HIERARCHICAL MULTIHOST BASED OPERATING SYSTEM FOR SIMULTANEOUS MULTIPLE APPLICATION EXECUTION ON MIP SCOC CLUSTER

A Thesis
Submitted to
Waran Research Foundation (WARFT)
by

Karthik Ganesan
Undergraduate Research Trainee

In Partial Fulfillment of the Requirements for the Research Training Program
@
Waran Research Foundation.

Vishwakarma - High Performance Computing Group

Waran Research Foundation (WARFT)
Chennai, India
URL: http://www.warftindia.org
email: waran@warftindia.org
Phone: 91-044-24899766
July 2006

Prologue

Waran Research Foundation is engaged in initiating a major research project focusing on supercomputing architectures and brain modelling. The thrust is towards evolving a simulation model for predicting the Multi-Million neuron interconnectivity involving Dendrites-Axon-Soma-Synapse of the Brain Regions whose BOLD-fMRI is known and Evolution of a Neurophysiologically Inspired Supercomputing Architecture for Modelling the Respective Brain Regions. In this direction a revolutionary design paradigm named MIP SCOC (Memory In Processor SuperComputer On Chip) for supercomputers has been evolved. To accomplish a realistic brain modelling of far reaching significance a Million node MIP SCOC Fusion cluster is under exploration. The highlight of this research initiative at WARFT is the dedicated involvement of undergraduates as research trainees. This thesis is directed towards the above research initiative.

Prof. N. Venkateswaran
Director, Waran Research Foundation

HIERARCHICAL MULTIHOST BASED OPERATING SYSTEM FOR SIMULTANEOUS MULTIPLE APPLICATION EXECUTION ON MIP SCOC CLUSTER

Approved by:

_____

Prof. N. Venkateswaran
Director,Waran Research Foundation,Adviser

Date:

_____

"Fear On Self
Is The
Beginning Of Wisdom"

– My Guru, Mr. N Namadurai

Abstract

The Memory In Processor Super Computer On a Chip (MIP SCOC) [1] [2] cluster being evolved at WARFT [3] is completely suitable for simultaneous multiple application execution. With proper application mix, the sustained performance and performance scalability become an inherent characteristic of the MIP SCOC cluster due to the heterogeneous characteristics of the MIP SCOC node. The application mix is chosen in such a manner leading to an algorithm mix that can exploit the topology efficiently, and on the whole, making the mapping process [4] topologically independent, to achieve performance scalability. The MIP SCOC cluster has been envisaged for beyond exaflops scale applications.

Though the simultaneous multiple application mapping may appear complex, it is resolved by the higher level (In terms of MIP Instructions–on an average 1 MIP instruction is approximately equivalent to 100-250 ALU instructions in the context of overall execution with regard to the application characteristics and size) library design resulting in a simplified mapping process [5]. The Libraries, designed based on extensive simulations and a static schedule are also included as a part for the efficient use of the MIP SCOC resources. This static schedule is fixed by simulating the mapping process based on the concept of simulated annealing and population theory [6] [7].

The simultaneous application mapping on a million node cluster needs an efficient host system to tackle the complexity involved. This necessitates a need for parallel and hierarchically based multiple host system. The focus of the thesis is to evolve an operating system to tackle multiple application map-

ping and to be implemented through an already proposed novel Hierarchical Multiple Host system for a million node MIP SCOC cluster.

The applications are presented to the MIP cluster at the primary host plane. The Operating System comprising of the primary and the secondary hosts is expected to perform the following highly complex operations for execution of simultaneous multiple applications on MIP SCOC cluster.

- Application partitioning among the Primary Host
- Problem partitioning among the Secondary Hosts done at the primary Host plane.
- Analysis of dependency between different algorithms of the applications.
- Implementing the static scheduling strategies (Efficient Load balancing with reduced communication spread)
- Packet formation.
- Memory Management and data mapping.
- Interrupts and Exception handling.
- Resource reporting up the hierarchy.
- Garner the output after execution from the nodes.

To handle the complexity involved in the process of simultaneous Multiple Applications, the Operating System is Silicon implemented by appropriate Hardware viz., Algorithm Level Functional units present in the primary and the secondary hosts. As a Part of my thesis, the architecture of the Algorithm Level Functional Units (ALFUs) and the architecture of the Primary and the Secondary Hosts are being designed. As a consequence of this Hardware Implementation, the operating system has higher reliability, Security and does not suffer from Aging.

Acknowledgements

I am very grateful to the Selfless, dedicated and my passionate research Guru, Prof. N Venkateswaran, affectionately known as Waran by his students. He is the mentor for this thesis and also coined us towards achieving research goals at WAran Research FoundaTion (WARFT), where I closely associated with him as part-time Research Trainee. Prof. Waran has always been motivating us through his golden word, "research". I express my sincere thanks and gratitude to him. His perseverance and passion towards research have always fascinated me, which I hope will take me to greater heights. His lectures and motivating words are engraved in my heart to take up research as a profession in the time to come.

It is not out of place to mention the constant motivation and the potential support given by my parents throughout my research program, which has instilled in me the thrust to attain greater heights in my life. I would also like to thank my peer research trainees at WARFT with whom I enjoyed the process of doing research.

Table of Contents

List of Figures

Chapter 1

Introduction

The single factor limiting the harnessing of the enormous computing power of clusters for parallel computing is the lack of appropriate software. Present cluster operating systems are not built to support parallel computing - they do not provide services to manage parallelism. The cluster operating environments that are used to assist the execution of parallel applications do not provide support for either Message Passing (MP) or Distributed Shared Memory (DSM) paradigms. They are only offered as separate components implemented at the user level as library and independent servers. Due to poor operating systems users must deal with computers of a cluster rather than to see this cluster as a single powerful computer. A Single System Image of the cluster is not offered to users. There is a need for an operating system for clusters. We claim and demonstrate that it is possible to develop a cluster operating system that is able to efficiently manage parallelism, support Message Passing and DSM and offer the Single System Image.

Because of the increasing demand [8] in the application areas like Monte Carlo algorithms, Computational Fluid Dynamics, Ocean Modeling, High-end Signal and Image processing applications, Simulations of Brain Modeling, Medical Imaging and Galaxy Modeling, it becomes mandatory to evolve supercom-

puting clusters whose node architecture has to be highly tuned towards the application. This, to a large extent, alleviates the time complexity which is currently predicted in petaflop years for applications running in general purpose supercomputers. But evolving such Super Computing architectures for specific applications is cost prohibitive. Under such conditions it is necessary to make the node architecture heterogeneous involving varied functional units covering wide applications. This necessarily led to the novel concept of simultaneous multiple applications mapping on such heterogeneous node clusters [9] [10].

Since the advent of the DSM Technology, steep rise in the processing performance in a single node boasting a conventional architecture is realizable. However, achieving high performance at the cluster level will be difficult, in spite of the powerful nodes being employed. The node architecture, the cluster architecture, the associated network topology and the mapping strategies together, play a major role in achieving sustained and scalable performance. This thesis proposes a novel Hierarchical Multihost based operating System for simultaneous Multiple application Execution on MIP SCOC Cluster, which is capable of achieving sustained performance as well as performance scalability at the cluster level.

1.0.1 Is Simultaneous Multiple Application Execution Realizable on a Conventional Cluster node?

In conventional clusters, the node is usually installed with an operating system with a single process minimalist kernel in which running multiple applications will not be possible and even if possible not at good efficiency. For example BlueGene uses the BLRTS kernel as the compute node operating system and a Linux image in the I/O nodes for every 64 compute nodes. But in the MIP

SCOC node architecture, the functional units are modeled as resources which may be exploited for maximum parallelism aiming at simultaneous multiple application execution.

High - end computations in scientific and industrial applications, call for significantly large high-performance computing resources on a massively parallel cluster as cited in [11]. In the present scenario, more computational nodes are being added to the cluster to improve the performance. Apart from realizing very high computing performance, it is essential to improve mapping capability to match with the available class of computational power and the network topology. If mapping is not efficient, communication complexity will override the computational power of the node, negating the advantage of increasing the number of nodes and affecting the sustained performance and performance scalability. The proposed architecture model will aid in facing the challenges to petaflop computing mentioned in [8].

Sustained performance is obtained by appropriate interleaving of the applications considering the dependencies among algorithms within an application. In an application mix, some algorithms will tend to exploit the topology more efficiently in comparison with others making the mapping process topologically independent on the whole, to achieve performance scalability. The foremost principle of the developed mapping strategy, deals with interleaving multiple applications, based on the dependencies between algorithms of the same application and deciding the proper application mix. The data flow model that we propose provide the most appropriate application level mapping for the cluster, with the observation that execution of a set of infused applications of varied characteristics, for any given topology, yields an improved sustained performance. This model would strive to utilize the full potential of the cluster

and call for maximum resource utilization, realized by highly efficient heuristics for parallel mapping of Simultaneous Multiple Applications (SMAPP).

This mapping is implemented by appropriate load balancing [12] [13] [14]techniques and dynamic scheduling [15] [16] heuristics, taking into consideration, the resource availability. We arrive at the best application mix, employing static heuristics and matching Inter-application algorithm mix, by comparing resource utilization and workload distribution, dynamically. We intend to demonstrate an enhanced sustained performance and an unmatchable scalability for the designed mapping and load balancing heuristics. Besides, the node architecture employed ought to be highly resourceful and computationally powerful similar to that of the MIP SCOC [17] to exploit the supremacy of the proposed data flow model and the corresponding mapping heuristics. To attain an enhanced sustained peak performance as stated earlier it is imperative to develop a novel node architecture, a cluster architecture and corresponding mapping heuristics for SMAPP.

Towards this I show in this thesis, the extreme adaptability of the MIP SCOC [2] architecture and the MIP cluster architecture for SMAPP execution. Critical applications like Monte Carlo algorithms, Computational Fluid Dynamics, Ocean Modeling, High-end Signal and Image processing applications, Simulations of Brain Modeling, Medical Imaging and Galaxy Modeling, that need high performance computing can be run simultaneously to maximize resource utilization. Majority of these applications involve different classes of algorithms including convex Hull, graph partitioning, matrix decomposition and many others. Synthetic applications involving these algorithms can be generated as inputs for simulating the SMAPP data flow model using the WARFT BENSIM which is a cluster level application benchmark [3].

This operating system design for simultaneous multiple applications need a hardware based solution to handle the complexity involved. So, the operating features are to be implemented directly as functional units which reside in the host system of the cluster. The Compilers On Silicon are the hardware scheduler and mapper units which are present in the node itself. This helps in an increased reliability and security through hardware functional units making the cluster suitable for Time critical applications. The next few chapters talk about the MIP node architecture and the cluster architecture and then the design of the host systems for the cluster which make up the cluster and its operating system respectively.

Chapter 2

MIP Node Architecture

This review is based on the past papers on the Memory In Processor architecture. With the DSM technology one can place hundreds of ALUs and Registers and evolve a powerful node. However such an approach is naive at best, as billions of low level instructions need to be executed and mapping within a node (resource allocation) becomes extremely complex. To reduce the above complexity, the node should possess a wide class of Algorithm Level Functional Units(ALFUs) like Matrix Multipliers and Graph theoretic units.In this context the Architecture of all these functional Units should be designed to have a memory like Cell-Based architecture.

Because of the increasing demand in the application areas like Space maneuvering, Space Docking, Space Medicine, Remote Sensing, Satellite Imaging, Galaxy simulation, Brain modeling etc., it becomes mandatory to evolve supercomputing clusters whose node architecture is highly tuned towards the application. This to a large extent alleviates the time complexity which is currently predicted in petaflop years for applications running in general purpose supercomputers.

But evolving such architectures tuned to specific applications may be cost

prohibitive. This necessarily leads to the evolution of a new concept of Simultaneous Multiple Application Mapping (SMAPP). Under such conditions it is necessary to make the node architecture heterogeneous involving varied functional units covering wide applications.

Though the simultaneous multiple application mapping may appear complex, it is resolved by the higher level ALISA based library design resulting in a simplified mapping process. On the other hand, the ALU-libraries for general purpose supercomputers are designed specific to every application and are bound to be complex due to their generic (ALU) nature. Thus the general purpose supercomputers become unsuitable for handling simultaneous multiple applications.

All Functional Units are organized as 2-D Cell Arrays integrating 1 Bit SRAM as in 2.1 with every cell of the Functional Units. The control of both the processor and the memory are also integrated at the physical level. Based on extensive analysis of wide class of Algorithms the different types of ALFUs are organized into Segments. These segments are logically grouped into columns. There are eight columns each having four segments. Inter and intra column communication is provided using three stage non-blocking Clos interconnection network.

2.0.2 MIP Instruction set Architecture

The MIP (Memory In Processor) Instruction Set Architecture is a higher order ISA designed to suite the algorithm level capabilities of the MIP SCOC functional units.

The MIP SCOC ISA is referred as Algorithm Level Instruction Set Archi-

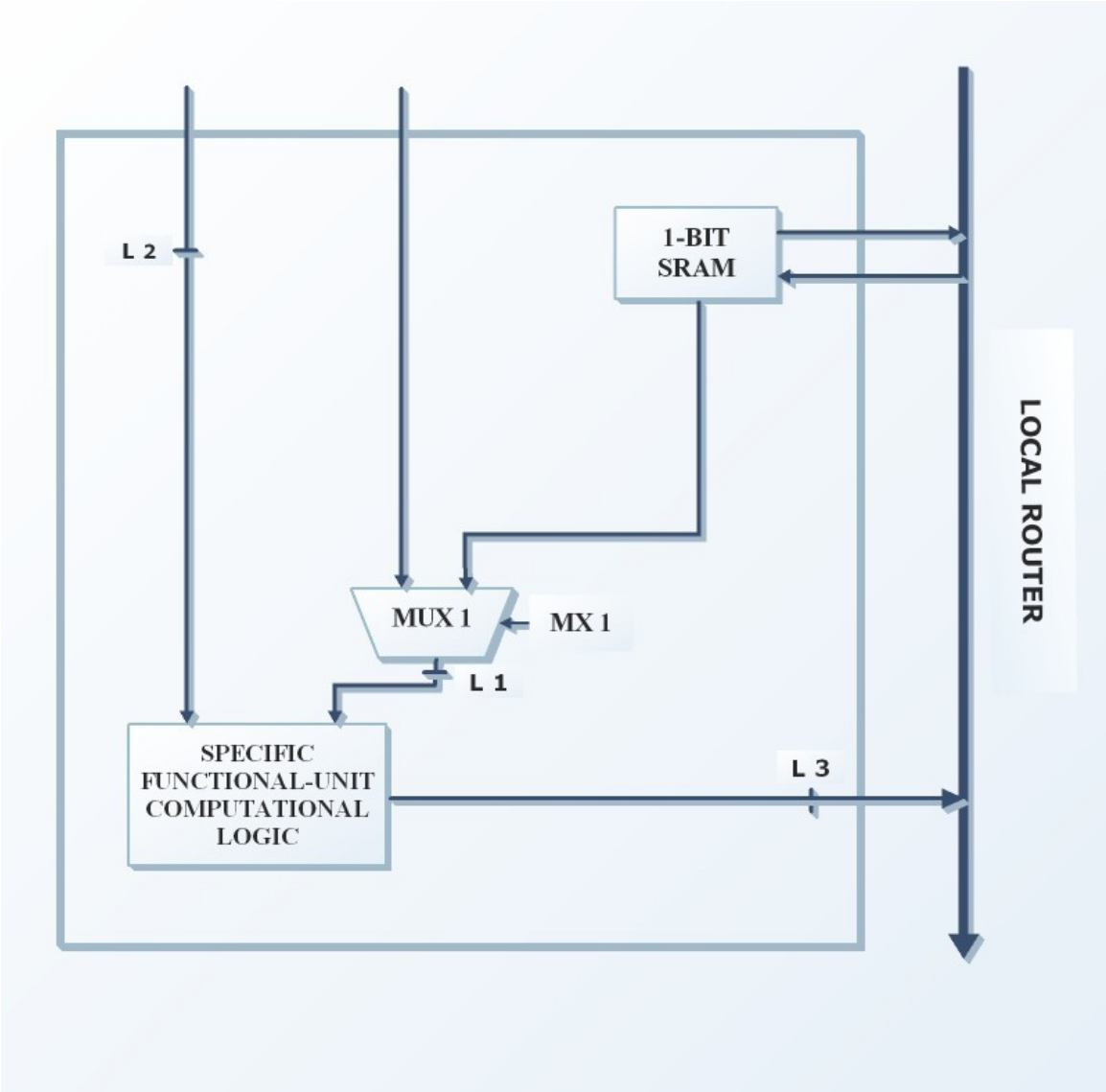

Figure 2.1: MIP Cell Architecture

tecture (ALISA) and is presented in shown below. The instructions of ALISA are categorized into Graph-theoretic, Vector-Related, Matrix Related, Scalar, ALFU - ALFU and SRAM - ALFU data transfer.

Every instruction in ALISA is a substitute for a large number of corresponding ALU instructions. Each instruction in ALISA is at a sub-algorithm level that gets executed on an ALFU directly.

The Memory In Processor Architecture aims at reducing the Power consumed by the chip. Firstly the instructions being at the Algorithm Level in the ALISA, the number of instructions gets reduced significantly when compared with the conventional architectures. Then, the fine grained integration of the memory with the processing elements in the architecture, along with the Algorithm Level Functional Units minimize the number of memory accesses. This decrease is mainly due to the reduction in the number of Instruction Fetches and the local availability of the intermediate data. The increased Memory processor bandwidth because of the integration of the memory with the processing elements helps in reducing the access time. The memory bandwidth is the one which decides the performance of the current day processors. The reduction in the number of memory accesses not only increases the performance but also brings down the power to a large extent. The architecture with its characteristic Low Power consumption suits the requirement of the On-Board Supercomputers.

