Parallelism Control in Multithreaded Multiprocessors

Namhoon Yoo

CENG 96-35

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4484

December 1996

PARALLELISM CONTROL IN MULTITHREADED MULTIPROCESSORS

by

Namhoon Yoo

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Engineering)

December 1996

Copyright 1997 Namhoon Yoo

Dedication

To my wife Duckhyun and my daughters Susie and Grace. Without your love and understanding, I would never have been able to finish the program.

Acknowledgements

I am indebted to my advisor, professor Jean-Luc Gaudiot, for his inspiration, guidance, and support. It was a privilege to have been one of his students. I would also like to express my gratitude to professors Viktor Prasanna and Doug Ieradie for serving on my dissertation committee. Their inquisitive questions have stimulated my research. I thank professor Sandeep Gupta and Mindeh Huang for being on my Ph.D guidance committee.

I would like to thank my former group members Professors Andrews Sohn, Chinhyun Kim, and Dr. Chih-Ming Lin for giving me advice and encouragement. I thank my colleagues Daekyun Yoon, Hiecheol Kim, Moez ayed and James Burns who have become my good friends over the years. Numerous discussions I had with them have stimulated my research greatly. I also acknowledge group members Yung-Syau Chen, Steve Jenks, Wen-yen Lin, Chulho Shin, Chung-Ta Cheng and Hung-Yu Tseng. Special thanks goes to Mary Zittercob, Rohini Montenegro, and Joanna Wingert for their assistance.

Contents

Dedication ii

Acknowledgements iii

List Of Tables vi

List Of Figures vii

Abstract ix

1 Introduction 1
- 1.1 Motivations and Research Objectives 2
- 1.2 Outline of the Dissertation 3

2 Background Research 6
- 2.1 Data-Flow and Multithreading 6
- 2.2 Compiler Supported Multithreading - TAM 12
- 2.3 Resource Control Issues in Data-Flow and Multithreading 14
- 2.4 Balance Equation in Multiprogramming 16
- 2.5 Performance Modeling of Multithreaded Architectures 19

3 Generic Multithreaded Machine 20
- 3.1 Resource Diagram and Scheduling 23
- 3.2 Processing Element Architecture 27
- 3.3 Instruction Set Description 30
  - 3.3.1 Load/Store Instruction Group 30
  - 3.3.2 Arithmetic/Logic Instruction Group 30
  - 3.3.3 Thread Control Instruction Group 32
  - 3.3.4 Communication Instruction Group 33
  - 3.3.5 Global Structure Access Instruction Group 34
  - 3.3.6 Resource Control Instructions Group 35

4 Resource Control in GMT 36
4.1 Effect of Decentralized Frame Memory Allocator 36
4.2 Effect of Associative Merge 40
4.3 Utilization of Computation Unit for Loops 43
4.4 Complementary Mapping for Loops of Triangular Parallelism . . . 48

5 Balance Equation Theory 52
5.1 Workload model in multithreaded machine system : the average runlength 54
5.2 How to obtain the average latency from a network model 56
5.3 Balance equation set by bottleneck analysis 57
5.4 Balance equation when the computation unit is the bottleneck . . 58
5.5 Projection Method 59
5.6 Comparison with other approaches 61

6 Experimental Result in EM-4 62
6.1 EM-4 Architecture 62
6.2 RL-LAT-K table in EM-4 63
6.3 Projection Method in EM-4 66

7 Conclusions and Future Research 69
7.1 Future Research Issues 70

Appendix A
GMT Simulator Implementation 76
A.1 SMPL 76
A.2 Simulation Interface 78
A.2.1 Batch Command Parameters 78
A.2.2 Interactive Commands 79
A.3 BNF Definition of the GMT assembly language 80
A.4 Sample Programs 81
A.4.1 Function Calls 81
A.4.2 Fibonacci - Recursive Function Call 82
A.4.3 Tak - Recursive Function Call 83
A.4.4 Matrix Multiplication (MMUL) : Array Access and Iteration 85

List Of Tables

2.1 Comparison Method Average Methods 18

4.1 Matrix Multiplication - Execution Time Table 42

5.1 Comparison Method Average Methods 61

6.1 Communication Overheads comparison in EM-4 64

List Of Figures

1.1 Research Overview 5

2.1 Data-Flow Model of Execution 7
2.2 Multithreading 8
2.3 Manchester Machine and Dynamic data-flow machine architecture 9
2.4 Explicit Token Store (ETS) and Decoupled architecture 10
2.5 Classifications of data-flow/multithreaded machine models 11
2.6 TAM Storage Hierarchy 13
2.7 K-Bounding Loop 15
2.8 Working Set Model in Multiprogramming 17

3.1 The GMT Function Calling Mechanism 23
3.2 The GMT Resource Hierarchy 24
3.3 The GMT Scheduling Hierarchy 27
3.4 The GMT System Architecture 28
3.5 The GMT Processing Element 29

4.1 Data-Flow Program Graph for Tak 37
4.2 Parallelism profile for the TAK Benchmark 38
4.3 Speed-up for the TAK Benchmark 39
4.4 Data-Flow Program Graph for Matrix Multiplication 41
4.5 Speed-up Comparison of Parallel and Serial Access for Matrix Multiplication 42
4.6 Parallelism Profile change when Loop Rate considered 45
4.7 Triangular Parallelism I 46
4.8 Bound of Parallelism for Triangular Parallel Loop 47
4.9 Array access pattern in Livermore Loop6 49
4.10 Work Balance Diagram of Complementary Mapping 50
4.11 The Effect of Complementary Mapping (Simulation Results) . . . 51

5.1 Overview to Performance Modeling Method 53
5.2 Example of Loop1 to thread generation 55
5.3 Latency Graph and Latency Model 56
5.4 Overlapping communication and computation threads 58

