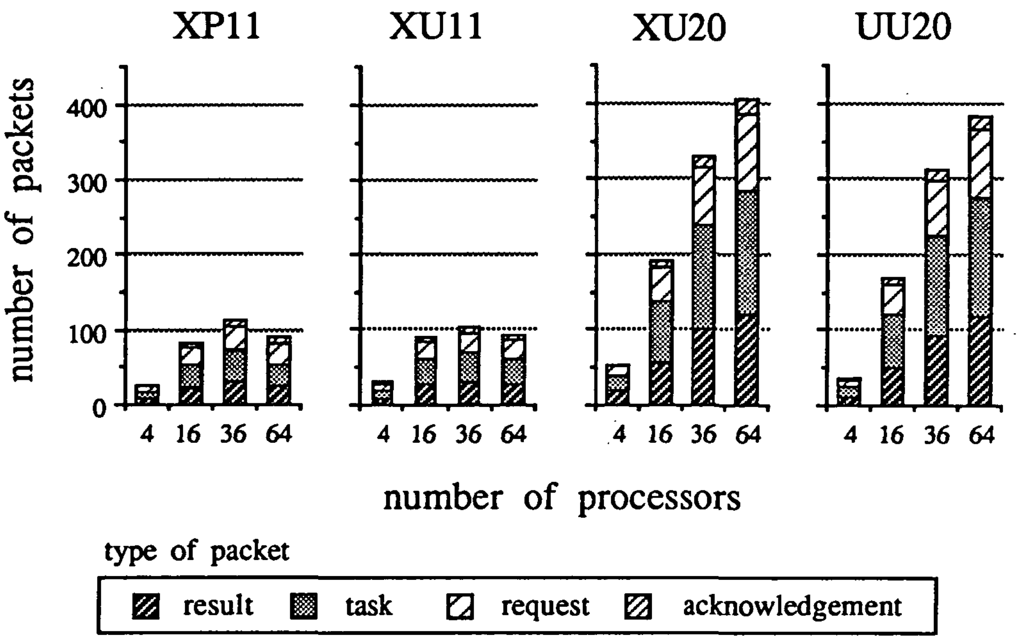


Figure 6-9: Details of the network traffic

patterns and uniformity of elementary tasks. In Figure 6-6, program models XU11 and UU20 show noticeably more overhead than XP11 and XU20. In the case of program model XU11, the task dependency pattern is more irregular than that of XP11 (random data-flow firing vs Poisson data-flow firing), thus leading to more imbalanced system load and more difficult resource allocation. In the case of program model UU20, elementary tasks are more uniform than those of XU20 (small variation of execution time vs exponentially distributed execution time). Since program model UU20 gains part of its parallelism from such uniformity of elementary tasks, remote executions of elementary tasks that disrupt the uniformity reduce the program parallelism and apparently result in more overhead. (But the gain from load sharing still outweighs the loss - as noted in the following observation.)

- Figure 6-7 shows that dynamic resource allocation is effective in exploiting parallelism. As the number of processors increases, the average program execution time decreases noticeably along with the corresponding ideal execution time. Even in the case that large resource management overhead is incurred (see curves for program model UU20), the exploitation of parallelism is remarkable: While the ideal execution time decreases by 392.5 time units from a uniprocessor system to a 64-processor system, the average execution time decreases by 391 time units (i.e., 99.6% gain).

- The utilization of the interconnection network is moderate, under 50% (see Figure 6-8). There are on the average 1-2 task packets per control packet (request or acknowledgement packet) and a slightly more than one task packets per result packet (see Figure 6-9). That is, each load request results an average of 1-2 task migrations and most remotely executed task migrates only once. These results show that the load sharing protocol used is effective and stable.

Conclusions

The preliminary study shows 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. four) or for programs that have non-uniform elementary tasks and less random task dependency pattern (such as XP11 and XU20). For systems with larger system multiplicity and programs that have either uniform elementary tasks or totally random task dependency patterns, the gain in exploiting parallelism outweighs considerably the resource management overhead incurred. The load sharing protocol used is shown to be effective and stable.