ALGORITHM LEVEL INSTRUCTION SET ARCHITECTURE:

- Addition - Scalar Adder/Subtractor
- Subtraction - Scalar Adder/Subtractor
- Multiplication - Scalar Multiplier
- Division - Scalar Divider

o Sorting - Sorter -Ascending order -Descending order

o Square Root - Square Rooter

o Multi Operand Addition - Multiple Operand Adder

o Random Number Generation- Random Number Generator

o Find Max - Max/Min Finder

o Find Min - Max/Min Finder

o Inner Product - Inner Product Generator

o Graph Traversal - Graph Traversal Unit -Breadth First Search -Depth First Search - Hamiltonian Path

o Inter-Intra Adjacency - Inter-Intra Adjacency Unit - Inter Graph Adjacency - Intra Graph Adjacency

o Matrix Addition - Matrix Adder/Subtractor

o Matrix Subtraction - Matrix Adder/Subtractor

o Matrix Multiplication - Matrix Multiplier

o Chain Matrix Addition - Chain Matrix Adder

o Matrix Inversion - Matrix Inverter

o LU Decomposition - Crout Unit

o Comparison - Comparator - Equality Check - Greater Than or Equal To Check - Lesser Than or Equal To Check - Greater Than - Lesser Than

2.0.3 MIP Compiler On Silicon

In MIP SCOC, due to the presence of ALFUs, a rigorous mapping process is involved in scheduling ALISA instructions and balancing computations within the ALFUs population. This mapping in the MIP SCOC node is a complex process. To manage the complexity, a hardware based compiler [18] is needed matching the MIP SCOC performance. Compilers for superscalar processors were the first to incorporate implicit detection of instruction level parallelism and scheduling the same. The most often used libraries will exist in the ex-

ecutable form. These executable libraries are bound to involve billions of instructions. Though the library exists in executable form, resource allocation (as in superscalars) during execution will be a time consuming process. In particular, if the resources within a node as in MIP SCOC become heterogeneous and large in number, a software based methodology will be too tedious a process for making optimal resource allocation. Hence, we resort to a hardware based solution (mapping and scheduling) for resource allocations in the MIP SCOC reducing the compilation process delay.

This hardware means is achieved by a set of eight Secondary Compilers On Silicon (SCOS) mesh connected units driven by a Primary Compiler On Silicon (PCOS) unit. These PCOS and SCOS units posses a MIP based design similar to that of ALFUs. In order to enable concurrent instruction issue to the various columns, the functionality of the COS has been partitioned into two levels specifically as the primary and secondary compilers on silicon. These SCOSs function under a common host-compiler (Primary COS - PCOS), which control the allocation of the sub-libraries. The problem is allocated by the hosts (in a cluster environment) to the PCOS in terms of built-in libraries. The PCOS takes up the incoming set of libraries, decomposes them into sub-libraries and passes them to the respective SCOSs for execution within a particular column.

The PCOS takes care of mapping the problem to the different SCOSs that are under its control. The execution of the sublibraries is carried out by the SCOSs independent of each other and they communicate with the PCOS upon completion. As problem execution proceeds, communication between the SCOSs may be required to move the sub-library results across the columns for further execution. A mesh connected topology is employed for linking the eight SCOSs for efficient execution. The Basic node architecture of MIP SCOC is shown in Fig 2.2.

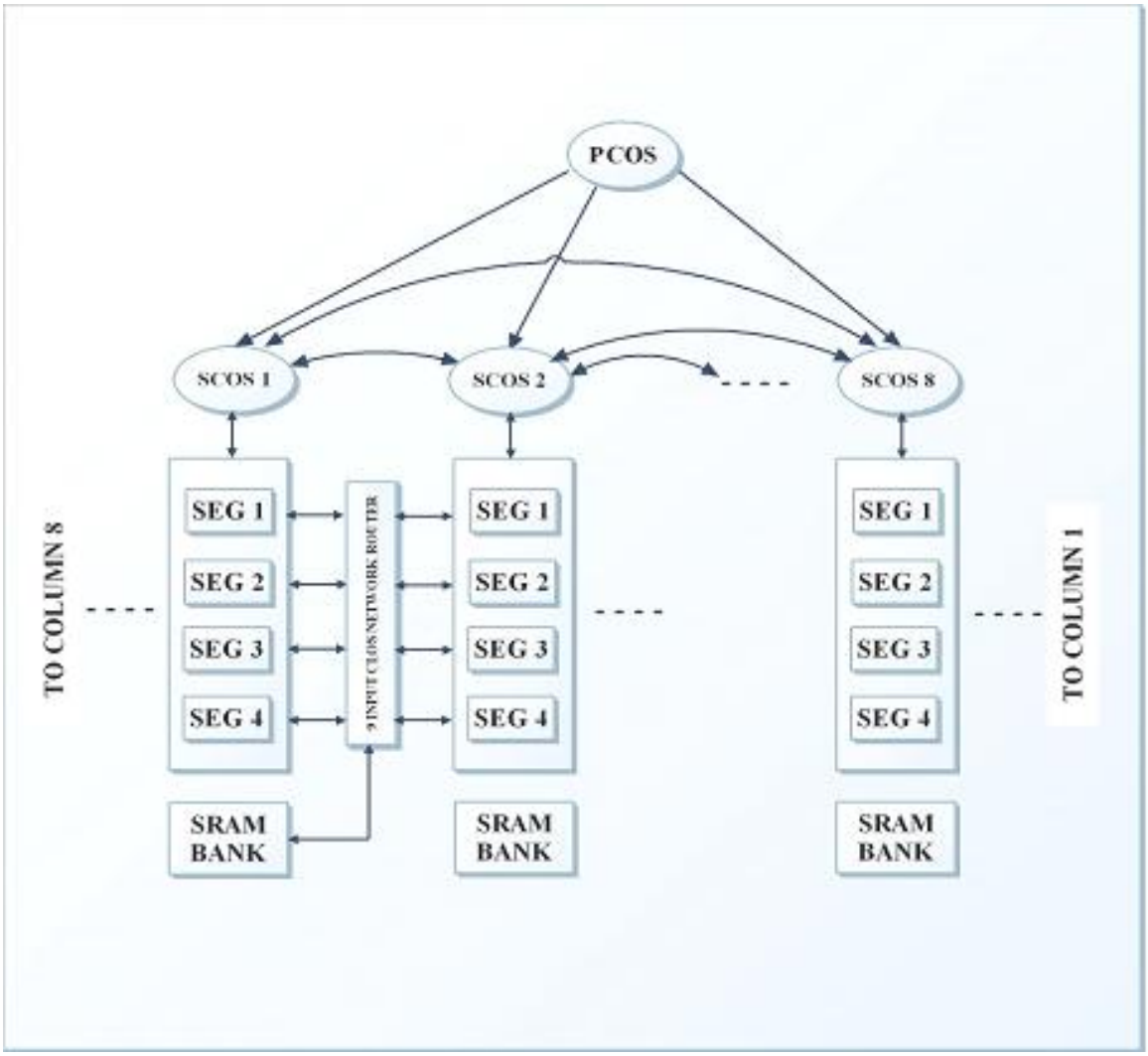

Figure 2.2: Compiler On Silicon organization

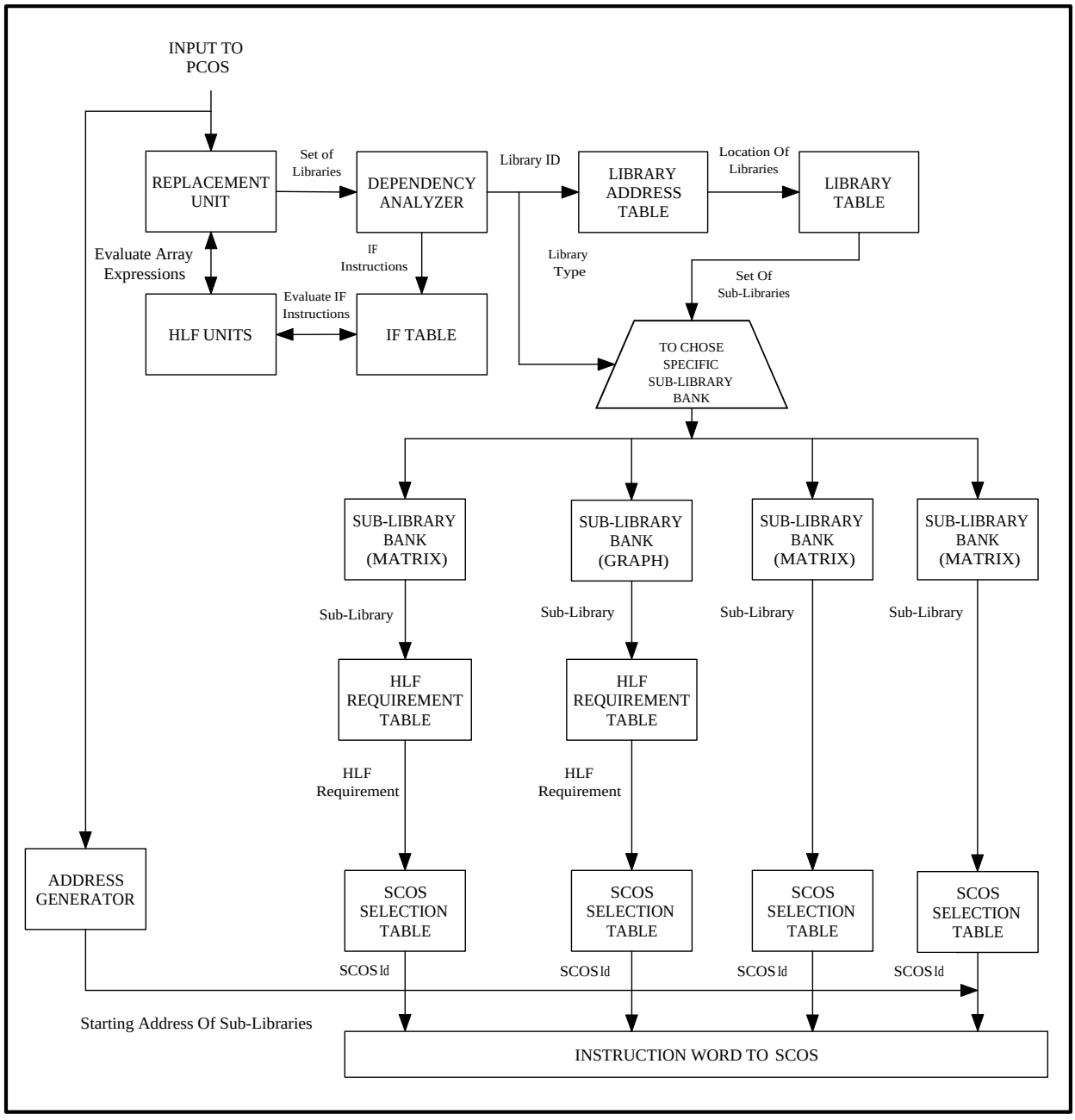

Figure 2.3: An overall Architecture of Primary Compiler On Silicon

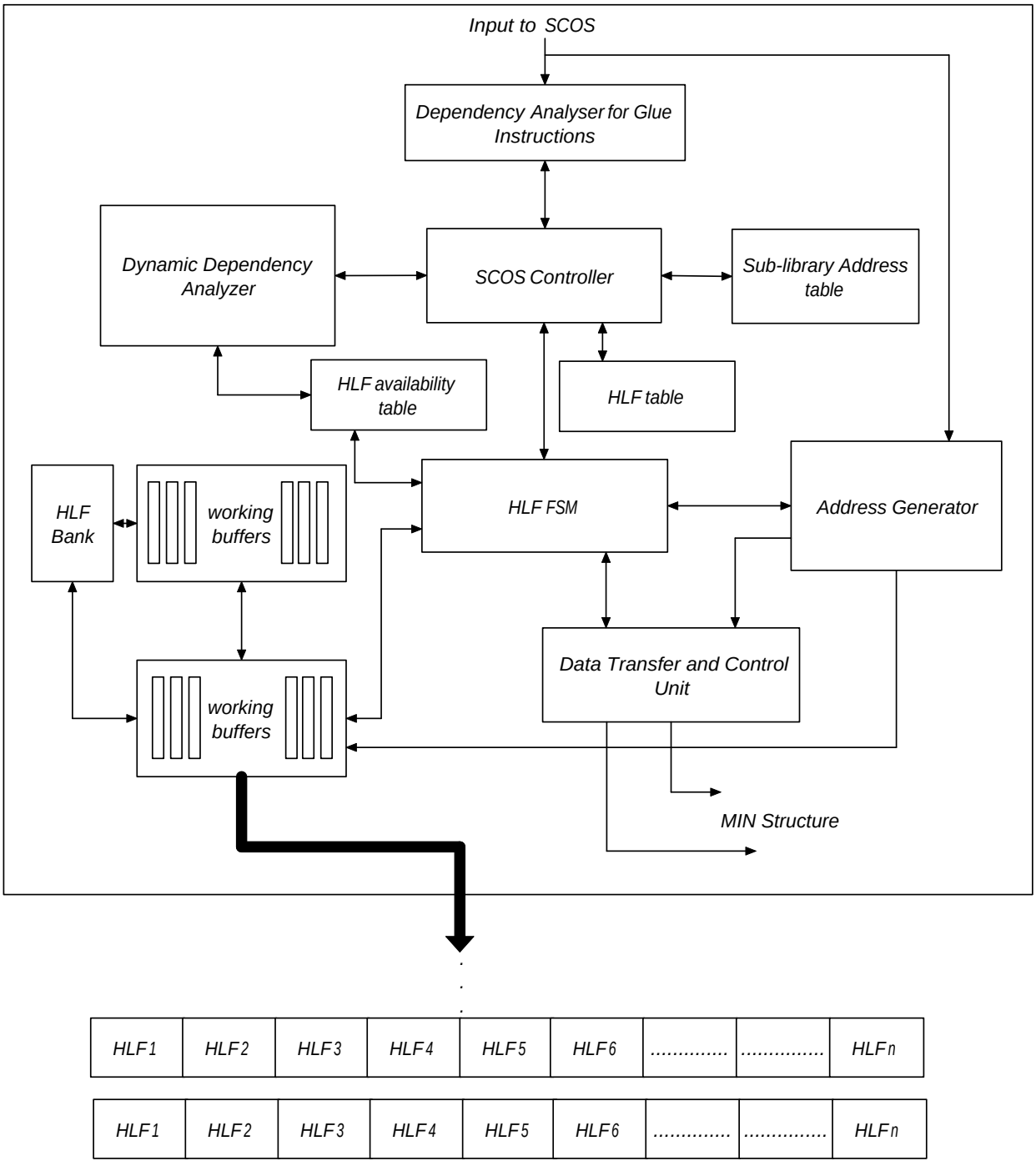

Figure 2.4: An overall Architecture of Secondary Compiler On Silicon

1) PCOS Library specification: The library specification within the PCOS provides details for decoding and performing the necessary operation. This specification provides information about the operation to be performed, logical schedule time instants, dependencies and ALFU units required. In the PCOS, the library is broken into sub-libraries. When multiple sub-libraries are associated with the same time-stamp, they are scheduled in parallel. Based on the SCOS resource constraints, PCOS might schedule the sub-libraries over multiple cycles.

2) SCOS Sub-Library specification: The sub-libraries are executed within a specific column controlled by a SCOS. These sub-libraries break into ALISA instructions that perform the required operation. A single instruction word can trigger multiple ALFU units simultaneously. Data movement within a column occurs due to dependency between the ALFU instructions. Based on operation the appropriate Move instruction triggers transfer within or across segments.

2.0.4 ALFU (Algorithm Level Functional Units)

The instruction control words for executing the algorithm level instructions in the respective ALFUs present in the column segments are issued by the different SCOSs. These instruction control words are generated with their associated operands and their corresponding ALFU id. These instruction control words trigger the ALFU's local controller and the necessary operations are performed to carry out the execution of the algorithm level instruction. To simplify the decoding process of identification of ALFU unit types, each ALFU type has a field in the instruction control word. If a particular ALFU unit is not being triggered for a particular operation, a 'NOOP' fills up that field.

Chapter 3

MIP Cluster Architecture

The simultaneous application mapping on a million node cluster needs an efficient host system to tackle the complexity involved. The whole thesis concentrates on the operating system design to tackle the problem of the 'mapping complexity' on a million node cluster with the added overhead of simultaneous multiple application mapping. The presence of abundant resources at the node level is not the only criterion to achieve performance. To attain sustained performance at the node level the mapping at the host level has to be time efficient and be capable of exploiting the amount of parallelism that can be harnessed at the node level. This necessitates a need for parallel and hierarchically based multiple host system. The MIP SCOC cluster [17][1] presented in 3.1 readily meets these requirements. The MIP SCOC cluster has been envisaged for beyond exaflops scale applications. A single host on a MIP SCOC cluster cannot run the entire system, since the parallel SMAPP mapping time complexity, would become a major bottleneck. Also, for a single host, to meet the feed rate for the computing MIP-nodes is an intricate task. The architecture of the MIP cluster has been proposed to be a hybrid pyramid cluster.