6.1 EM-4 System Architecture 63
6.2 EM-4 Experiment Set-up 65
6.3 EM-4 Latency vs Run-Length vs Concurrency 66
6.4 EM-4 Normalized Execution Time Projected 67
6.5 EM-4 : The optimum thread number change vs thread run-length 68
6.6 EM-4 Experiment Match Curve 68

Abstract

General purpose parallel computing raises two fundamental issues of *memory latency* and *synchronization delay.* These latency and delay increase inevitably as a system grows larger. Data-Flow architecture addressed these issues by scheduling every instructions asynchronously upon data availability. The data-flow architecture incurs no context switching overhead in accessing remote memory location. Evolved from this extremely fine-grained parallel approach, the *multithreaded* architectures provides a medium grained parallel machine. In the multithreaded machine, the granularity of scheduling is increased to *thread* – a sequence of instructions – from an instruction. This *multithreaded* computer is considered as a viable architecture of *hiding* the latency combined with the conventional *latency reducing* scheme such as cache. However, there are still open questions of how to virtualize the multithreaded computer efficiently and how to determine the resource type of machine to set the proper level of concurrency of threads in a processor. This thesis investigated the parallelism control issue in multithreaded computers.

We propose a virtual machine of Generic Multithreaded Machine (GMT), which is represented by a computation unit, a communication unit, and a memory unit. The GMT instruction set include the conventional instruction set of arithmetic operations and load/store operation from frame memory. It also defines thread control instructions, remote read operations and resource management calls. We have implemented a simulator of the GMT. We performed various test of parallelism control scheme such as decentralized frame memory allocator and

parallel associative merge access. Also we have studied the effect of data and program mapping in case of a serial loop of triangular parallelism. From this simulated study, we observed that the performance of multithreaded programs is determined by the average parallelism of the program and the communication delay ratio to the processor speed.

We established a performance evaluation method in the multithreaded machine. In the first stage, we characterized the programs with the average parallelism and the average runlength between remote accesses. Then we characterized the multithreaded computer by the relationship between the latency and the concurrency level. If the relationship cannot be obtained analytically, we proposed a synthetic loop method which measures the actual time of latency and produces the (RL,K,LAT) relation table. In the final step, we proposed a balance equation of choosing the best concurrency level given the average runlength RL. One of the methods is to use the inverse of the utilization formula from the (RL,K,LAT) table, which we call as *Projection* method.

We have experimented the *projection* method in the EM-4 machine of 80 processors, which is a prototype multithreaded machine built by ETL in Japan. The experimental result shows that the *projection* method can match roughly the actual performance graph, from which we can determine the suitable concurrency level. This method can be used for general multithreaded computers, because we do not assume any specific machine scheme other than the dynamic scheduling of threads. We based our *projection* method upon the measurement of the *(RL,LAT,K)* table from the real machine.

Chapter 1

Introduction

A typical massively parallel system consists of processor-memory pairs connected by some form of message-passing network. In such systems, remote memory requests suffer very high latency costs due to the large distances involved. The latency is even unpredictable due to synchronization delays and contention for limited system resources in the course of memory requests. In addition, the processor speed is an order of magnitude higher than memory or interconnection network speed. Indeed, *communication latency and synchronization* are the two most fundamental issues in the design of parallel architectures and their computation models. One of the most radical approaches is the *data-flow* model.

In the data-flow model, every instruction is scheduled dynamically depending on its data (operands) availability. A data-flow architecture is characterized by a large synchronization name space and an efficient scheduling mechanism. Ideally, data-flow machines should be able to exploit all levels of parallelism available in the program. However, the cost of individually scheduling each instruction proved prohibitive, and *multithreaded computer* emerged by increasing the unit of scheduling from single instruction to thread – a sequence of instruction. In the multithreaded machine, multiple computation threads are interleaved in a processor, overlapping the communication of remote memory accesses with computation of other threads. [13, 25, 4, 19].

1.1 Motivations and Research Objectives

Parallel execution of programs obviously requires more resources than the equivalent sequential execution. Indeed, every concurrent activity needs its own context space and its own synchronization name. Too many concurrent activities may compete for computation resources and network bandwidth. Therefore, the design of parallel programs must delicately balance the exposition of *too little* parallelism, thereby limiting the speedup, with the exposition of *too much* parallelism, thereby making inefficient use of resources or even leading to deadlocks. Unbounded unraveling of loops or recursive function calls would cause *deadlock* or can result in *poor performance* due to resource contention. Limiting concurrency to efficiently exploit the available resources is be the key role of the compiler.

Likewise, in *multithreading*, the compiler or the runtime system should determine the effective parallelism of the program in order not to waste the machine resource such as synchronization names and storage. Several parallelism control strategies such as *loop bounding* [10] or *throttling* [5] have been proposed in the context of data-flow computing. However, those schemes are not theoretically supported by any analytic model which would determine the level of concurrency, as a function of its architectural parameters, communication networks, program profiles.

The objective of this thesis is to find a methodology of parallelism control which can be generally applicable to multithreaded machine. Any parallel machine behavior is hard to be characterized by the nature of the complexity of processing pipeline and network topology. In addition to that, the parallel programs are hard to be analyzed due to the dynamic nature of conditional branching. However, we believe that the fine grained approach like data-flow or multithreading provide us a *parallelism slackness*, which tolerates a small mismatch between parallelism and required resources. Indeed, the performance is controlled by the overall balance

equation of the *average* behavior of programs and machine. If we concentrate on some key parameter of program and the target architecture, we can have a set of rules to find the best concurrency level in the multithreaded machine.

In this regard, we have designed a virtual machine and tested some program to see the effect of various resource manager and mapping strategy. We also figured out the limit of processor utilization by the constraint of program's parallelism. Then we have set up balance equations of system performance under various situation to find the best concurrency level. Finally, we have run a synthetic program under EM-4 machine from ETL in Japan and have shown that from the simple balance equation, we can determine the optimum concurrency level.