7. Summary and Future Research

Dynamic resource allocation is an important but very difficult problem in multiprocessor system design. The major difficulties lie in the exploitation of run-time parallelism and in the minimization of dynamic resource management overhead. A hierarchical system architecture which integrates considerations of processor structure and program structure was proposed to alleviate these difficulties. Preliminary evaluation indicates that with the proposed system approach, the complexity of dynamic resource management could be significantly reduced while achieving reasonably effective dynamic resource allocation. Many open issues remain to be addressed before the proposed system architecture can be fully evaluated. Some of these issues are :

- For a given ratio of network service time to computation time, what is the optimal dynamic resource allocation algorithm and system multiplicity? Stated in other words, for a given system multiplicity and network service time, what dynamic resource allocation algorithm and (the smallest) granularity of computation should be used?
- What hierarchical characteristics do parallel programs naturally exhibit?[5] What kind of program restructuring is desirable for efficient execution on the proposed architecture?
- What additional performance improvement can be achieved by dynamic scheduling (with static estimates)?
- How data referencing patterns of a program affect the performance of the hierarchical system? What is the best data placement strategy?

[5]Characteristics of interest are number of levels, function and data referencing patterns.

- What system performance can be achieved with other interconnection network topologies?
- How can the proposed system architecture be extended to support the use of specialized processors and to provide fault tolerance? Does the system architecture support the development of efficient and reliable parallel programs?

Studies on these issues will further explore the system architecture described here and expand the existing techniques to exploit parallelism.

References

[BrOP86] J.D. Brock, A.R. Omondi and D.A. Plaisted, "A Multiprocessor Architecture for Medium-Grain Parallelism," in *Proc. 6th Intl. Conf. on Distributed Computing Systems*, May 1986, pp. 167-174.

[BuSl81] F.W. Burton and M.R. Sleep, "Executing Functional Programs on Virtual Tree of Processors," in *Proc. ACM Symp. on Functional Programming Languages and Computer Architecture*, Oct. 1981, pp. 187-194.

[DaKe82] A.L. Davis and R.M. Keller, "Data Flow Program Graphs," in *IEEE Computer*, Feb. 1982, pp. 26-41.

[DaRe81] J. Darlington and M. Reeve, "ALICE - A Multipcessor Reduction Machine for the Parallel Evaluation of Applicative Languages," in *Proc. ACM Symp. on Functional Programming Languages and Computer Architecture*, Oct. 1981, pp. 65-75.

[Davi78] A.L. Davis, "The Architecture and System Method of DDM-1 : A Recursively-Structured Data Driven Machine," in *Proc. 5th Annual Symp. on Computer Architecture*, 1978.

[DeMi74] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," in *Proc. 2nd Annual Symp. on Computer Architecture*, Dec. 1974, pp. 126-132.

[Denn80] J.B. Dennis, "Data-Flow Supercomputers," in *IEEE Computer*, Nov. 1980, pp. 48-56.

[EaLZ85] D.L. Eager, E.D. Lazowska and J. Zahorjan, *A Comparison of Reciever-Initiated and Sender-Initiated Dynamic Load Sharing*, Technical Report 85-04-01, Department of Computer Science, University of Washington, April 1985.

[GKLS83] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "Cedar - A Large Scale Multiprocessor," in *Proc. 1983 Intl. Conf. on Parallel Processing*, pp.

524-529.

[GPKK82] D.D. Gajski, D.A. Padua, D.J. Kuck and R.H. Kuhn, "A Second Opinion on Data-Flow Machines and Languages," in *IEEE Computer*, Feb. 1982, pp.58-69.

[GuKW85] J.R. Gurd, C.C. Kirkham and I. Watson, "The Manchester Data-Flow Computer," in *Communication ACM*, Vol. 28, No. 1, Jan. 1985, pp.34-52.

[KeLP79] R.M. Keller, G. Lindstrom and S. Patil, "A Loosely-Coupled Allplicative Multi-Processing System," in *AFIPS Conf. Proc.*, Vol. 24, NCC, June 1979, pp. 613-622.

[KeLT84] R.M. Keller, F.C.H. Lin and J. Tanaka, "Rediflow Multiprocessing," in *Digest of Papers, IEEE Spring 1984 COMPCON*, pp. 410-417.

[LaLR82] E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan, "Recent Developments in Deterministic Sequencing and Scheduling : A Survey," in *Deterministic and Stochastic Sceduling*, M.A.H. Dempster et.al. (eds.), D. Reidel Publishing Company, Dordrecht, 1982, pp. 35-73.

[LiKe86] F.C.H. Lin and R.M. Keller, "Gradient Model : A Demanded-Driven Load Balancing Scheme," in *Proc. 6th Intl. Conf. on Distributed Computing Systems*, May 1986, pp. 329-336.

[Schw86] H. Schwetman, "CSIM : A C-Based, Process-Oriented Simulation Language," in *Proc. Winter 1986 Simulation Conference*.