Three stages of problem decomposition have been evolved to efficiently map the applications onto the MIP cluster. These stages are represented as computational planes in the hybrid pyramid cluster architecture as shown in Figure

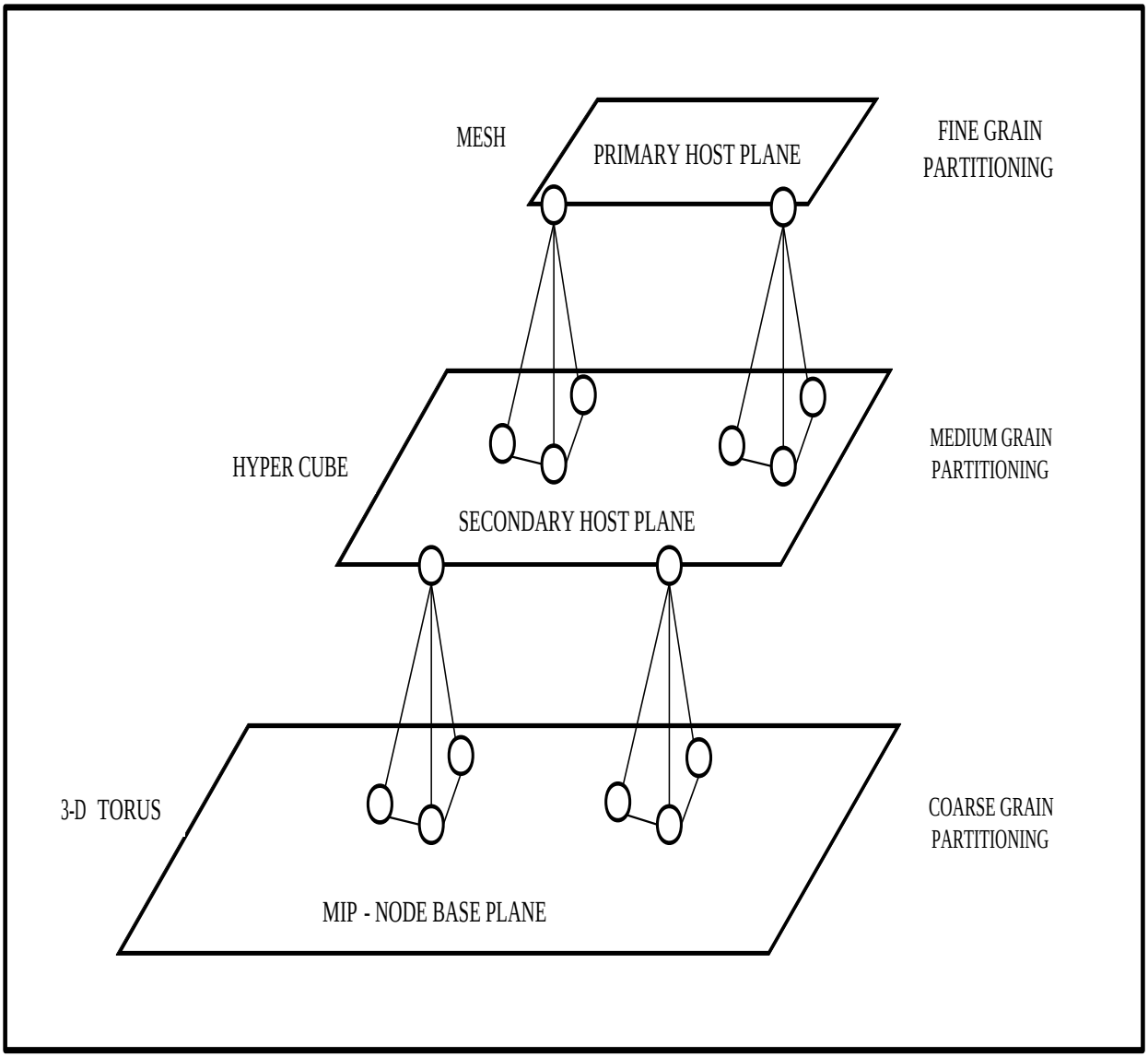

Figure 3.1: Hierarchical computational planes of MIP SCOC

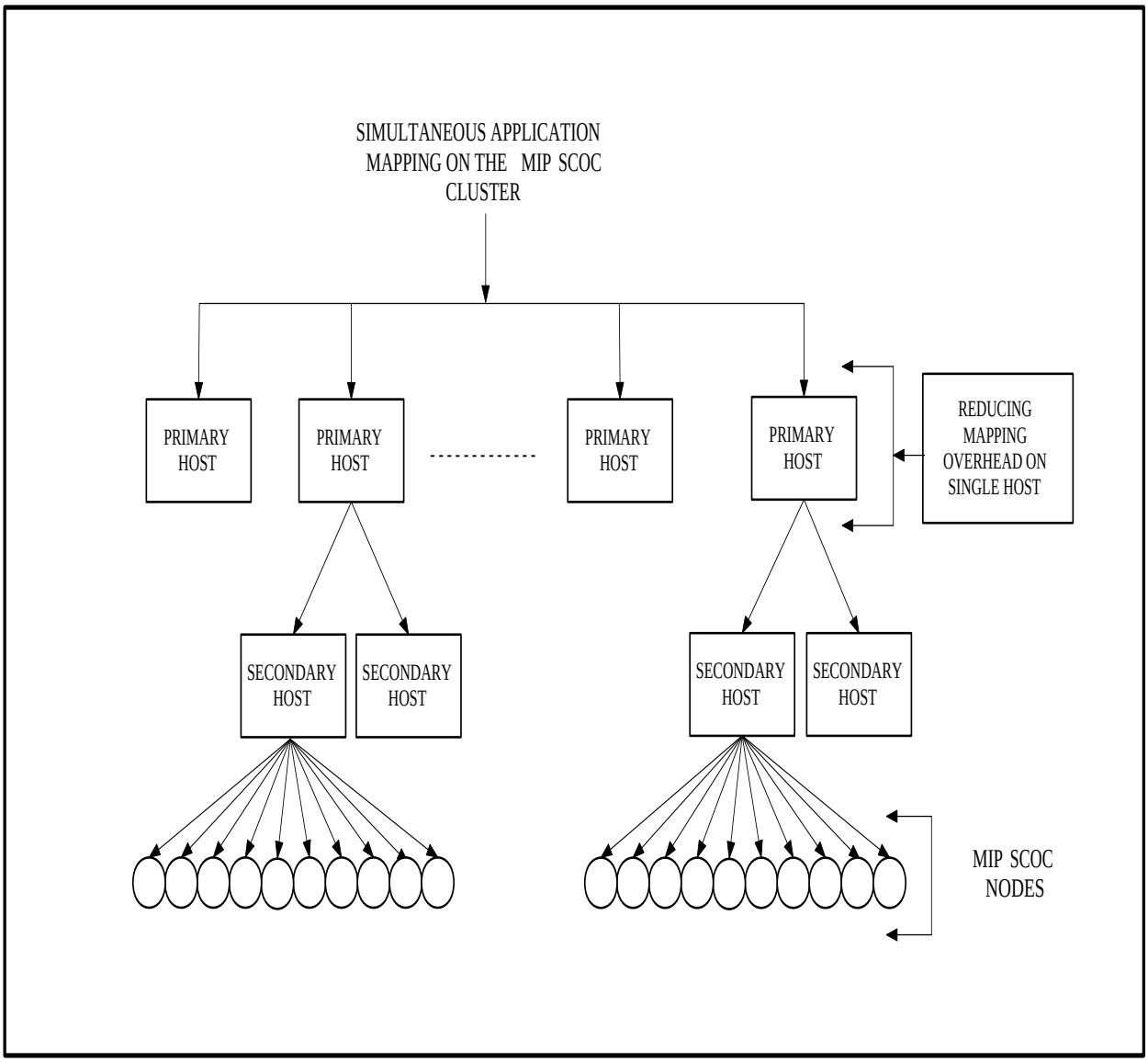

Figure 3.2: Overview of the MIP SCOC cluster

3.1. Three different topologies are employed at the aforesaid three stages to support efficient problem decomposition. This has been represented in the Figure 3.2. Each stage of the cluster does a part of the application mapping for efficient problem execution. The applications are presented to the MIP cluster at the primary host plane. This plane has a mesh-connected Inter-Connection Network (ICN). The primary hosts perform parallel decomposition of the applications into their constituent sub-problems. This coarse-grain partitioning requires close interaction among the primary hosts to exchange mapping information. Frequent exchanging of control information involving the current mapping status amongst the primary hosts is important for them to work cohesively. Hence these primary hosts are to be well connected.

The second plane of the MIP cluster hierarchy involves of set of secondary hosts under each of the primary hosts. The secondary hosts are interconnected by a hypercube topology. They further decompose the sub-problems into libraries and allocate them to the MIP SCOCs. Since the secondary hosts must communicate well for the efficient exchange of data across them, the hypercube topology fits well.

The third plane of the hierarchy consists of the MIP SCOCs, interconnected through a 3-D torus topology. 3-D torus provides an efficient communication topology for exchanging data amongst themselves. Since each of the SCOCs handles a significant amount of computations, the data input to them is higher. Therefore the communication requirements are higher and only the 3-D torus can efficiently scale up to meet these requirements.

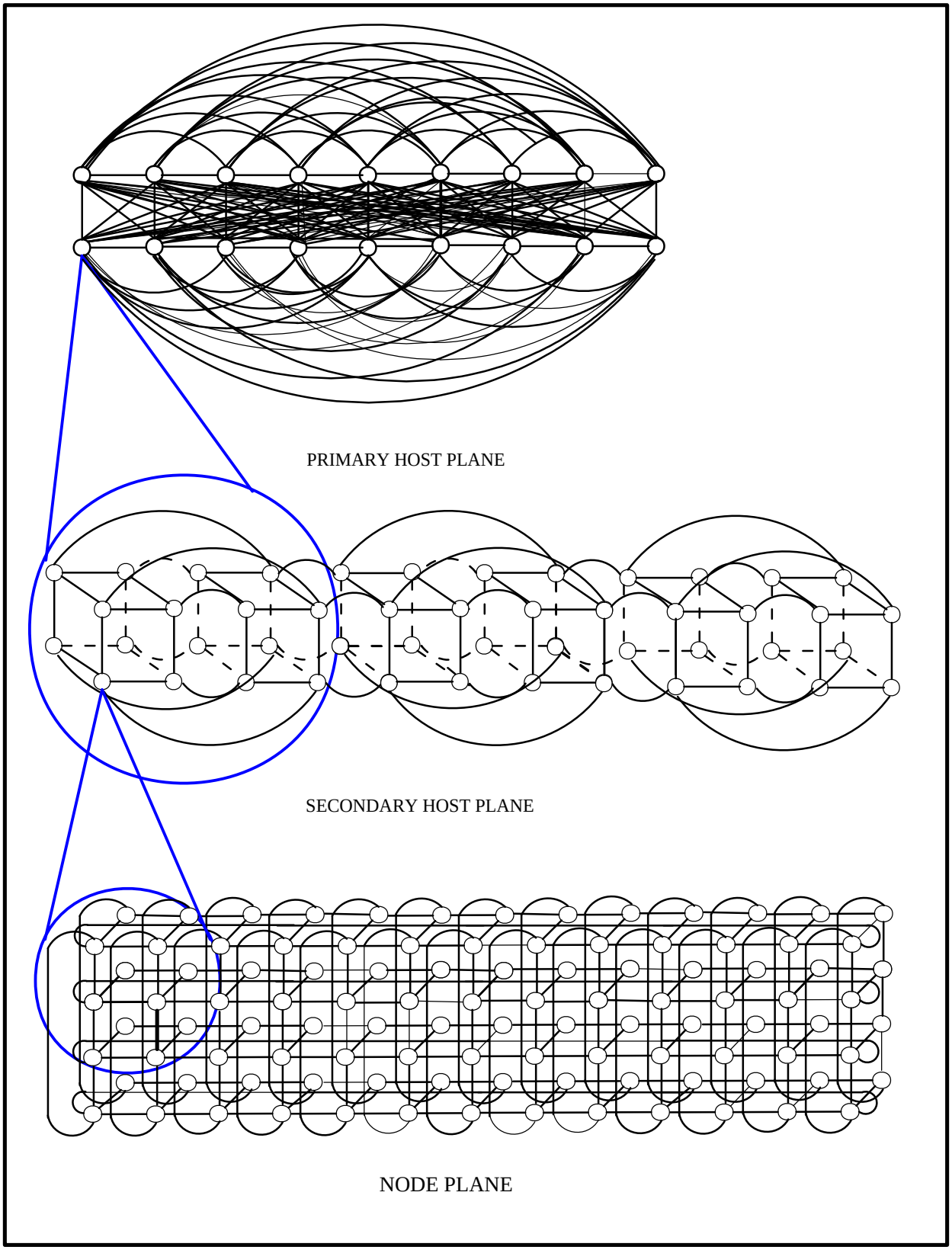

Figure 3.3: Hierarchical Topology Control Model

3.0.5 The MIP Cluster Operating System

The single factor limiting the harnessing of the enormous computing power of clusters for parallel computing is the lack of appropriate software. Present cluster operating systems are not built to support parallel computing - they do not provide services to manage parallelism. The cluster operating environments that are used to assist the execution of parallel applications do not provide support for either Message Passing (MP) or Distributed Shared Memory (DSM) paradigms. They are only offered as separate components implemented at the user level as library and independent servers. In the existing cluster operating systems, users must deal with computers of a cluster rather than to see this cluster as a single powerful computer. A Single System Image of the cluster is not offered to users. There is a need for an operating system for clusters. We claim that it is possible to develop a cluster operating system that is able to efficiently manage parallelism, support Message Passing and DSM and offer the Single System Image.

Chapter 4

Process Scheduling in MIP SCOC Cluster

The input problem to the cluster is fed at the primary host level through different I/O ports available. The user is allowed to choose the number of input points for the application injection based on the size of the input. The input problem should be in the MIP Parallel Programming language enabling the direct execution of the application.

To enable the execution of applications already coded in C, C++, FORTRAN etc., a MIP trans-compiler can be used to convert the applications directly into the Parallel programming language. Because of the Algorithm Level Functional units of the MIP node, this process cannot be fully automated. With a little of manual intervention, the applications are converted into a Pseudo Library format and then into the constituent PPL instructions.

The MIP parallel programming language has high level instructions (i.e) at an abstract level which may be the different algorithms of the applications. The user codes the applications with these instructions specifying the problem size of the algorithms. This makes the process of coding very simple. For example, when a user codes an application which involves matrix opera-

tions, he will be using direct instructions like LUD (LU Decomposition), SVD (Singular Value Decomposition), MATMUL etc with the specification of the problem size. The parallel programming language is facilitated with constructs for specifying the execution schedule along with the code so that it aids the compiler in the process of parallelizing and mapping the code. The schedule is spatio-temporal and specifies where and when the different instructions of the code has to mapped so that the code can be executed in minimum time. The user should also specify the starting points of execution in the code that is fed at the primary host level. The parallel programming language has constructs to specify the starting point of execution of the code.

4.0.6 Mix Formation At The Primary Host Level

The code that the user inputs is stored in the secondary memory of the hosts. The code is given as input to the PPL compilers at the different nodes in which they were fed as input by the user. These compilers check for the various lexical, syntax and semantic errors in the code. The syntax trees for the different tokens are all stored in all the Primary hosts for easy compilation. The syntax errors that are generated by the compiler are all returned back to the user at the different I/O points through which the code was entered.

The user is given choice for parallel debugging based on the debugger available. These debuggers are programs that run in the primary hosts which are capable of finding errors in the parallel programming language code

After the errors are resolved, based on the constructs of the parallel programming language the different algorithms are assigned application ID for future referencing of the output. These application IDs are generated based on a communication methodology between the different primary hosts. The

Primary hosts need to maintain consistency among the different application IDs that are generated. For this purpose, the application IDs are generated based on the user's input of the number of applications. The numbers varying from 0 to n-1 are assigned based on the user's choice for the applications. Then the sub problems are assigned IDs based on the communication methodology defined. Here the primary hosts that receive the code themselves assign the sub-problem IDs which is an extension of the application ID based on the demarcations of the PPL constructs. If one application is entered through more than one primary hosts, the user takes the responsibility of informing the primary hosts about the parts of the applications.