1.2 Outline of the Dissertation

The rest of the thesis is organized as follows:

In chapter 2, we review several previous and on-going researches from the data-flow to the multithreaded model and their parallelism control strategy. We also have reviewed a functional language, Sisal, as a target language to determine the program characteristic.

In chapter 3, we propose a virtual machine of generic multithreaded machine (GMT), which are represented by three major units of computation, communication, and memory. The GMT instruction set comprises not only the conventional instruction set of arithmetic operations and load/store operation but also defines thread control instructions, remote read operations and resource management calls.

In chapter 4, we have tested various cases of parallelism control such as the decentralized frame memory allocator and the parallel associative access upon the system performance. We have also studied the effect of data and program mapping in case of a serial loop in the multithreaded machine paradigm. From this

simulated study, we observed that the multithreaded machine can be characterized by the average parallelism of program and the communication delay ratio to the processor speed.

In chapter 5, we established a performance evaluation method in the multithreaded machine. In the first stage, we characterized the programs with the average parallelism and the average runlength between remote accesses. Then we characterized the multithreaded computer by the relationship between the latency and the concurrency level. If the relationship cannot be obtained analytically, we proposed a synthetic loop method which measures the actual time of latency and produces the (RL,K,LAT) relation table. In the final step, we proposed a balance equation of choosing the best concurrency level given the average runlength RL. One of the methods is to use the inverse of the utilization formula from the (RL,K,LAT) table, which we call as *Projection* method.

In chapter 6, we experimented the projection method in EM-4 machine of 80 processors, which is a prototype multithreading machine developed by EDT in Japan. The experimental result shows the the projection method can match roughly the actual performance graph, from which we can determine the proper concurrency level.

In chapter 7, we summarize our works and propose the possible future research directions in this area

Appendix A describes the detailed implementation of the GMT simulator,

The overview of this thesis work are shown in the figure 1.1. The overall frame of research work is based on the Generic MultiThreaded model (GMT). The real world experimentation in done under EM-4 machine with threaded C language support. The balance equation set provide the interconnection between the theoretic model and the EM-4 experimentation.

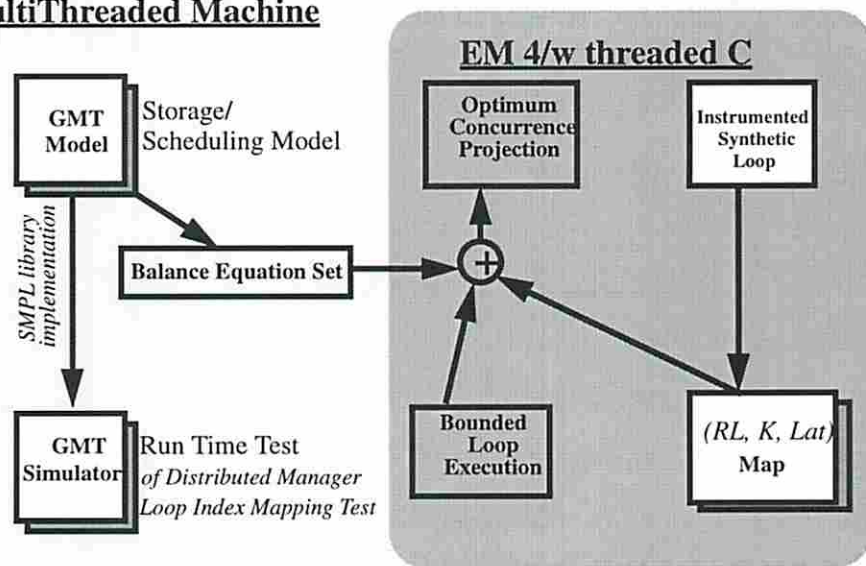

Figure 1.1: Research Overview

Chapter 2

Background Research

In this chapter, we explained how data-flow idea came out and how those ideas evolved into *multithreading.* We classified those data-flow architectures and multithreaded architectures under one category, depending upon those key concepts that differentiated those architectures each other. We introduced the TAM model, which is an important abstract model of multithreading showing the hiearchy of thread scheduling and in the hierarchy of memory model. We briefly reviewed what parallelism control strategy have been proposed in the data-flow machines. We compared the some important balance equation formula which was proposed in the domain of multiprogramming and parallel software design, which motivated our approach in the domain of the multithreaded machines. Finally, we have introduced some analytic approaches to the performanc of multithreaded machines.

2.1 Data-Flow and Multithreading

Memory Latency is an inevitable delay in moving a datum between the processing unit and the memory system. The computation unit can be implemented much faster than the memory system by the pipelining technology. In the implementation of any uniprocessor system, caching and prefetching are the core technology addressing the mismatch between the processor speed and the memory's. In a

multiprocessor system, the memory latency becomes longer due to the increased physical distance between the memory system and the computation unit. The length of the latency even unpredictable due to the synchronization among different computation units. The issue of *latency* and *synchronization* has been at the core of researches in the design of multiprocessor system.

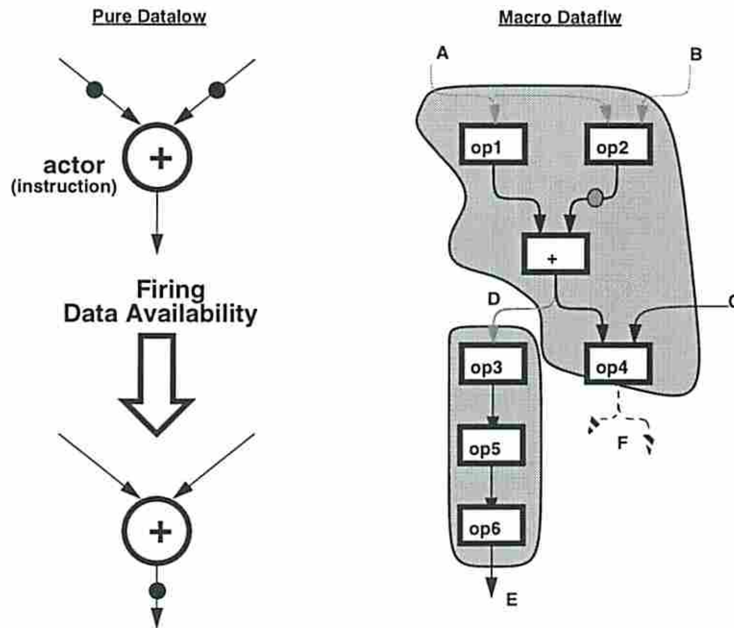

Figure 2.1: Data-Flow Model of Execution

Data-flow programming model schedules an instruction (actor) solely by its operand availability (firing rule). The data-flow machine language is a partially ordered program graph contrary to the totally ordered one of von Neumann computation model. In the machine language graph, the node specifies operation of how to process data carried into the node and the edge defines to which instructions the result data should be carried as shown in the figure 2.1. In a static data-flow scheme, we allow one token in one edge. In a dynamic data-flow scheme, we allow multipl tokens of different colors which should be matched associatively in the scheduling harware. The scheduling is fully asynchronous and incurs no

context switching overheads. Natural extension to the fine-grained approach is to increase the granularity of dynmamic scheduling unit, which is called a macro data-flow. This macro-actor approach evolves into *Multithreading* by introducing the explicit instruction of thread communiction and remote accesses. In *Multithreaded* machine, several logical threads of program control are coordinated in a processor overlapping the communication latency of a thread with the computation of other threads as shown in the figure 2.2 [1]

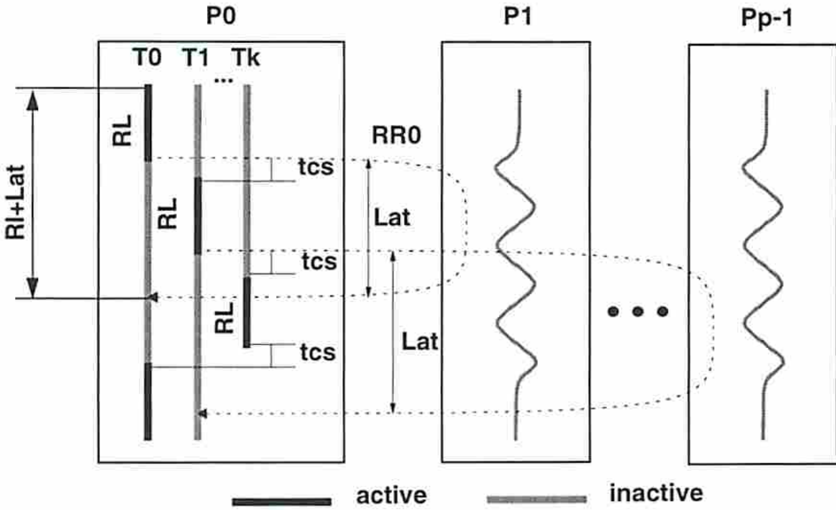

Figure 2.2: Multithreading

We can consider the data-flow an extreme approach to the multithreading in which each instruction is one thread. Indeed, the data-flow machine performs the dynamic scheduling per instruction based upon the availability of data for the instruction. There is no differentiation between scheduling instructions and handling external messages for the processor (i.e., the messages are fed directly into the instruction pipeline with small set of operand data.), which incurs less context switching overhead than the conventional one with whole set of registers.

[1] *Multitasking* is similar to *multithreading* in that both schemes perform efficient context switching between different computations. However, *multitasking* switches between tasks of

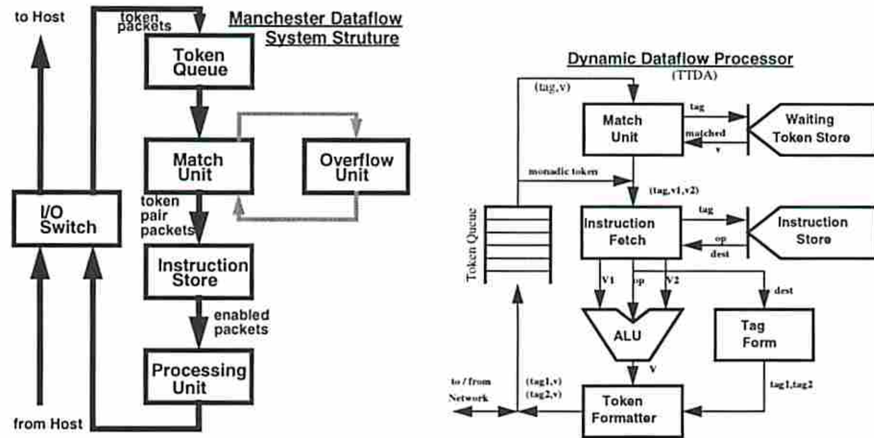

Figure 2.3: Manchester Machine and Dynamic data-flow machine architecture

There have been several implementation to this fine grained data-flow approach such as Manchester Data-flow [21] and MIT's Tagged Token Data-flow Architecture [2] as shown in the figure 2.3. These implementation are of *dynamic* data-flow machine which allows multiple colored token in one edge. These pure data-flow machines have been designed by tightly coupling the synchronization into the pipeline of processing unit with the minimum context of the computation thread. The essential part of these dynamic data-flow machines is the matching unit in which the incoming tokens are waited and matched with its partner tokens. This mechanism requires a special hardware of associative memory, which is costly and limited in size.