The parallel programming language constructs are in such a manner that the parallel modules are clearly demarcated. Thus the cohesively dependent set of PPL code (at the sub-problem level) within one application, calling it a C-Module can be clearly identified. A rough estimate of execution time of these C-Modules is also quantized based on the critical path of the module. Based on the libraries associated with the different C-Modules, the amount of computations involved is also approximated and characterized based on the different functional cores of the node. Now, based on the Mix templates generated by the simulator, the primary host starts with the mix formation procedure. The Templates are stored as conditionals of what all algorithms (PPL instructions) can be put together for one time stamp of execution, so that they effectively use the underlying architecture core for maximum throughput. For example, a matrix algorithm, a graph theoretic algorithm, a scalar algorithm etc can be grouped together so that they execute in parallel and give maximum resource utilization within a node. Thus the different C-Modules of different applications are grouped together under one primary host so that it enables efficient mapping thru the secondary host plane and finally resulting

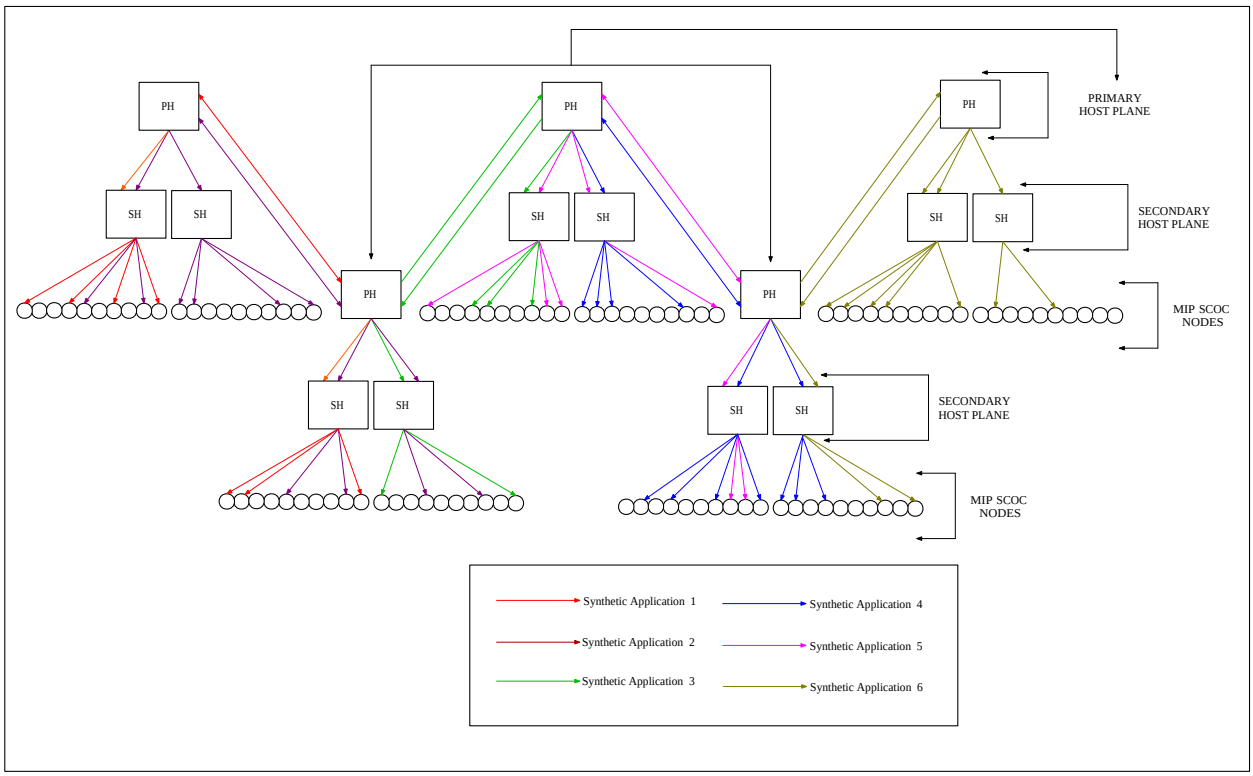

Figure 4.1: Flow of Applications in the Hierarchical Cluster

in increased throughput at the node plane. This process of blending the C-modules is done as a sender based load balancing in which the C-Modules are distributed spatio-temporally across the different primary hosts for generating efficient mixes which are capable of getting executed for the whole critical path of the longest C-Module and as well they give sustained performance (throughput) at the node level throughout the execution.

Firstly, the set of primary hosts which get the input application classify the C-Modules based on the rough time to execute their critical path. The independent C-Modules of similar execution times are grouped together and they are distributed to the neighboring primary hosts. This takes place as a sender initiated load balancing. After good amount of load balancing, the primary hosts proceed with the process of mix formation based on the Mix templates available as a result of the simulator. In this process of blending,

the primary hosts assign a vector characterized by the functional cores that are present in the underlying node architecture showing the amount of computations for every PPL module. This can be done because the libraries of the parallel programming language instructions are already assigned with these vector values along with a rough estimate of their execution time.

Now, they firstly start adjusting the different C-modules temporally within themselves to attain minimum variance in the vector values for every time cycle of execution. If the mean of these variances is found to be higher than the threshold value, they proceed with the process of exchanging C-Modules. For every exchange, they find the variance at every time stamp among the different vectors and based on the mean of these variances, they find whether the new mix is efficient or not based on the info available as mix templates. This takes place for a specified amount of time till every primary host is replenished with C-Modules capable of executing together and giving maximum thought put.

Now, Every such primary host splits its load into n different strands of execution where n is the number of secondary hosts in one primary host. This process of strand formation aims at a situation where every strand in the primary host is said to have minimum variance of the vector values at every time stamp of execution. This process is done by once again altering the C-modules temporally and spatially between the different strands.

The primary and the secondary hosts architectures are designed in such a way that they are capable of executing the mapping algorithms that the user uses. For this purpose, they are designed as generic Superscalar ALU based architectures capable of executing the mapping strategy that the user chooses. For this purpose, we will discuss a few mapping strategies that can be incor-

porated in mapping.

The mapping problem can be defined as the placement of communicating processes on processors of a distributed memory parallel machine. A survey of the different methods proposed in the literature to deal with this problem may be found in [19]. The mapping problem is a combinatorial optimization problem which can be reduced to the graph partitioning problem, shown to be NP-complete in [20]. The mapping problem is then NP-complete. Consequently, heuristic methods should be used to deal with it. They may find solutions that are only approximations of the optimum, but they will do it in a reasonable amount of time. An example for such heuristics is genetic algorithms.

The different allocation strategies that have been proposed in the literature are based on one of these approaches: mathematical programming [4], graph theory [21], queuing theory [22] and heuristics Figure 4.2. The first three approaches give optimal solutions but are time consuming, given that the problem is NP-complete. To speed up the search, approximate algorithms have been used; they are based on one of the above optimal approaches but are limited by the time search used [23]. Another solution to the problem is the utilization of heuristics. They may be divided in two categories: greedy and iterative. The greedy algorithms are initialized by a partial solution and search to extend this solution until a complete mapping is achieved [24]. At each step, one process assignment is done and we can't change this decision in the remaining steps. Iterative algorithms are initialized by a complete mapping and search to improve it by moving a process to another processor or by exchanging the mapping of two processes.

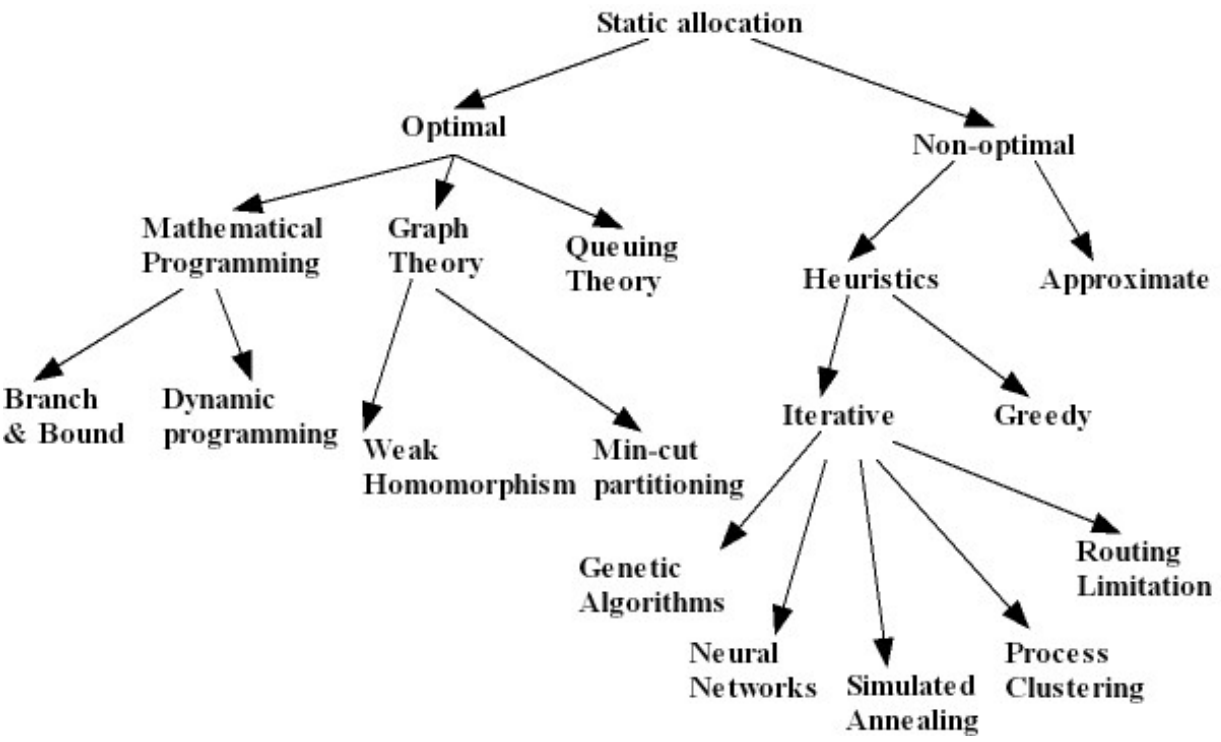

Figure 4.2: Taxonomy of Allocation Methods

4.0.7 Allocation Strategies - Genetic Algorithms based Approach

Genetic algorithms are stochastic search techniques, introduced by Holland twenty years ago [25], inspired by adaptation in evolving natural systems. Development of massively parallel architectures made them very popular in the very last years. They have recently been applied to combinatorial optimization problems in various fields, such as, for instance, the traveling salesman problem [26], the optimization of connections and connectivity of neural networks [27], and classifier systems [28]. Standard genetic algorithms with large populations take an extremely long time to execute but nowadays people have come up with parallel algorithms [19] to speed up the genetic process.

Two approaches to parallel genetic algorithms may be considered:

Standard parallel approach:

In this approach, the evaluation, the reproduction and the replacement are done in parallel. However, the selection is still done sequentially, because parallel selection would require a fully connected graph of individuals as any two individuals in the population may be mated.

Decomposition approach:

This approach consists in dividing the population into equal size sub-populations. Each processor runs the genetic algorithm on its own sub-population, periodically selecting good individuals to send to its neighbors and periodically receiving copies of its neighbors' good individuals to replace bad ones in its own sub-population [29]. The processor neighborhood, the frequency of exchange and the number of individuals exchanged are adjustable parameters.

The standard parallel model is not flexible in the sense that the communication overhead grows as the square of population's size. Therefore, this approach is not adapted to distributed memory parallel architectures, where the cost of communication has a great impact on the performance of parallel programs. In the decomposition model, the inherent parallelism is not fully exploited as treatment of sub-populations may be further decomposed.

4.0.8 Allocation Strategies - Simulated Annealing based Approach

The simulated Annealing algorithm is taken as a case study in the explaining the process of static scheduling in the section Library Design. The illustration is given as a figure General Algorithm for Simulated Annealing.

Here is a simple illustration of the implementation of Applying Genetic Algorithms for the above given mapping problem.

Say, this is one critical Module

Time stamp 1 - 12 23 4 56
Time stamp 2 - 5 58 2 26
. 12 14 32 03
. 82 02 23 26
. 25 02 32 12

Critical Module 2:
25 23 26 25
21 32 65 54
02 23 32 5
12 32 45 21
12 95 32 12

Similarly say there are 100 critical modules,

Now, one can put them all together to form one group like,

cp1 cp3 cp5 cp54 cp23 cp12 cp50 cp8 cp32 cp82

So, there will be some 10 groups without any Critical Modules repeating, which is a sample solution.

The Fitness function can be calculated as follows,

I take every group, say the example group above,

time stamp 1 - 12 23 4 56	CP2 - 25 23 26 25	cp3 - cp10
time stamp 2 - 5 58 2 26	21 32 65 54	
12 14 32 03	02 23 32 5	
82 02 23 26	12 32 45 21	
25 02 32 12	12 95 32 12	

Now for time stamp 1, add all the values of the first vector say 12+25+..... Similarly for time stamp 2 - time stamp 5.

So, finally a group will be like this,

time stamp 1 - 452 236 124 263

Figure 4.3: GA based Approach Illustration

time stamp 2 - 238 22 362 365
124 323 66 323
124 23 326 125
124 235 362 263

Now for every timestamp, Find the variance,

So, a groups looks like this

125
23
236
232
21

This is the fitness function, i.e., the variance in each should b minimum for every group of the 10 groups by exchanging the critical paths between them. The mean of all the variances can also be fixed as the fitness function.

Illustration for Temporal adjustment of the Critical paths:

Critical Module 1:

Time stamp 1 - 238 235 200 187
Time stamp 2 - 02 12 500 600

Critical Module 2:

Timestamp 1 - 122 500 0 0
Timestamp 2 - 12 325 236 236

In combining them, if we combine the second one pushed one timestamp down, they will suit well in the blend.

Figure 4.4: GA based Approach Illustration 1

ALGORITHM:

First assign an unique id for each of the critical Modules.

cm----1
12 23 4 56
5 58 2 26
12 14 32 03
82 02 23 26
25 02 32 12

cm----2
12 23 4 56
5 58 2 26
12 14 32 03
82 02 23 26
25 02 32 12

The Chromosome is defined as the following,

"1(1) 2 4 -1 5(2) 3 6"

Which means that the first group has the Critical Modules of 1, 2, 4, the Second Group with the critical modules 5,3,6. The numbers in the bracket show the number of time delays that has been added to the Critical Module in the process of temporal adjustments. The symbol '-1' is used as the separator between the Critical Modules. There will be n-1 "-1"s for n groups.

Figure 4.5: GA based Approach Illustration 2

Operators:

Shift down.. Temporal--Mutation
1(1) 2 4 -1 5(2) 3 6
*-> here the shift is changed from 2 to 1.This position is determined at random.

1(1) 2 4 -1 5(1) 3 6
*-> Exchange: spatial- Swapping Mutation
1(1) 2 4 -1 5(2) 3 6

Swapping random positions....2 and 4

1(1) 2 5(2)-1 4 3 6

Cycle Crossover

4 3 1-1 5 2 6
1 2 4 -1 5 3 6

Remove -1s

4 3 1 5 2 6
1 2 4 5 3 6

Random generated locci.2

4 3 1 5 2 6
1 2 4 5 3 6

4 3 1 5 2 6
1 2 4 5 3 6

This forms a cycle. And the rest of it is swapped.

1 3 4 5 2 6
4 2 1 5 3 6

Figure 4.6: GA based Approach Illustration 3

General Simulated Annealing (SA) algorithm

$S \Leftarrow$ Initial Solution S_0
$T \Leftarrow$ Initial Temperature T_0
repeat
  repeat
    $S' \Leftarrow$ Some random neighboring solution of S
    $\Delta \Leftarrow C(S') - C(S)$
    $Probability \Leftarrow min(1, e^{\frac{-\Delta}{K_b T}})$
    if $random(0,1) \leq Probability$ **then**
      $S \Leftarrow S'$
    end if
  until some condition is met
  Update T
until some condition is met
Output Best Solution

Figure 4.7: General Algorithm for Simulated Annealing

4.0.9 Allocation Strategies - Greedy Approach

Greedy Method is one of the approaches in which the primary hosts go about exchanging the code modules based on the communication complexity of the individual by using a minimum spanning tree algorithm like the prim's algorithm or the Dijkstra's algorithms.

Then after the different assignments at the secondary host level is over, the mapping at the node level is done based on the schedule that is present inherently in the code. In case of static scheduling, the Primary host takes up the responsibility of sending the corresponding static schedule associated with the different code blocks to the different secondary hosts.

4.0.10 Library Design

The next step in the process of execution is decomposition of these High level instructions into the corresponding library level instructions by the different secondary hosts. Normally, the libraries are statically scheduled based on the node architecture and the cluster Interconnect Topology. This is done with the help of a simulator which uses Simulated Annealing in the process of finding the best schedule for the libraries which gives the maximum resource utilization and efficiency in execution. The algorithm proceeds by first making a random assignment of the libraries. Then these library instructions are exchanged based on a few heuristics and then the mean for load and the mean for the buffer length are computed. Based on the effectiveness of the decrease in the load or the buffer length, the exchange is made or the system reverts back to the original state. Thus the system goes through the process of hill climbing i.e. a search through the solution space in a random manner. Finally based on a time limit, the process of simulated annealing comes to an end. Now the schedule or the mapping that the algorithm proposes is fixed as the static schedule for those set of libraries. These schedule information are added to the libraries with the help of the parallel programming language constructs where the nodes are given relative referencing from one central node.

Based on the parallel constructs, the secondary hosts have to parse through the various scheduling information and then map the instructions accordingly. The libraries need to be transferred first following which the associated data should also be transferred. The libraries and the data need to be packeted before they are transferred to the corresponding nodes. In this process of packet formation, the various error correction codes need to be added along with the header and the trailer of the packet.

Chapter 5

Overview of the Linux Kernel

Some of the text presented here is compiled together from existing sources. In my humble opinion, it would be pointless to try to formulate concepts in my own words that other more gifted authors have done clear and precise. The sections Purpose of the Kernel and Overview of the Kernel Structure are taken from Conceptual Architecture of the Linux Kernel [30] by Iwan T. Bowman.

5.0.11 Purpose of the Kernel

The Linux kernel presents a virtual machine interface to user processes. Processes are written without needing any knowledge of what physical hardware is installed on a computer – the Linux kernel abstracts all hardware into a consistent virtual interface. In addition, Linux supports multi-tasking in a manner that is transparent to user processes: each process can act as though it is the only process on the computer, with exclusive use of main memory and other hardware resources. The kernel actually runs several processes concurrently, and is responsible for mediating access to hardware resources so that each process has fair access while inter-process security is maintained.

5.0.12 Overview of the Kernel Structure

The Linux kernel is composed of five main subsystems:

- The Process Scheduler (SCHED) is responsible for controlling process

access to the CPU. The scheduler enforces a policy that ensures that processes will have fair access to the CPU, while ensuring that necessary hardware actions are performed by the kernel on time. ”

- The Memory Manager (MM) permits multiple processes to securely share the machine's main memory system. In addition, the memory manager supports virtual memory that allows Linux to support processes that use more memory than is available in the system. Unused memory is swapped out to persistent storage using the file system then swapped back in when it is needed.

- The Virtual File System (VFS) abstracts the details of the variety of hardware devices by presenting a common file interface to all devices. In addition, the VFS supports several file system formats that are compatible with other operating systems.

- The Network Interface (NET) provides access to several networking standards and a variety of network hardware.

- The Inter-Process Communication (IPC) subsystem supports several mechanisms for process-to-process communication on a single Linux system.

The figure 5.1 Kernel Subsystem Overview shows a high-level decomposition of the Linux kernel, where lines are drawn from dependent subsystems to the subsystems they depend on.

This diagram emphasizes that the most central subsystem is the process scheduler: all other subsystems depend on the process scheduler since all subsystems need to suspend and resume processes. Usually a subsystem will suspend a process that is waiting for a hardware operation to complete, and

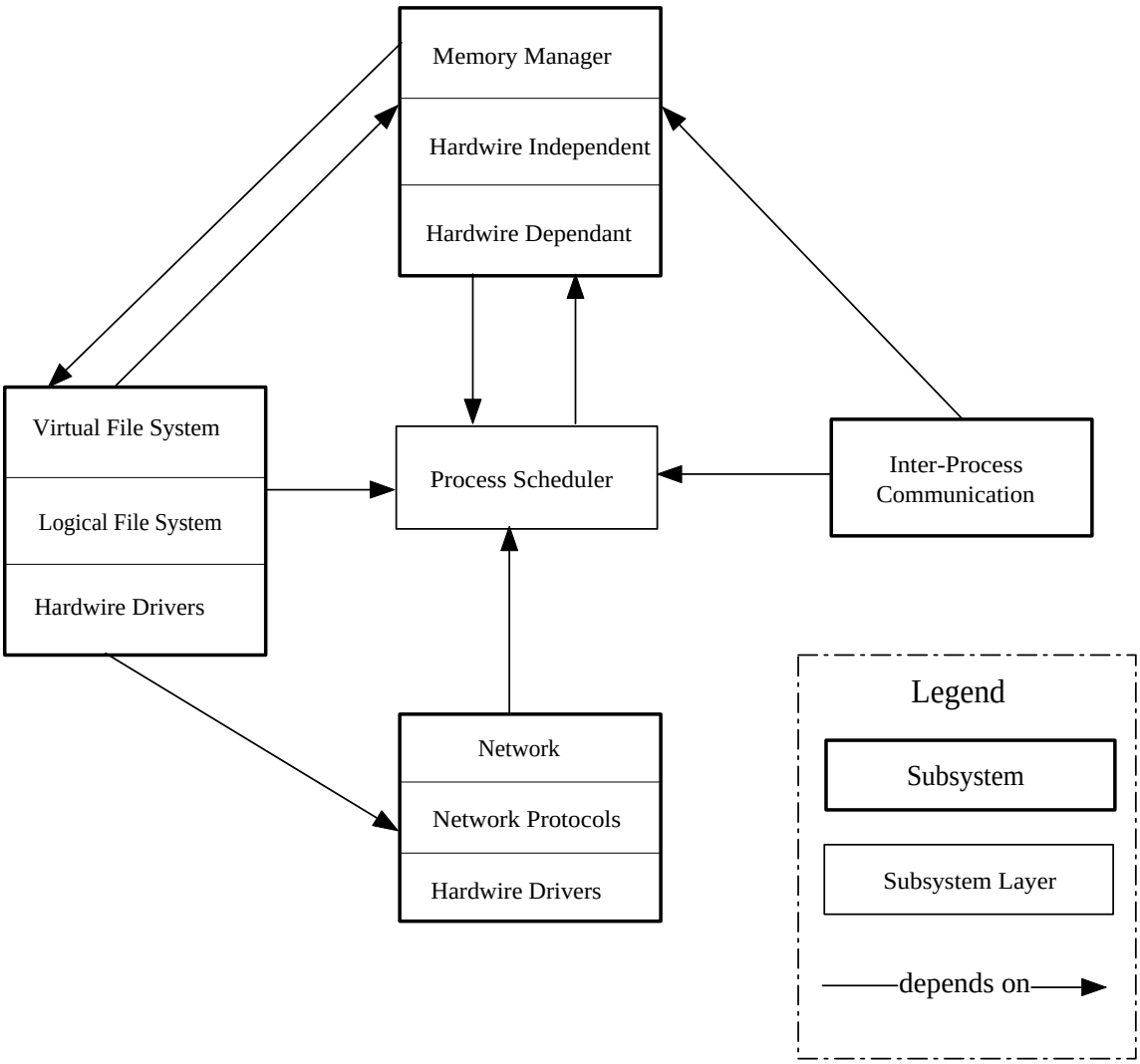

Figure 5.1: Linux Kernel Structure

resume the process when the operation is finished. For example, when a process attempts to send a message across the network, the network interface may need to suspend the process until the hardware has completed sending the message successfully. After the message has been sent (or the hardware returns a failure), the network interface then resumes the process with a return code indicating the success or failure of the operation. The other subsystems (memory manager, virtual file system, and inter-process communication) all depend on the process scheduler for similar reasons.

This diagram emphasizes that the most central subsystem is the process scheduler: all other subsystems depend on the process scheduler since all subsystems need to suspend and resume processes. Usually a subsystem will suspend a process that is waiting for a hardware operation to complete, and resume the process when the operation is finished. For example, when a process attempts to send a message across the network, the network interface may need to suspend the process until the hardware has completed sending the message successfully. After the message has been sent (or the hardware returns a failure), the network interface then resumes the process with a return code indicating the success or failure of the operation. The other subsystems (memory manager, virtual file system, and inter-process communication) all depend on the process scheduler for similar reasons.

The other dependencies are somewhat less obvious, but equally important:

- The process-scheduler subsystem uses the memory manager to adjust the hardware memory map for a specific process when that process is resumed.
- The inter-process communication subsystem depends on the memory

manager to support a shared-memory communication mechanism. This mechanism allows two processes to access an area of common memory in addition to their usual private memory.

- The virtual file system uses the network interface to support a network file system (NFS), and also uses the memory manager to provide a ramdisk device.

The memory manager uses the virtual file system to support swapping; this is the only reason that the memory manager depends on the process scheduler. When a process accesses memory that is currently swapped out, the memory manager makes a request to the file system to fetch the memory from persistent storage, and suspends the process.

5.0.13 Process Scheduling

Like every other time sharing system, Linux achieves the magical effect of an apparent simultaneous execution of multiple processes by switching from one process to another in a very short time frame. The Scheduling policy introduces the choices made by Linux in the abstract to schedule processes. The scheduling algorithm discusses the data structures used to implement scheduling and the corresponding algorithm. Finally, the system calls related to scheduling describes the system calls that affect the process scheduling.

5.0.14 Scheduling Policy

The scheduling algorithm must fulfill many conflicting objectives like fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low and high priority processes, and so on. The set of rules used to determine when and how to select a new process to run is called scheduling policy.

Linux Scheduling is based on the time sharing technique: several processes run in "time multiplexing" because the CPU time is divided into slices, one for each runnable process." Of course, a single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or quantum expires, a process switch may take place. Time sharing relies on the timer interrupts and is thus transparent to the processes. No additional code needs to be inserted in the programs to ensure CPU time sharing.

The scheduling policy is also based on ranking processes according to their priority. Complicated algorithms are sometimes used to derive the current priority of a process, but the end result is the same: each process is associated with a value that tells the scheduler how appropriate it is to let the process run on a CPU.

In Linux, process priority is dynamic. The scheduler keeps track of what processes are doing and adjusts their priorities periodically; in this way, processes that have been denied the use of a CPU for a long time interval are boosted by dynamically increasing their priority. Correspondingly, processes running for a long time are penalized by decreasing their priority.

When speaking about scheduling, the processes are basically classified as CPU bound and I/O bound. The I/O processes make heavy use of the I/O devices and spend most of their time in waiting for I/O operations to finish. The CPU bound applications are mostly number crunching applications which require a lot of CPU time.

5.0.15 Process Preemption

When a process enters the TASK-RUNNING state, the kernel checks whether its dynamic priority is greater than the priority of the currently running process. If it is, the execution of the current is interrupted and the scheduler is invoked to select another process to run. Of course, a process also may be preempted when its time quantum expires.

The quantum duration is critical for the system performance: it should be neither too long nor too short.

If the average quantum is too short, the system overhead caused by process switches becomes excessively high. If the average quantum duration is too long, processes no longer appear to be executed concurrently.

5.0.16 The Scheduling Algorithm used in Linux

The scheduling algorithm used in earlier versions of Linux was quite simple and straight forward: at every process switch the kernel scanned the list of runnable processes, computer their priorities, and selected the "best" process to run. The main drawback of that algorithm is that the time spent in choosing the best process depends on the number of runnable processes; therefore, the algorithm is too costly - that is, it spends too much time in high end systems running thousands of processes. The scheduling algorithm of Linux 2.6 is sophisticated. By design, it scales well with the number of runnable processes, because it selects the process to run in constant time, independently of the number of runnable processes. It also scales well with the number of processors because each CPU has its own set of runnable processes. Further more the new algorithm does a better job of distinguishing interactive processes and batch processes. As a consequence, users of heavily loaded systems feel that

interactive applications area more responsive.

5.0.17 Data Structures Used by the Linux Scheduler

The process list links together all process descriptors, while the runqueue list links together the process descriptors of all runnable processes–that is, of those in a TASKRUNNING state.

In both cases, the inittask process descriptor plays the role of list header. Each process descriptor includes several fields related to scheduling:

need-resched

A flag checked to decide whether to invoke the schedule( ) function policy The scheduling class. The values permitted are:

SCHED-FIFO

A First-In, First-Out real-time process. When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the runqueue list. If no other higher-priority real-time process is runnable, the process will continue to use the CPU as long as it wishes, even if other real-time processes having the same priority are runnable.

SCHED-RR

A Round Robin real-time process. When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the runqueue list. This policy ensures a fair assignment of CPU time to all $\text{SCHED}_R Rreal - timeprocessesthathavethesamepriority.$

SCHED-OTHER

A conventional, time-shared process. A way of voluntarily relinquishing the processor without the need to start an I/O operation or go to sleep. The scheduler puts the process descriptor at the bottom of the runqueue list

rt-priority

The static priority of a real-time process. Conventional processes do not make use of this field.

priority

The base time quantum (or base priority) of the process.

counter

The number of ticks of CPU time left to the process before its quantum expires; when a new epoch begins, this field contains the time-quantum duration of the process. Recall that the update-process-times( ) function decrements the counter field of the current process by 1 at every tick.

When a new process is created, do-fork( ) sets the counter field of both current (the parent) and p (the child) processes in the following way:

```
current -> counter >> = 1;

p->counter = current->counter;
```

In other words, the number of ticks left to the parent is split in two halves, one for the parent and one for the child. This is done to prevent users from getting an unlimited amount of CPU time by using the following method: the parent process creates a child process that runs the same code and then kills itself; by properly adjusting the creation rate, the child process would always get a fresh quantum before the quantum of its parent expires. This programming trick does not work since the kernel does not reward forks. Similarly, a

user cannot hog an unfair share of the processor by starting lots of background processes in a shell or by opening a lot of windows on a graphical desktop. More generally speaking, a process cannot hog resources (unless it has privileges to give itself a real-time policy) by forking multiple descendents.

Notice that the priority and counter fields play different roles for the various kinds of processes. For conventional processes, they are used both to implement time-sharing and to compute the process dynamic priority. For SCHED-RR real-time processes, they are used only to implement time-sharing. Finally, for SCHED-FIFO real-time processes, they are not used at all, because the scheduling algorithm regards the quantum duration as unlimited.

5.0.18 Linux/SMP scheduler data structures

An aligned-data table includes one data structure for each processor, which is used mainly to obtain the descriptors of current processes quickly. Each element is filled by every invocation of the schedule ( ) function and has the following structure:

```
struct schedule-data  {

struct task-struct * curr;

unsigned long last-schedule;

};
```

The curr field points to the descriptor of the process running on the corresponding CPU, while last-schedule specifies when schedule ( ) selected curr as the running process.
Several SMP-related fields are included in the process descriptor. In particular, the avg-slice field keeps track of the average quantum duration of the

process, and the processor field stores the logical identifier of the last CPU that executed it.

The cacheflush-time variable contains a rough estimate of the minimal number of CPU cycles it takes to entirely overwrite the hardware cache content. It is initialized by the smp-tune-scheduling( ) function to:

$$(CachesizeinKB/5000) * cpufrequencyinKHZ \tag{5.1}$$

Intel Pentium processors have a hardware cache of 8 KB, so their cacheflush-time is initialized to a few hundred CPU cycles, that is, a few microseconds. Recent Intel processors have larger hardware caches, and therefore the minimal cache flush time could range from 50 to 100 microseconds.

The schedule ( ) function:

When the schedule( ) function is executed on an SMP system, it carries out the following operations:

1. Performs the initial part of schedule( ) as usual.

2. Stores the logical identifier of the executing processor in the this-cpu local variable; such value is read from the processor field of prev (that is, of the process to be replaced).

3. Initializes the sched-data local variable so that it points to the schedule-data structure of the this-cpu CPU.

4. Invokes goodness( ) repeatedly to select the new process to be executed; this function also examines the processor field of the processes and gives a consistent bonus (PROC-CHANGE-PENALTY, usually 15) to the process that was last executed on the this-cpu CPU.

5. If needed, recomputes process dynamic priorities as usual.

6. `Sets head-data -> curr to next`

7. `Sets next->has_cpu to 1 and next->processor to this_cpu.`

8. Stores the current Time Stamp Counter value in the t local variable.

9. `Stores the last time slice duration of prev in the this-slice local variable; this value is the difference between t and sched_data->last_schedule.`

10. `Sets sched_data->last_schedule to t.`

11. `Sets the avg_slice field of prev to (prev->avg_slice+this_slice)/2; In other words, updates the average.`

12. Performs the context switch.

13. When the kernel returns here, the original previous process has been selected again by the scheduler; the prev local variable now refers to the process that has just been replaced. If prev is still runnable and it is not the idle task of this CPU, invokes the reschedule-idle( ) function on it.

14. Sets the has-cpu field of prev to 0.

5.0.19 Memory Management in Linux

The dynamic part of the memory (RAM) is a valuable resource, needed not only by the processes but also by the kernel itself. Infact, the performance of the entire system depends on how efficiently dynamic memory is managed. Therefore, all current multitasking operating systems try to optimize the use of dynamic memory, assigning it only when it is needed and freeing it as soon as possible.

Page Frame Management:

The size of the page frame plays a major role in page frame management. The Intel Pentium processor can use two different page frame sizes: 4KB and 4MB. Linux adopts the smaller 4KB page frame size as the standard memory allocation unit. This makes things simpler for 2 reasons:

- The Page Fault exceptions issued by the paging circuitry are easily interpreted. Either the page requested exists but the page is not allowed to address it, or the page does not exist. In the second case, the memory allocator must find a free 4 KB page frame and assign it to the process.
- Although both 4 KB and 4 MB are multiples of all disk block sizes, transfers of data between main memory and discs are in most cases more efficient when the smaller size is used.

Page Descriptors:

The kernel must keep track of the current status of each page frame. For instance, it must be able to distinguish the page frames that are used to contain pages that belong to processes from those that contain kernel code or kernel data structures. Similarly it must be able to determine whether a page frame in the dynamic memory is free. A page frame in dynamic memory is free if it does not contain any useful data. It is not free when the page frame contains data of a User mode process, data of a software cache, dynamically allocated kernel data structures, buffered data of a device driver, code of a kernel module, and so on. State information of a page frame is kept in a page descriptor of type page. All page descriptors are stored in the mem-map array. Here are a few fields of the page descriptor.

Flags Array of flags. Also encodes the zone number to which the page frame belongs. It describes the status of the page frame.

Count Page Frame's reference counter. If it is set to -1, the corresponding page frame is free and can be assigned to any process or to the kernel itself. It is set to a value greater than or equal to 0, the page frame is assigned to one or more processes or is used to store some kernel data structures.

Mapcount Number of Page Table entries that refer to the page frame.

Private Available to the kernel component that is using the page. If the page is free, this field is used by the buddy system.

Mapping Used when the page is inserted into the page cache or when it belongs to an anonymous region.

Index Used by several kernel components with different meanings. For instance, it identifies the position of the data stored in the page frame within the page's disk image or within an anonymous region

Lru Contains pointers to the lest recently used doubly linked list of pages.

Memory Zones:

Linux partitions the physical memory of every memory node into three zones.

ZONE-DMA Contains page frames of memory below 16 MB.

ZONE-NORMAL Contains page frames of memory at and above 16 MB and below 896 MB.

ZONE-HIGHMEM Contains page frames of memory at and above 896 MB.

The ZONE-DMA and ZONE-NORMAL zones include the "normal" page frames that can be directly accessed by the kernel through the linear mapping

in the fourth gigabyte of the linear address space. Conversely, the ZONE-HIGHMEM zone includes page frames that cannot be directly accessed by the kernel through the linear mapping in the fourth gigabyte of the address space.

When the kernel invokes a memory allocation function, it must specify the zones that contain the requested page frames. The kernel usually specifies which zone it is willing to use.

Memory allocation requests can be done in two different ways. If enough free memory is available, the request can be satisfied immediately. Otherwise, some memory reclaiming should take place, and the kernel control path that made the request is blocked until additional memory has been freed.

The Zoned Page Frame Allocator:

The kernel subsystem that handles the memory allocation requests for groups of contiguous page frames is called the zoned page frame allocator. The component named zone allocator receives the requests for allocation and deallocation of dynamic memory. In the case of allocation requests, the component searches a memory zone that includes a group of contiguous page frames that can satisfy the request. Inside each zone, page frames are handled by a component named "buddy system".