In order to obviate the special hardware, Explicit Token Store (ETS) has been proposed replacing such matching hardware by directly addressable linear memory [11], which was implemented in the Monsoon architecture [17]. Similar concept like *Direct matching* has been implemented in the EMC-R in the EM-4 system [30] as a successor to the Sigma-1. The EMC-R and the Monsoon are of *pure data-flow* in that the instruction is scheduled per instruction as incoming token. But the

different programs, whereas *multithreading* switches between threads of the *same* program within the same memory space.

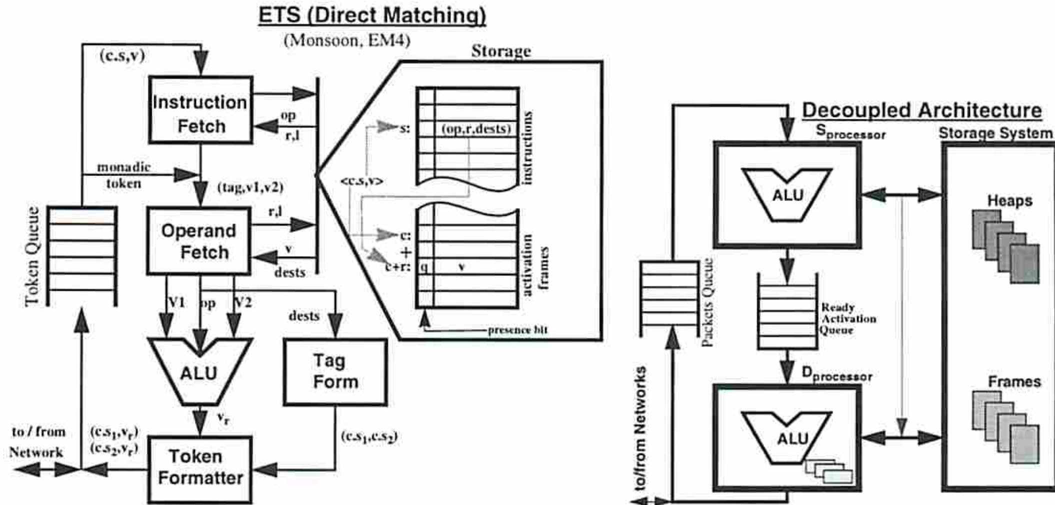

Figure 2.4: Explicit Token Store (ETS) and Decoupled architecture

efficiency of sequential scheduling in the von Neumann computation model is incorporated into the data-flow machine, which resulted in various hybrid machine proposals such as *P-RISC* by Nikhil [28], Hybrid Machine by Iannucci[27], *USC Macro Data-Flow* [23]. In those hybrid models, program counter based sequentiality can be exploited, whenever such program's locality can be found from the program graph.

In general, it was reported that the data-flow architectures execute three times more instructions in a single processor implementation than conventional RISC processor[17]. In order for a data-flow processor to provide a good performance at a single processor implementation, it is needed that the full advanced techniques of existing RISC processor should be exploited while the data-flow scheduling is maintained. In this respect, *decoupled processor* has been proposed[13, 26] as shown in the figure 2.4, where the data processor can be replaced by conventional RISC processor which would relieve lots of engineering work in the design of the new processor. The scheduling processor handles incoming messages asynchronously, which is fundamental to the data-flow idea for efficient mixing communication and computation. The incoming messages usually require short service

of saving the carried data to the memory, of which action should not disrupt the on-going thread of computation. So the scheduling processor is decoupled with the data processor buffered by a continuation queue [2], which is deemed to be a good candidate of multithreaded machine. [26].

In summary, we figured out a classification diagram of various data-flow and multithreaded machine in the figure 2.5 by the following characteristics: the Static/Dynamic, Micro/Macro Data-Flow, the existence of explicit token store or the decoupling characteristics. From the diagram, the underlined are the machines whose prototype machines were built in hardware. In general, the area of lower right corner indicates the recently evolved machines which are being researched actively.

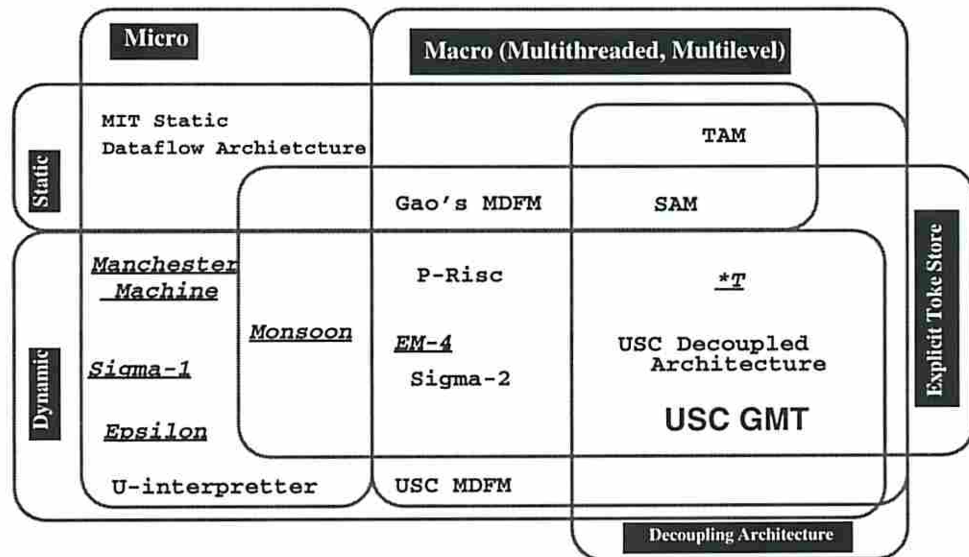

Figure 2.5: Classifications of data-flow/multithreaded machine models

[2]Actually the traditional pure data-flow machine is a degenerated case when this continuation queue has only one slot of waiting thread.

2.2 Compiler Supported Multithreading - TAM

It has been believed that the fine grain parallelism would be exploited only by a special architectural support for the dynamic synchronization. However, the compiler-directed multithreading was proposed as Threaded Abstract Machine (TAM)[7]. The Threaded Abstract Machine (TAM) is an execution model in which the hierarchy of memory is explicit to the compiler so that the dynamic scheduling of data-flow and asynchronous message handling can be controlled by compiler[6]. The hierarchy of any implementation of memory system will limit the number of ready threads which can be scheduled to the processing unit *without causing substantial context switching overheads.* TAM tries to model this hierarchy in its scheduling scheme by allowing the compiler to control the multithreading explicitly.

A TAM program is represented by global structure definition and code blocks: the code blocks are composed of declarations, inlets(message handler which interface the receiving of arguments, return value of a function or heap-access response), and threads. *Activation* is created by invoking a code block together with allocated storage called as activation frame. Within the frames, two level scheduling queues are maintained: one is the linked list of frames in a processor and the other is the*continuation vector* which holds the addresses of threads to be scheduled. The scheduling of activation is done by checking the continuation vector of the frame in which ready threads are accumulated by receiving messages to the frames asynchronously. Once an activation frames is scheduled, threads continues from its continuation vector until no ready thread remains in the *continuation vector.* This unit of scheduling in an activation frame is called as *quantum.* Inlet is a special kind of thread which provide message handling to the activation frame. Heap storage contains objects that are not local to a code-block invocation, including statically and dynamically allocated array. The following

figure shows the tree of activation frames with the two level scheduling queues of inter-frames and intra-frames.

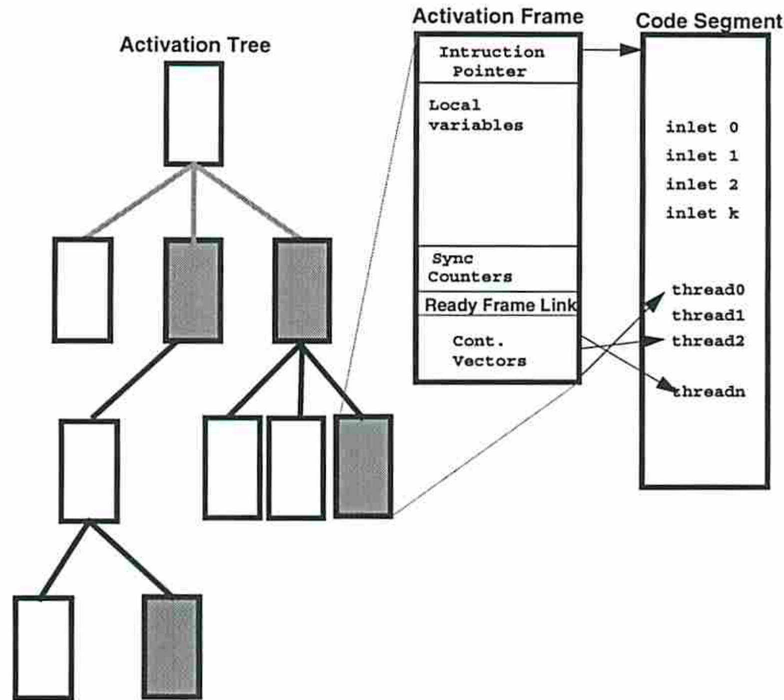

Figure 2.6: TAM Storage Hierarchy

There is no assumption of special architectural support for the dynamic scheduling in TAM. However, *how to handle the asynchronous events of message arrivals* is at the core of compilation issue. In the first implementation of TAM for CM-5, compiler insert codes of polling message queue of CM-5 processor and therefore the asynchronous messages are handled without requiring special context switching hardware[6]. If separate unit of message handling is provided like **T* or *decoupled architecture*, the polling burden of compiler-directed polling ion would be relieved and more efficient in overlapping of communication and computation. TAM does provide only a foundation for static scheduling and resource control by the compiler for multithreading model. How to partition threads and how to allocate the heap structure across processors are still open issues to be refined further. Many programs of dynamic features limit the compiler's role to allocate resources and

reuse them as long as possible. Fast run-time mechanism of resource allocation and reclamation is crucial for dynamic program features.

2.3 Resource Control Issues in Data-Flow and Multithreading

Parallel execution requires more resources than sequential execution. When N iterations of loop are executed in parallel, the resources are needed N times of the resource in executing the same loop in serial. The dynamic data-flow machine, where all levels of parallelism are to be exploited from instruction level to function level, effective control of the parallel activity has been one of the key issues[5, 10, 3] in dynamic data-flow machines.

One way of limiting the parallel activities is by software method, in which some code is added to the original program in a way to limit the parallel activities of activating new set of instructions. In particular, *Function Input Synchronization*, in which a function will be executed only when all input parameters are available as values (not pointer), is a useful. [3] *Loop Serialization* is another one to limit the parallel activities by synchronizing a termination signal of one loop to initiate the next loop, which means there are only one loops active. This *Loop Serialization* can be extended as *K-Bounded Loops* by Culler [10], in which the number of concurrent loop is limited by some compiler-time defined number K. The run-time scheduler activates within the limit of the available system resources but not exceeding the K instances. In the scheme, the termination of loop I is used to trigger loop *(K+I)* recycling the space of loop I as shown in figure 2.7. Beck and Pingali extended the K-loop bounding by further partitioning a loop body into stages [3]. This scheme requires less space than K-bounded loop, but requires more

[3] In a dynamic data-flow machine, this non-strict invocation of function can be implemented naturally without inserting any code.

synchronization between stages and loops. A key issue here is how to determine the parameter K. As there is no definite theory in determining the value K, some experimental tuning methods were used in TAM compilation for CM-5.