The Buddy System Algorithm:

The kernel must establish a robust and efficient strategy for allocating groups of contiguous page frames. In doing so, it must deal with a well known memory management problem called external fragmentation: frequent requests and

releases of groups of contiguous page frames of different sizes may leads to a situation in which several small blocks of free page frames are scattered inside blocks of allocated page frames. As a result, it may become impossible to allocate a large block of contiguous page frames, even if there are enough free pages to satisfy the request.

The technique adopted by the Linux kernel to solve the external fragmentation is based on the well-known buddy system algorithm. All free page frames are grouped into 11 lists of blocks that contain groups of 1,2,4,8,16,32,64,128,256,512 and 1024 contiguous page frames respectively. The largest request of 1024 page frames corresponds to a chunk of 4 MB of contiguous RAM. The physical address of the first page frame of a block is a multiple of the group size - for example the initial address of a 16 - page frame block is a multiple of 16 x 2 pow 12.

Here is an example showing the working of the algorithm. Assume there is a request for a group of 256 contiguous page frames. The algorithm checks first to see whether a free block in the 256-page-frame list exits. If there is no such block, the algorithm looks for the next larger block-a free block in the 512-page-frame-list. If such a block exists, the kernel allocates 256 of the 512 page frames to satisfy the request and inserts the remaining 256 page frames into the list of free 256-page-frame blocks. If there is no free 512-page block it looks out for the next higher memory block.

5.0.20 Memory Area Management

The buddy system algorithm adopts the page frame as the basic memory area. This is fine dealing with relatively large memory requests, but how is linux dealing with requests for small memory areas, say a few tens or hundreds of

bytes?

Clearly, it would be quite wasteful to allocate a full page frame to store a few bytes. A better approach instead consists of introducing new data structures that describe how small memory areas are allocated within the same page frame. In doing so, we introduce a new problem called the internal fragmentation. It is caused by a mismatch between the size of the memory request and the size of the memory area allocated to satisfy the request.

A classical solution consists of providing memory areas whose sizes are geometrically distributed; in other words, the size depends on a power of 2 rather than on the size of the data to be stored. In this way, no matter what the memory request is we can ensure that the internal fragmentation is always smaller than 50 percent.

The Slab Allocator:

Running a memory area allocation algorithm on top of the buddy algorithm is not more particularly efficient. A better algorithm is derived from the slab allocator schema that was adopted in the Sun Micro systems Solaris 2.4 operating system.

The concept of a slab allocator views the memory areas as objects consisting of both a set of data structures and a couple of functions or methods called the constructor and destructor.

To avoid initializing objects repeatedly, the slab allocator does not discard the objects that have been allocated and then released but instead saves them

in memory. When a new object is then requested, it can be taken from memory without having to be reinitialized.

The kernel functions tend to request memory areas of the same type repeatedly. For instance, whenever the kernel creates a new process, it allocated memory for some of the fixed size tables such as the process descriptor, the open file object etc.

The slab allocator classifies the requests for memory areas according to their frequency. Requests of particular size that are expected to occur frequently can be handled most efficiently by creating a set of special purpose objects that have the right size, thus avoiding internal fragmentation.

Chapter 6

Reliability, Security and Aging

One of the main goals of this hardware operating system is reliability. Below we discuss some of the more important principles that enhance the reliability of an operating system. We aim at increasing the reliability of the operating system by implementing most of the software part using hardware.

The paper [31] discusses the various bugs that exist in the present operating systems which lead to a reduced reliability. The results of the test is given in the table below, which shows the result of applying different test heuristics, (i.e) the number of bugs reported by them on linux.

At first glance, we can say that the vast majority of bugs are in drivers. This effect is especially dramatic for the Block and Null checkers. While not always as striking, this trend holds across all checkers. Drivers account for over 90 percent of the Block, Free, and Interrupt bugs, and over 70 percent of the Lock, Null, and Variable bugs. Since drivers account for the majority of the code (over 70 percent in this release), they should also have the most bugs.

There are a few possible explanations for these results, two of which we list here. First, drivers in Linux and other systems are developed by a wide range of programmers who tend to be more familiar with the device rather than the

Check	Nbugs	Rule checked
`Block`	206 + 87	To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held.
`Null`	124 + 267	Check potentially NULL pointers returned from routines.
`Var`	33 + 69	Do not allocate large stack variables ($> 1K$) on the fixed-size kernel stack.
`Inull`	69	Do not make inconsistent assumptions about whether a pointer is `NULL`.
`Range`	54	Always check bounds of array indices and loop bounds derived from user data.
`Lock`	26	Release acquired locks; do not double-acquire locks.
`Intr`	27	Restore disabled interrupts.
`Free`	17	Do not use freed memory.
`Float`	10 + 15	Do not use floating point in the kernel.
`Real`	10 + 1	Do not leak memory by updating pointers with potentially `NULL` realloc return values.
`Param`	7	Do not dereference user pointers.
`Size`	3	Allocate enough memory to hold the type for which you are allocating.

Figure 6.1: Operating System Bugs

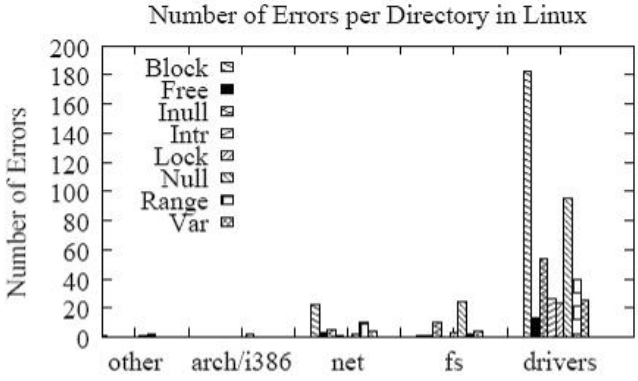

Figure 6.2: The total number of Bugs for each Checker across each main sub-directory in Linux 2.4.1

OS the driver is embedded in. These developers are more likely to make mistakes using OS interfaces they do not fully understand. Second, most drivers are not as heavily tested as the rest of the kernel. Only a few sites may have a given device, whereas all sites run the kernel proper.

These principles also enhance security, since most security flaws are due to attackers exploiting bugs in the code, so greater reliability will also improve security.

Reduce the software size of the kernel Monolithic operating systems (e.g., Windows, Linux, BSD) have millions of lines of kernel code. There is no way so much code can ever be made correct. In contrast, the software part of the kernel of the operating system is made as small as possible by which we believe this code can eventually be made fairly close to bug free. This is done by hardware implementing the device driver functionalities.

Cage the bugs In monolithic operating systems, device drivers reside in the kernel. This means that when a new peripheral is installed, unknown, untrusted code is inserted in the kernel. A single bad line of code in a driver can bring down the system. This design is fundamentally flawed. We aim at implementing most of the functionalities of the device drivers using hardware functional units. When software part of it has to be necessarily executed, they are run as separate user-mode processes. Drivers cannot execute privileged instructions, change the page tables, perform I/O, or write to absolute memory, rather they call the most reliable hardware functional units which implement these functionalities through kernel calls where each call is checked for authority.

Limit drivers' memory access In monolithic operating systems, a driver

can write to any word of memory and thus accidentally trash user programs. Thus a mechanism has to be evolved to prevent these accidental accesses by the driver programs. In operating systems like MINIX, when a user expects data from, for example, the file system, it builds a descriptor telling who has access and at what addresses. It then passes an index to this descriptor to the file system, which may pass it to a driver. The file system or driver then asks the kernel to write via the descriptor, making it impossible for them to write to addresses outside the buffer. This part it is handled by the memory access unit, which is hardwired.

Survive bad pointers Dereferencing a bad pointer within a driver will crash the driver process, but will have no effect on the system as a whole. The reincarnation server will restart the crashed driver automatically. For some drivers (e.g., disk and network) recovery is transparent to user processes. In monolithic systems, dereferencing a bad pointer in a (kernel) driver normally leads to a system crash.

Tame infinite loops If a driver gets into an infinite loop, the scheduler will gradually lower its priority until it becomes the idle process. Eventually the reincarnation server will see that it is not responding to status requests, so it will kill and restart the looping driver. In a monolithic system, a looping driver hangs the system.

Limit damage from buffer overruns We may use fixed-length messages for internal communication, which eliminates certain buffer overruns and buffer management problems. Also, many exploits work by overrunning a buffer to trick the program into returning from a function call using an overwritten stacked return address pointing into the overrun buffer. This attack does not work when instruction and data space are split and only code in (read-only) instruction space can be executed.

Restrict access to kernel functions Device drivers obtain kernel services (such as copying data to users' address spaces) by making kernel calls. We can implement a bit map for each driver specifying which calls it is authorized to make. In monolithic systems every driver can call every kernel function, authorized or not.

Restrict access to I/O ports The kernel also maintains a table telling which I/O ports each driver may access. As a result, a driver can only touch its own I/O ports. In monolithic systems, a buggy driver can access I/O ports belonging to another device.

Restrict communication with OS components Not every driver and server needs to communicate with every other driver and server. Accordingly, a per-process bit map determines which destinations each process may send to.

Reincarnate dead or sick drivers A special functional unit, called the reincarnation server, periodically pings each device driver. If the driver dies or fails to respond correctly to pings, the reincarnation server automatically replaces it by a fresh copy. The detection and replacement of nonfunctioning drivers is automatic, without any user action required.

6.0.21 Security Aspects

A low level security system provides limited discretionary access controls and identification and authentication mechanisms. Discretionary access controls identify who can have access to system data based on the need to know. Mandatory access controls identify who or what process can have access to data based on the requester having formal clearance for the security level of the data. A low-level system is used when the system only needs to be protected against human error and it is unlikely that a malicious user can gain

access to the system.

A higher level security system provides complete mandatory and discretionary access control, thorough security identification of data devices, rigid control of transfer of data and access to devices, and complete auditing of access to the system and data. It is always a trade off between the level of security and performance of the system. If the security level is increased, the performance of the system is reduced by some extent.

To take care of these operating system aspects, the system libraries can be designed for both levels of security and based on the requirement of the User or the application, the corresponding security level can be implemented.

Access Control:

An application-space access control mechanism may be decomposed into an enforcer component and a decider component. When a subject attempts to access an object protected by the mechanism, the enforcer component must invoke the decider component, supplying it with the proper input parameters for the policy decision, and must enforce the returned decision. A common example of the required input parameters is the security attributes of the subject and the object. The decider component may also consult other external sources in order to make the policy decision. For example, it may use an external policy database and system information such as the current time.

Access control is the main security feature of an operating system that needs to be taken care of in an operating system that is designed for an High performance System. In a cluster, when it comes to communication with other

networks, a network level firewall needs to be implemented. This can be done by introducing bastion hosts in the places of input to the cluster. These bastion hosts will be the same normal secondary or primary hosts with the corresponding system library for the security measure being running in it.

6.0.22 Software Aging

Unplanned computer system outages are more likely to be the result of software failures than of hardware failures. Moreover, software often exhibits an increasing failure rate over time, typically because of increasing and unbounded resource consumption, data corruption, and numerical error accumulation. This constitutes a phenomenon called software aging, and may be caused by errors in the application, middleware, or operating system.

Under aging conditions, the state of the software degrades gradually with time, inevitably resulting in undesirable consequences. Some typical causes of this degradation are memory bloating and leaking, unterminated threads, unreleased file-locks, data corruption, storage-space fragmentation, and accumulation of round-off errors.

To counteract software aging, a proactive technique called software rejuvenation has been devised. It involves stopping the running software occasionally, "cleaning" its internal state (e.g., garbage collection, flushing operating system kernel tables, and reinitializing internal data structures) and restarting it. An extreme but well-known example of rejuvenation is a system reboot.

One of the benefits of clustering is its natural redundancy of hardware and software components. A number of single points of failure can be removed by cluster systems. As part of a clustered system, a "failover" process is usually

employed which runs in the secondary host, which transfers workload to another node when a hardware or software failure occurs. An objective of the failover process is Fault tolerance which makes the failures occur gracefully, without an end user knowing that a failure has occurred. In practice, the successful achievement of this objective is highly application-dependent. Also, in order to ensure the ability to perform a failover, sufficient spare resources must be available to accommodate the migrated workload. Generation of these spare resources is very much possible because of the hierarchical host system by making an efficient mapping in the next clock cycle.

Another advantage of a Hierarchical host system is its ability to improve system maintenance. For example, if a specific resource of a cluster is in need of repair and a spare resource is available, it is possible, in a planned manner, to move the load from the resource being repaired, perform a shutdown, and remove and replace the resource, if necessary.

Software rejuvenation technology is a natural fit with Hierarchical host system. Within a set of nodes under a Secondary host, rejuvenation can be performed by invoking the failover mechanisms, either on a periodic basis based on prior experience of the time to resource exhaustion, or extemporaneously, upon prediction of an impending resource exhaustion. Using the node failover mechanisms, one can maintain operation (though possibly at a degraded level) while rejuvenating one node at a time, assuming that a node rejuvenation takes less time to perform and is far less disruptive than recovering from an unplanned node failure. Because the user's application has presumably been written to survive node failovers anyway, this environment has the added advantage of allowing rejuvenation to be transparent to the application.

Simple time-based rejuvenation policies, in which the nodes are rejuvenated at regular intervals, can be implemented easily, using the existing hierarchical host architecture.

Chapter 7

Host Architecture design

The primary and the Secondary hosts play the most important part of the Cluster Operating System. The host architectures are discussed in this chapter.

7.0.23 Secondary Host Architecture

The secondary host is the one that is present in the midway between a set of nodes and the primary host. The main jobs of the Secondary host include Application scheduling at a finer level, Memory Management, I/O Handling, Interrupts and Exception handling, Data packet generation, Resource reporting up the hierarchy, Performance monitoring, Garnering the output of execution. The Secondary host architecture has been illustrated in the figure 7.1 which shows the various units of the secondary host architecture and how they communicate.

The secondary host is an ASIC in which the functional units are fixed based on the requirements of the cluster operating system. The software part of the kernel in the secondary host is kept to be minimal by designing most of the kernel functionalities using hardwired functional units. Each of the secondary hosts has a corresponding high speed memory which can be compared with the hard disks of a normal processor. These secondary memories are multi-

ported to support the feed rate of the execution in the nodes. The controllers of this high speed memory is also hierarchically designed such that they enable parallel transfer of data through the different ports. The memory can be partitioned such that each DRAM of a node has a corresponding counterpart in the secondary memory. These partitions are also zone based, based on the functionality of the memory as done by the linux kernel. The memory can also be partitioned library based (i.e.) based on the subproblems that are allocated to that corresponding secondary host. This type of partitioning is done to associate the dependent libraries and their corresponding data. The secondary memory partitions may be accessed in a multiplexed fashion by the various ports which may not be equal to the number of partitions. The multiplexed accesses should take care of write serialization to maintain consistency of data. The number of partitions, partition size, the number of ports are fixed by simulating the MIP node architecture and find its I/O requirements. There are special utilities to make allocation for partitions like creation, deletion, modification etc. which are resident as system libraries which also get executed as hardware instructions.

As discussed in the cluster architecture, the secondary host has dedicated bus lines to nodes and as well the primary hosts. The data transfer logic between the secondary memory and the DRAM of the nodes is also hardwired such that the secondary memory controllers and the DRAM controllers interact directly in data requests and sends. Software routines are avoided to achieve higher reliability and speed. When a data request is met, the address has to be first converted into the physical address, for which the address decoder is employed. Then this physical address has to be checked across each partition to find the partition in which the data resides. This part is aided with comparator registers which are placed to check the upper and the lower bound of

access for that data packet in a partition. Thus the process of maneuvering through the address space is done completely using hardware. The partition is checked for the availability of access into that partition for the particular packet.

Thus the operating system that is residing in the host is very simple which has most of the instructions corresponding to the hardware functional units which are capable of doing most of the operating system functionalities.

There are numerous DMAs (Direct Memory Accesses) which are allocated to aid in data transfer between the secondary memory and the I/O based on the requests. When the request is from the node for some intermediate storage of data, DRAM miss, storing the output after execution or recorded performance table entries, DMAs are allocated to 'DRAM - Secondary memory' data transfer through the I/O unit. In other cases like arrival of the input application from the primary host, performance reporting above the hierarchy, communication with the primary host is required. In such cases DMAs are allocated to the port that are used to communicate with the primary hosts. This allocation of the DMAs is done by the Memory Mapper Unit of the Secondary Host.