Manchester groups reported that these software methods are not enough to effective control of resource in their Manchester Data-Flow Machine so that they proposed a *Hardware Throttle* which structures the token store of Manchester Machine suitable for scheduling in accordance with some simple, predefined rules[5]. Stack structure which favors depth-first execution has been considered and was useful for recursive programs. However, queue-like structure is fair and better for iterative programs. Hence Intelligent Token Queue (ITQ) using both structure has been proposed. Still, the critical part of the design is how to select a new process[4]

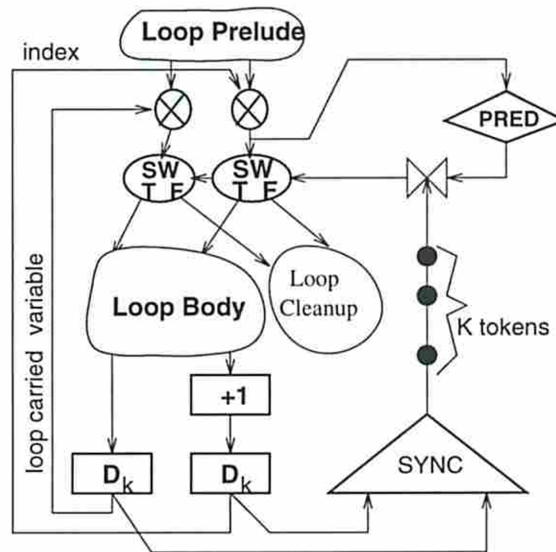

Figure 2.7: K-Bounding Loop

Under multithreaded machine, some data and workload mapping strategy was investigated under EM-4[32], which has shown the effectiveness multithreading.

[4]A process in Manchester machine is Activation Name (token tag).

The following questions are under review for multithreaded machine for resource control.

- How to determine the compile time parameter K?
  In case of *FORALL-LOOP* the K can be determined solely by run-time manager according to the dynamic situation of system load and resource availability at the time of the resource request. However, in case of *REPEAT-UNTIL*, there is loop carried dependency between iterations. Compiler can analyze the loop and define the K assuming an *unloaded* system.

- Do we need a special hardware for resource allocation?
  In the report[17], the portion of resource control instructions are more than 20 requires run-time mapping heavily, an efficient hardware implementation of the resource mapping function would enhance the program execution. In many fine grained parallel execution, the resource control instruction itself compete for the the resource of the machine, which is often overlooked. Then the issue is what is the merit of separate design of manager function?

- Dynamic information of system load is crucial in the throttling the parallel activities. In the Manchester machine, the length of token queue is used for the system load index. However in the multithreaded architectures, there many queue such as message queue and continuation queues. What measure can be used for throttling?

2.4 Balance Equation in Multiprogramming

Multithreading is analogous to multiprogramming in that it is also an attempt at enhancing efficient use of CPU resource by spawning multiple tasks on a CPU. Multiprogramming tries to overlap computation with the *swapping of pages*. Similarly, multithreading overlaps computation with *communication*. If the degree

of multiprogramming (DOM) is inappropriately high, thrashing occurs when the swapping activity prevails over useful computation as shown in figure 2.4 (a).

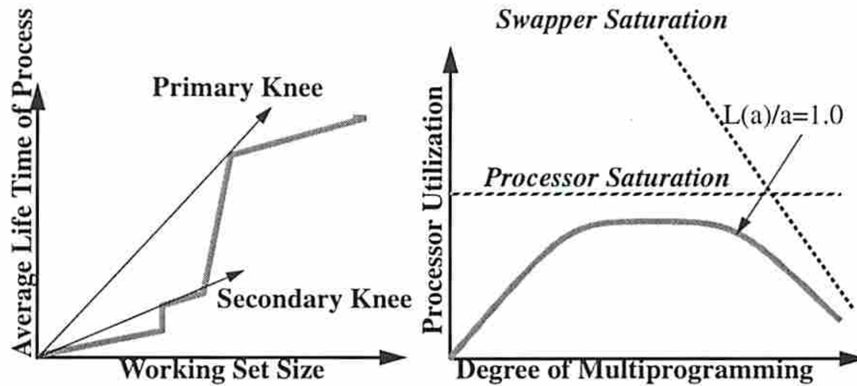

Figure 2.8: Working Set Model in Multiprogramming

Several models have been designed to find the optimum DOM. They are based on analyzing the relationship between working set size and the mean life time of processes [18]. For examples, the *L=S criterion* aims at keeping the average life time of process as high as the page-transfer time for a page fault. From the figure 2.4 (a), this point is where $\frac{L(a)}{a} = 1$. The *knee criterion* seeks to achieve the maximum ratio of the lifetime of a process to the memory size allocated to a process in a multiprogramming set. The goal is to find the working set size such that the ratio of the mean time between faults and the working set size is maximized. This is shown in the figure 2.4 (b) as primary and secondary knee. The *space-time product* method consists in minimizing the space time product, which is the integral of a program's resident size over the time T during which the program is running or waiting for a missing page to be swapped in. It should be noted that the *knee criterion* is the same as the *space-time method* if the page transfer time is dominant. There have been many empirical models to establish the relationship between the working set size and the mean life time such as the *Belady model* and the *Chamberlin model.*

	Working Set Model	Eager	USC Balance Equation
Objective	the optimum degree of multiprogramming	the optimum number of processors	the optimum number of threads
Software Model	mix of processes	parallelism profile	average run-length
System Description	swapping service time	No overheads: Processor sharing	Average communication latency
Balanced Point	L=S	P=SP	kR=L or Bottleneck Analysis

Table 2.1: Comparison Method Average Methods

While the working set model was designed for multiprogramming for uniprocessor, D. L. Eager [12] has proposed an analytic bound for speedup and efficiency in a general software system of work-conserving scheduling [5]. When the average parallelism of a program is known, the *knee* point, where the ratio between the efficiency and execution time is maximized, does exist and can be found by the parallelism profile of the program. The optimum number of processors at the knee is between half and two times the average parallelism. Furthermore, given a program of average parallelism P_{avg} running on P_{avg} processors, we can expect more than 50% of processor utilization.