Dynamic and Static Scheduling:

The Secondary hosts are also equipped with facilities for dynamic scheduling of the application. In this process of dynamic scheduling, the host is provided with the capabilities of executing the different scheduling algorithms efficiently. The different scheduling algorithms are discussed in the other chapters and the different hardware functional units of the dynamic scheduler are also designed based on the requirement of the scheduling algorithms that are

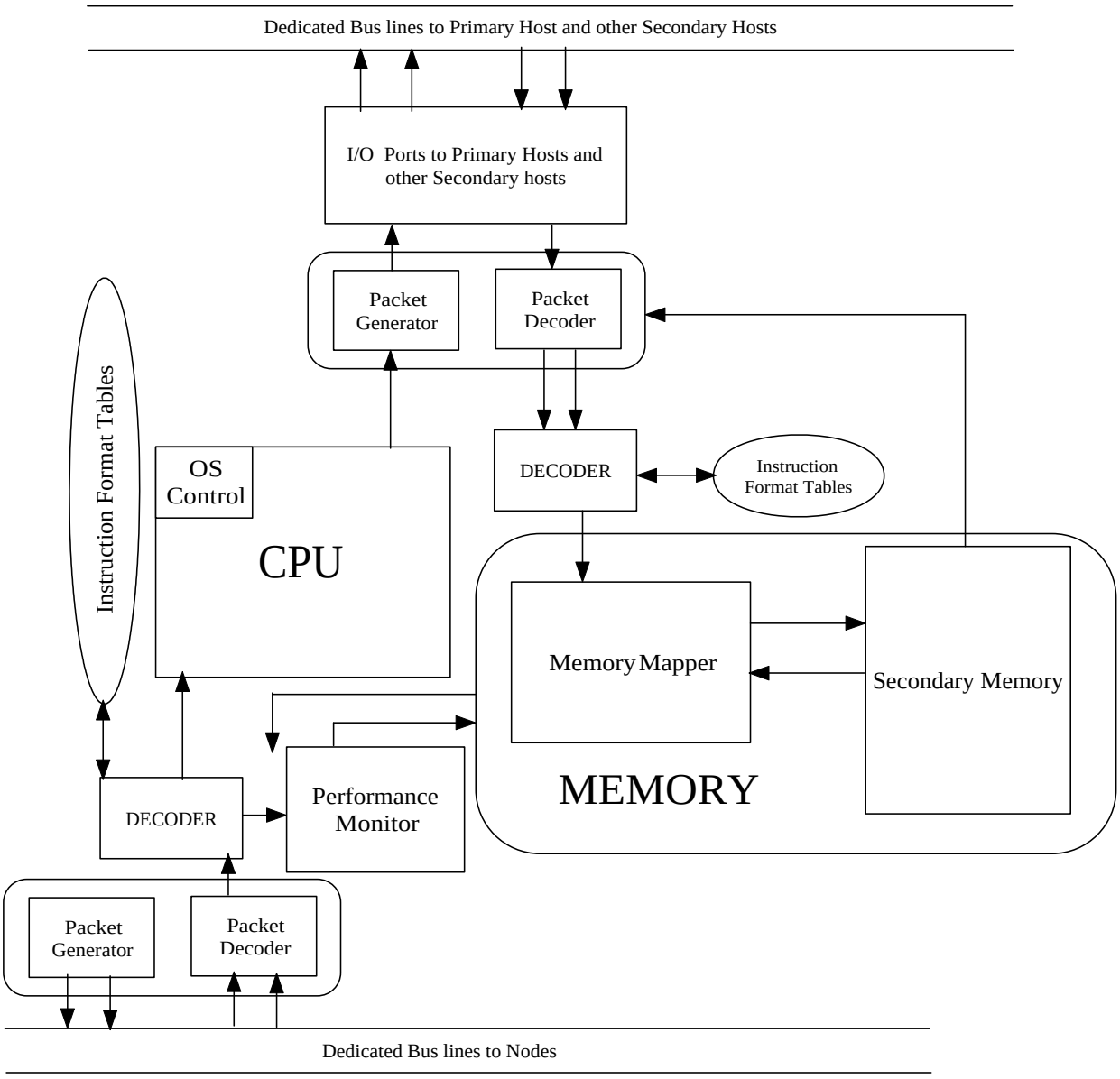

Figure 7.1: Secondary Host organization

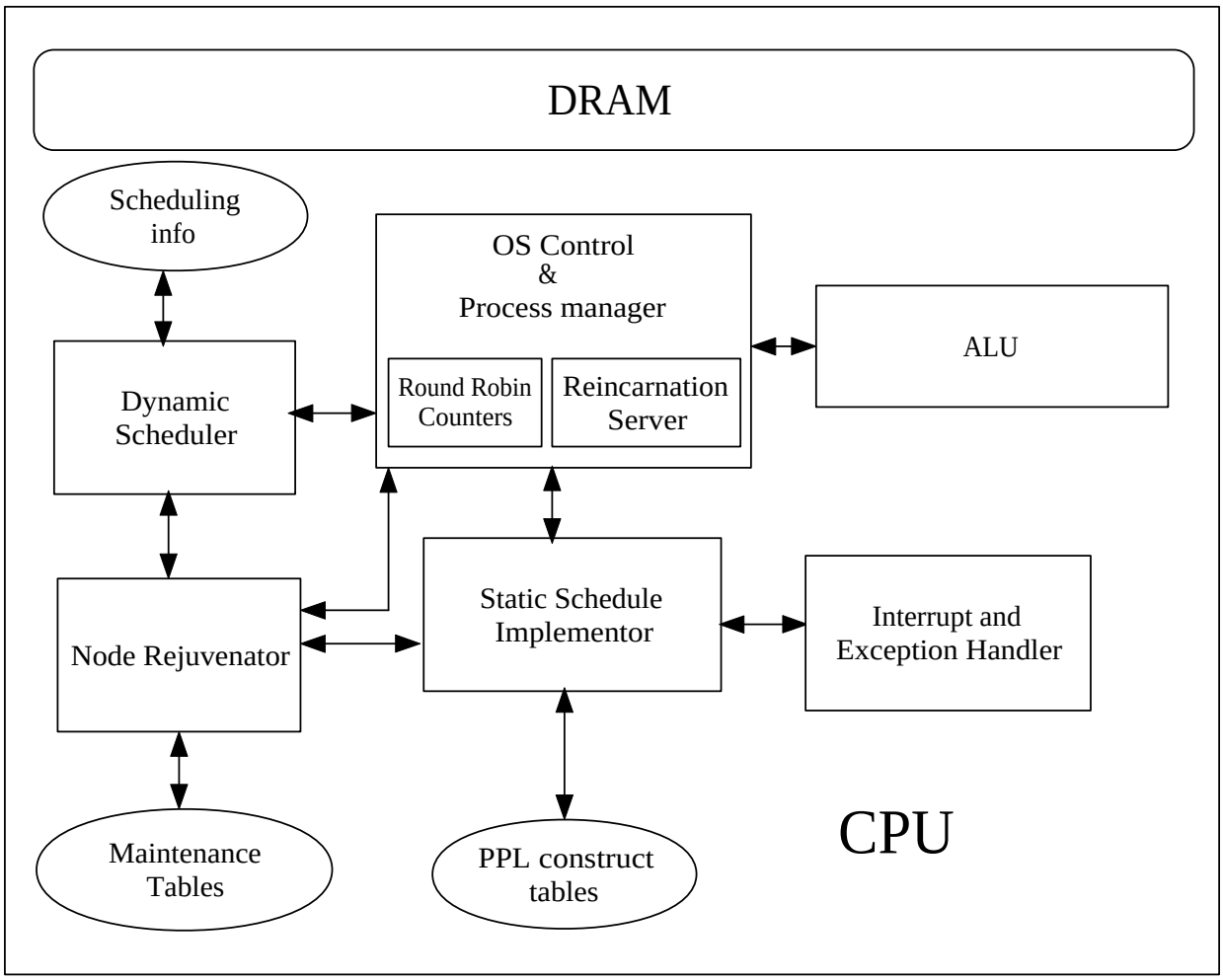

Figure 7.2: Central Processing Unit of the Secondary Host

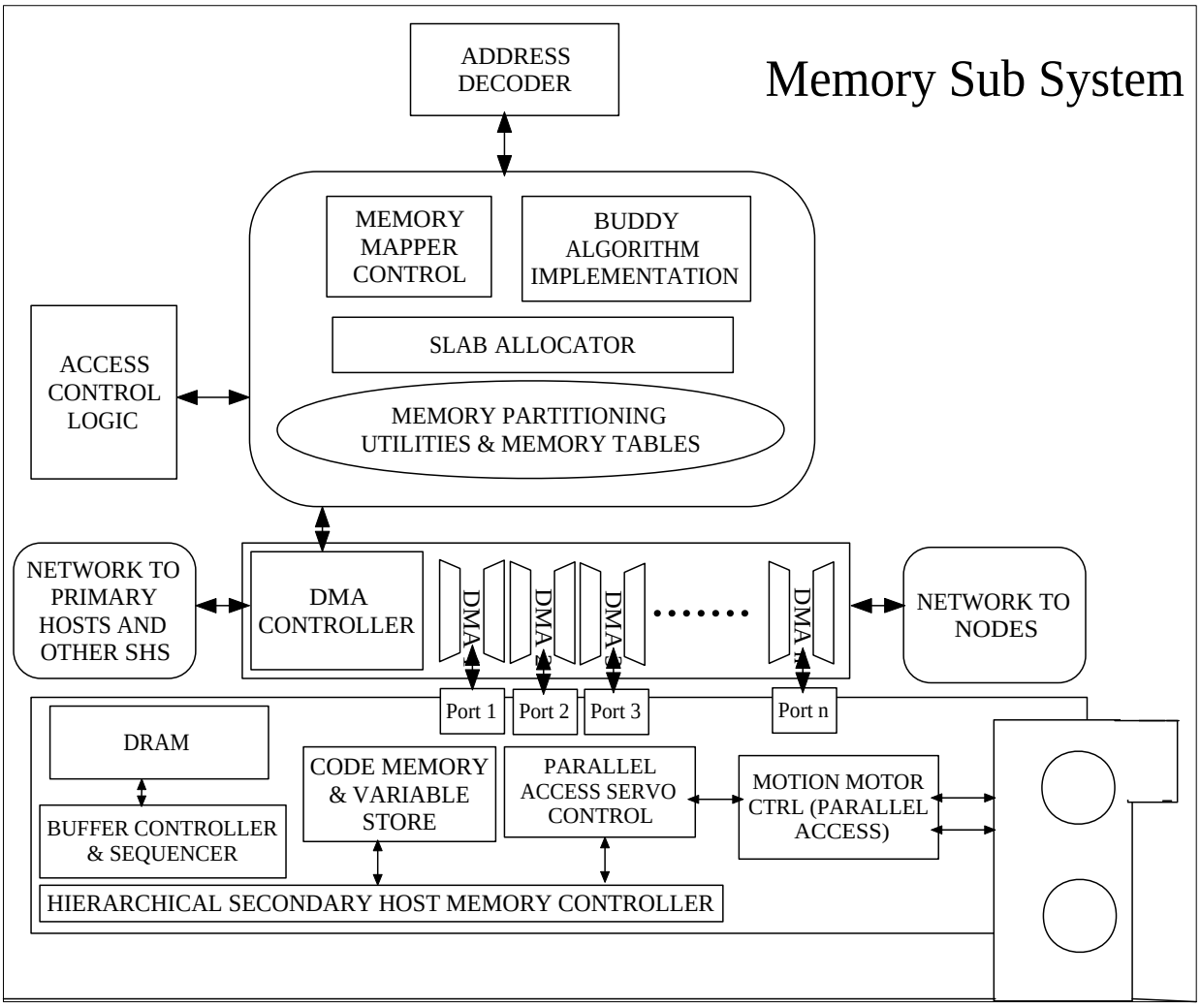

Figure 7.3: Memory Subsystem of the Secondary Host

employed in the Present day Supercomputers.

The Instructions from the Primary Host, the node etc., have to be decoded using a Decoder unit. There may be these Host specific instructions from the Primary host. For example, specifying scheduling related instructions which are needed to be executed in the Secondary host. These instructions have to be decoded by the decoder. There may be requests for data, other instructions that may be arising from the nodes during execution, which needs to be done dynamically. These instructions should also be decoded based on the instruction format. When there are requests rising interrupts and exceptions, they are passed on the Interrupt and the Exception handler unit that is present in the Secondary host.

Packet Generation:

When the secondary host needs to communicate with the other entities, it needs to do the packet formation. In this process of packet formation, the various Headers, Trailers, Error correction codes, information about the encryptions used needs to be added to every packet that is to be transmitted. This is taken care of by the Packet Generation unit, which is given the data and the destination as the input. The data packet should be embedded with the corresponding library ID which can be done only after the process of breaking down the sub-problems into libraries at the Secondary host level. The data packets sent out from the Primary Host bear the sub-problem IDs which are at a coarse level.

The operating system processes are nothing but, subroutines that are made of higher level Instructions which directly trigger the corresponding hardware

functional units. These processes are scheduled by the Operating system process scheduler in the Secondary Host. These processes decide whether the code has to statically scheduled or dynamically scheduled based on the availability of the static schedule. When they are statically scheduled, a schedule based partitioning of the hard disk takes place (i.e.), the hard disk partition for each node is filled with the corresponding application part that has to be sent to that node. To aid in the process of loading the application based on the static schedule onto the different partitions, the static mapper is employed, which is capable of understanding the format of the static schedule given using the parallel programming language and dynamically load them into the corresponding node partitions. In the case of dynamic scheduling, the hard disk is partitioned based the sub-problem. The scheduling algorithm takes care of the mapping the library partitions to the node partitions. To aid in the process of dynamic scheduling higher level functional units like search unit, sorter unit etc are used.

In the case of Operating system processes, the process scheduling algorithms like real time Round Robin algorithm are aided with Round Robin Register Counters, which are directly implemented in hardware so that the Round Robin algorithm gets implemented quickly and reliably. The I/O unit also has multiple ports for the communication to aid in primary host - secondary host, secondary host - secondary host, secondary host - node communication independent of each other by parallel DMA assignments.

Memory Management:

The memory mapper unit takes care of the memory area management and as well the process of memory allocation. The memory areas are modeled as objects which are capable of holding some memory area along with construc-

tor and destructor functionalities which is very similar to the slab allocator present in the Linux. The memory area management algorithm of Linux also runs in the memory mapper. The memory mapper has these slab allocator, memory area management as their functional units along with the different tables which hold the data about status of the memory areas in the secondary memory.

In the memory management, the assignment of the DMAs play a major role in the performance of the memory system. Since the number of ports in the secondary memory and the available DMAs will be lesser than the number of nodes that make concurrent data request, the requests are queued. To make the memory accesses uniform, priorities are assigned to the request based on the frequency of the request, the time of request and the importance of the process that makes that data request. Based on these priorities, the secondary memory controller and the memory mapper satisfy the various data requests.

Performance Monitoring:

The secondary host also takes care of performance monitoring. We divide cluster monitoring into three stages: gathering, consolidation, and transmission. The gathering stage is responsible for loading the data from the operating system, parsing the values, and storing the data in memory. The consolidation stage is responsible for bringing the data from multiple sources together, determining if values have changed, and filtering. The transmission stage is responsible for the compression and transmission of the data.

For gathering data, statistics about the CPU, Disk, Network etc can be collected by using standard system libraries written for each of them. These

libraries take care of gathering these information in the node and then dumping them into the DRAM of the node. These dumped data are then taken up the hierarchy and they reach the secondary memory where the performance data is consolidated. In the process of consolidating the collected data, the data is filtered based on redundancy. The data is then compressed and stored back in the secondary memory. When the performance characteristics need to be displayed to the user, the compressed data is worked on with the help of performance measuring libraries which generate graphs as their output. These graphs are then plotted onto the output device through the primary host.

7.0.24 Primary Host Architecture

The cluster will be connected to the outside world through the primary hosts. Thus, the principal consideration at the primary host level are the security issues. The application partitions of the cluster that are stored in the secondary memory are given access control through a special mechanism.

The related partitions, (i.e.) the partitions of the same application are grouped together and the security aspects are implemented through hardware with the help of comparators. The sequence of libraries in a sub-problem is given a code based on the libraries involved. These comparators are used to compare the existing library sequences code with that of what the user enters and based on the correctness of the comparison the user is given access permissions to the different libraries and the data associated with them. These comparators are also kept locked such that they are enabled only when the user enters the correct application ID. The comparator array is thus maintained such that it can be programmed based on the size of the application partitions. The library sequence codes are fixed based on heuristics such that breaking the code will be a non-polynomial problem.

To have a sustained performance, the load needs to be balanced at the node level plane. To achieve this load balancing at the node level, the primary host has to partition the applications appropriately at the primary host plane, then the same has to continue at the secondary host plane and finally at the node plane. The primary host is the one which is responsible to share the load (applications) in a balanced way between them. This process can be visualized as a graph embedding problem at the primary host plane. The different application partitions and the dependencies between them is the graph which has to be embedded on the graph with the node resources and the cluster interconnect topology. Every library is characterized with the different types of instructions in them and the amount of computations involved. This data is used at the primary host in the process of balancing the load among themselves. The process of load balancing is discussed in the chapter process scheduling where the characterizations of load, the process of balancing them are dealt with in detail. This process involves a lot of addition and comparison operations and thus the Primary host is provided with arrays of MOAs (Multiple Operand Adders) and comparators.

Before the process of load balancing, the application code has to be parsed with the usual lexical, syntax and semantic analysis. To do these compiler related jobs, the Parallel programming constructs need to be stored in the primary host as lookup tables. The code generation or the conversion into the MIP library instructions is done at the secondary host plane. The primary host is also responsible for the application and the sub-application ID generation. This part is done using the simulator in the case of static scheduling. But when the code has to be dynamically scheduled, the primary host takes care of the consistency of the IDs across the different primary hosts.

After the process of ID generation, the sub-applications have to be parceled along with the corresponding data and sent to the different secondary hosts. The process of packet formation is also similar to the one in the secondary host, where the different headers, trailer, ECC need to be appended to the packet before it is transmitted.

The primary hosts are interconnected by a mesh topology as given in the MIP cluster architecture. When the cluster is connected to other networks viz., internet, the mesh network is provided with bridges which are capable of connecting the buses of varied data rates. The current and voltage conversions are taken care of by the bridge in the process of transferring the data to and from the external network. The bridge also takes care of non-standard data packets which arrive from the external network, which need to be converted into the standard of the MIP cluster.

The primary host architecture is similar to that of the secondary host with the above additional features added as functional units into the CPU architecture of the primary host. The CPU of the Primary Host is shown in the figure Primary Host- CPU architecture. The other modules of the secondary host are also present in the primary host. The instruction set in the hosts may be classified as CPU control Instructions, Memory Instructions, ALU instructions, Interrupts and the peripheral access instructions. The memory instructions mainly include the memory partitioning instructions. The other peripheral instructions are the ones that are used to make the I/O from the Primary hosts and the Nodes. The data transfer instructions mainly include the transfer from the secondary memory partition to the DRAM of the nodes in the case of application transfer. The transfer of the output data up the hierarchy

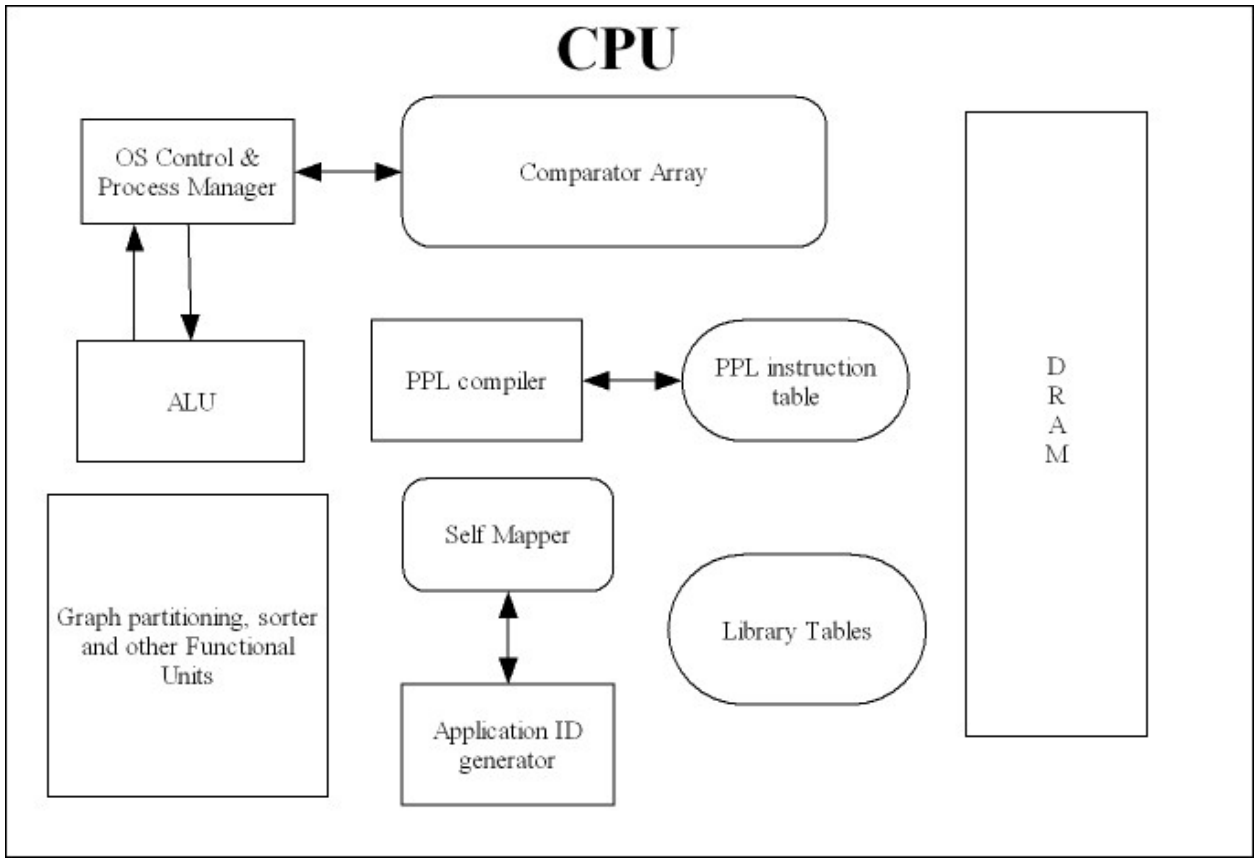

Figure 7.4: Primary Host CPU architecture

is enabled by the transfer between the DRAM of the node and the secondary memory of the secondary host. Then, the data is later grouped based on the application IDs associated with them and the data is transferred back to the primary host. The I/O part of the primary host takes care of presenting the output to the user console.

Chapter 8

Conclusion

This thesis focused on the design of a hardware based Operating System for the MIP SCOC Cluster that is being evolved at WARFT. Most of the features of this operating system being implemented in hardware has the advantages of speed and reliability over the other software based architectures. This is the first attempt towards the design of an ASIC for implementing a cluster operating system which is capable of simultaneously executing multiple applications.

The MIP SCOC node architecture attempts to integrate the memory and the processing units at the bit level which enormously reduces the Von-Neumann bottleneck [32][33]. The Algorithm Level Functional Units and the Algorithm Level Instruction Architecture are the keys for high performance at the node level which is further enhanced by the Primary and the Secondary Compilers on Silicon helping in the process of very efficient scheduling. The reduced memory fetches results in Higher performance and power aware computing.

When this node functions in a cluster which has a hierarchical host system capable of extracting maximum performance out of the node architecture by efficient application mapping and scheduling, the performance at the clus-

ter level is expected to be in petaflops meeting the needs of the current day Grand Challenge applications. Because of the presence of the ALFUs, the MIP Cell architecture (overcoming Von-neumann Bottleneck),Mixing applications of wide characteristics at the node level, the hardware implementation of the Operating System, the silicon compilers play the key role in the process of achieving PetaFLOPs performance.

Bibliography

[1] N Venkateswaran, Arrvindh Shriraman, and S. Niranjan Kumar. Memory in processorsupercomputer on a chip processor design and execution semantics for massive single chip performance. *Fifth Workshop on Massively Parallel Processing (WMPP), IPDPS*, 2005.

[2] N. Venkateswaran, Aditya Krishnan, Niranjan Soundarajan, and Arrvindh shriraman. Memory in processor : A novel design paradigm for supercomputing architectures. *ACM SigArch Computer Architecture News*, June 2004.

[3] www.warftindia.org.

[4] J.K.Aggarwal S-Y.Lee. A mapping strategy for parallel processing. *IEEE Trans. On Comp., Vol.C-36, No.4, pp.433-442*, April 1987.

[5] Karthik G et all N. Venkateswaran. Memory and power efficient application execution in mip scoc. *Dhi Yantra 06, WARFT Workshop on Brain Modeling and Supercomputing,*, March 2006.

[6] N. Venkateswaran and Sivaramakrishnan Nagarajan. Smart communication library for performance scalability of multi million node fusion cluster. *WARFT Thesis.*

[7] N. Venkateswaran and Sudharsan Gunanathan. Application of non-stationary process and population theory towards multi million node cluster mapping. *WARFT Thesis.*

[8] Thomas Sterling. Challenges to petaflops: then and now. *PETAFLOPS II*, February 1999.

[9] Karthik G et all N. Venkateswaran. High performance low power single chip reconfigurable supercomputer for high-end aerospace applications. *8th International MAPLD 05 Conference conducted by NASA,*, September 2005.

[10] Karthik G et all N. Venkateswaran. Parallel mapping of simultaneous multiple applications (smapp) on multi-host hierarchical mip scoc cluster. *Dhi Yantra 06, WARFT Workshop on Brain Modeling and Supercomputing,*, March 2006.

[11] John D.McGregor and Arthur M. Riehl. The future of high performance computers in science and engineering. *Communications of the ACM*, September 1989.

[12] Mario Cannataro Ya. D. Sergeyev Giandomenico Spezzano and Domenico Talia. A dynamic load balancing strategy for massively parallel computers. *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, 1993.

[13] Kumar K. Goswami Murthy Devarakonda and Ravishankar K. Iyer. Prediction-based dynamic load-sharing heuristics. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, June.

[14] Behrooz A. Shirazi and Ali R. Hurson et al. Scheduling and load balancing in parallel and distributed systems. *IEEE Computer society Press*, 1995.

[15] Denis Trystram. Scheduling parallel applications using malleable tasks on clusters. *15th International Parallel and Distributed Processing Symposium (IPDPS '01)*, 2001.

[16] Elie Krevat and Jose G. Castanos et al. Job scheduling for the blue gene/l system. *Euro-Par 2002*, 2002.

[17] N. Venkateswaran, Aditya Krishnan, Niranjan Soundarajan, and Arrvindh shriraman. The mip project : Evolution of a novel supercomputing architecture. *MEDEA Workshop - PACT Conference*, September 2003.

[18] N. Venkateswaran and Niranjan Soundararajan. Hardware compilation for the mip s.c.o.c and hierarchically based multiple host system for the mip cluster. *WARFT Thesis*, June 2004.

[19] E-G.Talbi and T.Muntean. A new approach for the mapping problem: A parallel genetic algorithm. *Laboratoire de Gnie Informatique / Institut IMAG BP 53X F-38041 Grenoble, France.*

[20] T.Muntean E-G.Talbi. Static allocation of communicating processes on a parallel architecture. *Research Rep. RR-833-I-, LGI/IMAG, INPG,*, November 1990.

[21] W-H.Tsai C-C.Shen. A graph matching approach to optimal task assignment in distributed computing systems using a minmax criterion. *IEEE Trans. on Comp.,Vol.C-34, No.3, pp.197-203*, March 1985.

[22] J.R.Agre R.M.Bryant. A queueing network approach to the module allocation problem in distributed systems. *Performance Evaluation Review, Vol.10, No.3, pp.191-204*, 1981.

[23] S.Narita H.Kasahara. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. on Comp., Vol.C-33, No.11, pp.1023-1029*, November 1984.

[24] J.M.Kurtzberg M.Hanan. A review of the placement and quadratic assignment problems. *SIAM Review, Vol.14, No.2, pp.324-342*, April 1972.

[25] J.H.Holland. Adaptation in natural and artificial systems. *Ann Arbor: Univ. of Michigan Press*, 1975.

[26] J.J.Grefenstette. Incorporating problem specific knowledge into genetic algorithms. *Genetic algorithms and Simulated annealing, L.Davis ed., Morgan Kaufmann Publishers, pp.42-60*, 1987.

[27] C.Bogart D.Whitley, T.Starkweather. Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing, Vol.14, No.3, pp.347-361*, August 1990.

[28] G.Robertson. Parallel implementation of genetic algorithms in a classifier system. *Genetic algorithms and Simulated annealing, L.Davis ed., Morgan Kaufmann Publishers, pp.129-140*, 1987.

[29] Mary Hall and Peter Kogge et al. Mapping irregular applications to diva, a pim-based data-intensive architecture. *Proceedings of the ACM/IEEE SC99 Conference (SC99)*, 1999.

[30] Iwan T. Bowman. Conceptual architecture of the linux kernel.

[31] Andy Chou Junfeng Yang Benjamin Chelf Seth Hallem and Dawson Engler. An empirical study of operating systems errors.

[32] Dr. Peter M. Kogge. In pursuit of a petaflop : Overcoming the bandwidth latency wall with pim technology. 1999.

[33] Sunaga T and Peter M. Kogge et al. A processor in memory chip for massively parallel embedded applications. 1996.

[34] G. Henry and B. Cole et al. Performance of the intel tflops supercomputer. *Intel Technology Journal*, 1998.

[35] Timothy G. Mattson and Greg Henry. An overview of the intel tflops supercomputer. *Intel Technology Journal*, 1998.

[36] A. M. Goscinski J. T. Rough. The development of an efficient checkpointing facility exploiting operating systems services of the genesis cluster operating system, future generation computer systems. *v.20 n.4, p.523-538*, May 2004.

[37] J. Silcock A. Goscinski, M. Hobbs. Genesis: an efficient, transparent and easy to use cluster operating system, parallel computing. *v.28 n.4, p.557-606*, April 2002.

[38] Derek L. Eager Edward D. Lazowska and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng*, May 1986.

Appendix A

Supercomputing Applications - An Overview

Some applications with enormous amount of computations bring in the need for building computers with higher performance. A few challenging ones are galaxy simulation, protein analysis, molecular dynamics, computational neuroscience, ocean current modeling, brain modeling, graphics rendering etc. These applications handle a large quantum of data of higher orders there by exploiting the resources in the systems on which they run. Based on the computation and communication strategies, the algorithms of the application have to be mapped on to the cluster. Most of the applications involve matrix, Scalar and graph operations of varying complexities. These real world applications are estimated to take petaflop years to complete their execution.

Galaxy simulation It is based on galaxy collision emphasis. An accurate galaxy simulation is infeasible using current microcomputers. The simulation process was based on a galactic center that is a relatively massive body, representing a super massive black hole. The gravitational force exerted on each star as well as among the various centers of mass is modeled using large sample data on a supercomputer. The method is to

- Accumulate forces by finding the force F(i,j) of particle(any particle body) j on particle i.

- Integrate the equations of motion (which includes the accumulated forces)
- Update the time counter.
- Repeat for the next time step.

Protein analysis It is based on simulating the protein structure formation given an amino acid sequence. Large sample sets are taken into simulation and folding of proteins into a structure is tracked and matched with the existing samples.

Molecular dynamics A computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating their equations of motion. A molecular dynamics simulation allows for example to evaluate the melting temperature of a material, modeled by means of a certain interaction law.

Brain modeling This application aids in investigating genomic, molecular, cellular mechanisms governing chemical and morphological development of the human brain, mechanisms by which genes and environment interact to build the brain, and ways in which the brain becomes affected by pharmacological agents, injury and other pathological processes.

A.0.25 Review of Existing Supercomputers

The top supercomputers [29] [34] [35] are listed in the table below.

1	IBM Bluegene e-server soln.	183500 GFlops	Proprietary	130000
2	IBM Bluegene e-server soln.	114688 GFlops	Proprietary	40960
3	Columbia NASA	60960 GFlops	Numalink	10160
4	NEC Earth Simulator	35860 GFlops	Multi-stage Crossbar	5120

Appendix B

Cluster Operating Systems - An Overview

If a collection of interconnected computers are designed to appear as a unified resource, we say it possesses a single system image (SSI).[36][37] The SSI is supported by a middleware layer that resides between the operating system and user level environment. The different layers in which the SSI resides are,

- Hardware
- Operating system kernel
  - Application and subsystems - middleware
  - Applications
  - Run Time systems (File systems)
  - Resource management and scheduling software

The operating system kernel or gluing layer support gang-scheduling of parallel programs, identify idle resources in the system and allows globalized access to them. It should optimally support process migration to provide dynamic load balancing [38] as well as fast inter-process communication.

B.0.26 Case Studies

Mosix:

MOSIX is a software package that extends the Linux kernel with cluster computing capabilities. The enhanced Linux kernel allows any size cluster of Intel based computers to work together like a single system, very much like a SMP (symmetrical multi processor) system. MOSIX operates silently, and its operations are transparent to user applications. Users run applications sequentially or in parallel just like they would do on a SMP. Users need not know where their processes are running, nor be concerned with what other users are doing at the time. After a new process is created, MOSIX attempts to assign it to the best available node at that time. MOSIX continues to monitor all running processes. In order to maximise overall cluster performance, MOSIX will automatically move processes amongst the cluster nodes when the load is unbalanced. This is all accomplished without changing the Linux interface.

Existing applications do not need any modifications to run on a MOSIX cluster, nor do they need to be linked with any special libraries. In fact, it is not even necessary to specify the cluster nodes on which the application will run. MOSIX does all this automatically and transparently. In this respect, MOSIX acts like a SMP's "fork and forget" paradigm. Many user processes can be created at a home node, and MOSIX will assign the processes to other nodes if necessary. If the user was to run a "ps", he would be shown all his owned processes as if they running on the node he started them on. This is called providing the user with a single server image. It is because MOSIX is implemented in the OS kernel that its operations are completely transparent to userlevel applications.

At the core of MOSIX are its adaptive load-balancing and memory ushering, resource management algorithms. These respond to changes in the usage of cluster resources in order to improve the overall performance of all running processes. The algorithms use pre-emptive process migration to assign and reassign running processes amongst the nodes. This ensures that the cluster takes full advantage of the available resources. The dynamic load balancing algorithm ensures that the load across the cluster is evenly distributed. The memory ushering algorithm prevents excessive hard disk swapping by allocating or migrating processes to nodes that have sufficient memory. The MOSIX algorithms are designed for maximum performance, minimal overhead cost and ease-of-use.

The MOSIX resource management algorithms are decentralised. That is, every node in the cluster is both a master for locally created processes, and a server for remote (migrated) processes. The benefit of this decentralisation is that running processes on the cluster are minimally affected when nodes are added or removed from the cluster. This greatly adds to the scalability and the high availability of the system.

Glunix:

GLUnix was originally designed as a global operating system for the Berkeley Network of Workstations (NOW). The NOW project's goal was to construct a platform that would support both parallel and sequential applications on commodity hardware. In order to fulfill the goal of the NOW project, it was decided that a specialized cluster operating system was required.

After considering the alternatives, the Berkeley researchers decided to build a global runtime environment at user level rather than modify any existing operating system. This is different to MOSIX, for example, where the clustering capabilities are added to the kernel. GLUnix, as actually built, provides remote execution but not entirely transparently. The operating system provides intelligent load balancing but not process migration. GLUnix runs existing binaries. It can handle many node failures but not a failure of the node running the GLUnix master process.

GLUnix provides a cluster wide name space by using global unique network process identifiers (NPIDs) for all GLUnix jobs. Also processes constituting a parallel application are assigned Virtual Node Numbers (VNNs). The process identifiers are 32-bit. GLUnix uses one NPID to identify all the N processes constituting a parallel program. This enables the standard job control signals such as Ctrl-C and Ctrl-Z to function as expected for both sequential and parallel programs. Together NPIDs and VNNs, can unique distinguish any process throughout the cluster. Communication between processes is thus accomplished by using NPIDs and VNNs. GLUnix provides a set of command line tools to manipulate NPIDs and VNNs. These include, glurun, glukill, glumake, glush, glups, glustat. These tool perform similar functions to their standard UNIX counterparts. The glurun utility enables an existing application to be run under GLUnix without modification.