This model does not consider the overhead of communication and scheduling. It does, however, provide a good starting point of estimating the proper size of parallel machine.

As we have investigated a simple balance equation can be used to set a proper concurrency point depending on the target machine and program characteristic, we are trying to develop a model of setting level of multithreading in the next section. The following table compares the working set model, Eager's model and our model to be proposed in the chapter refch:theory:

[5]Work conserving scheduling refers to one which does not increase nor decrease the total work amount to be executed when different schedule is used.

2.5 Performance Modeling of Multithreaded Architectur

Sakane et.al. investigated the multithreading characteristics on the EM-X multiprocessor system.[16] They provided a micro benchmark program which generates synthetic workload on each processor to investigate multithreading behavior of the system. The multithreading parameters include the number of threads per processor and the average run-length of the threads. Processor efficiencies are measured on machine models on different network assumption to expose the characteristics of the EM-4 system.

Nemawarkar et.al. applied an analytic model to analyze the performance of the EARTH-MANNA multithreaded multiprocessor system[29]. The performance model is based on closed queuing networks. They developed heuristics to account for the realistic subsystem interactions and multithreaded workload. Inputs to the analytic model are architectural parameters of the EARTH-MANNA system, and program work load parameters. Predictions of the performance model include processor utilization and network latency. They validate the analytic model through runtime measurements from the actual program execution on the EARTH-MANNA system.

However, the dynamic data-flow machine allows multiple tokens of different context (color) at the same edge of program graph, as U-interpreter established its logical model[1]. Prototype machines have been built around this model : e.g., Manchester Machine[21], Sigma-1[31] and Epsilon[33] and TTDA[2][6]

[6]This was implemented as an emulated machine among LISP workstations.

Chapter 3

Generic Multithreaded Machine

This chapter defines a Generic MultiThreaded architecture (GMT), which characterize the basic features of multithreaded machine: the split phased operation of remote loads and the efficient thread switching by hardware context. By defining a generic model of multithreading, we tried to test various schemes of compiler's static mapping and run-time control without resorting to specific instructions of specific multithreaded architecture. The introduction of GMT is not intended to define a new architecture or computational model. Instead, we tried to integrate the important features of various multithreaded machine model such as RISC-like scheduling of P-RISC, decoupling of scheduling unit and processing unit like *T, USC decoupled architecture or McGill Architecture, and the storage hierarchy model of TAM. Also the GMT model is inspired and evolved from the macro data-flow model[20]. We added the vector typed packets and augmented a suspended execution of macro actor to the macro data-flow model. Important features of GMT are listed as below.

1. GMT system is composed of a set of nodes (processor and memory) which are communicating each other by sending and receiving messages.

2. The messages are handled in *non-blocking manner* at each destined processor without disrupting the on-going computation thread at the node.

3. The memory model hiearchy is registers, local frame and global heap. The global heap is accessed in split phased way.

4. The node architecture has decoupled units of data processor, communication processor and manager unit(figure 2.4). Decoupling implies a buffering between units, which enables asynchronous operations of each unit. Communications between the processing units are done by exchanging packets, which carries data and thread identification to be triggered by the packet.

5. The size of packet is not fixed. Therefore, the granularity study of synchronization is possible. In order to support the vector synchronization, it is assumed that the frame storage for the vector data are allocated accordingly.

6. The instruction set of GMT processor has three operand type. The addressing modes of operand are immediate, frame or indirect frame. The other special communication instructions such as SEND/RECEIVE is split phased. The manager request instruction of GNF (Get New Frames) or GNA (Get New Array) are also split phased which are handled by a separate manager unit.

7. There are two-level scheduling hierarchy under hardware control. The scheduling within a thread is done sequentially until STOP instruction is encountered. The switching between threads is done by checking the thread queue in the activation frame and retrieving ready thread. The scheduling between activations are done using the hardware queue between the scheduling unit and processing unit.

The run time storage model of GMT is like TAM as shown in the previous section. *Activation frame* is like an instance of a function associated with local frames which are allocated and initialized during run time. Within an activation

frame, there is a thread queue. In the thread queue, there is a number of threads which are basically structures with instruction pointer and some synchronization storages within the frame.

In GMT, the switching between activations is done dynamically by selecting a ready activation from the hardware queue between communicating processor and data processor. But the scheduling are under compiler's control in TAM. The compiler controlled scheduling would be more flexible in using various run-time and static information than the hardware queue. However, we believe that there is a good trade-off point between the hardware controlled scheduling and compiler's static decision.

Each thread is associated with IP (Instruction Pointer to the codeblock) and two values (accumulator and auxiliary accumulator). The auxiliary accumulator is useful for receiving two data values into the waiting thread,which is comparable to a dyadic actor in the pure data-flow model. Communication between activations can be made by referring to the pair of thread number and activation identification which is uniquely defined throughout the GMT run-time system.

A simulator of GMT has been implemented with its assembler [24]. It is coded in a way to experiment various multithreaded model and to test the queuing effect on the various execution units in the processing elements. The communication network has been assumed to be a cross-bar network where pairs of processors are uniformly spaced.

A GMT program graph are be represented by a set of code blocks, where a code block is a sequence of instructions which are similar to RISC instructions. Explicit connection between code blocks is made by sending messages among instantiated codeblocks.

An sample of assembly code for function calling is shown in figure 3.

fcall.,eps

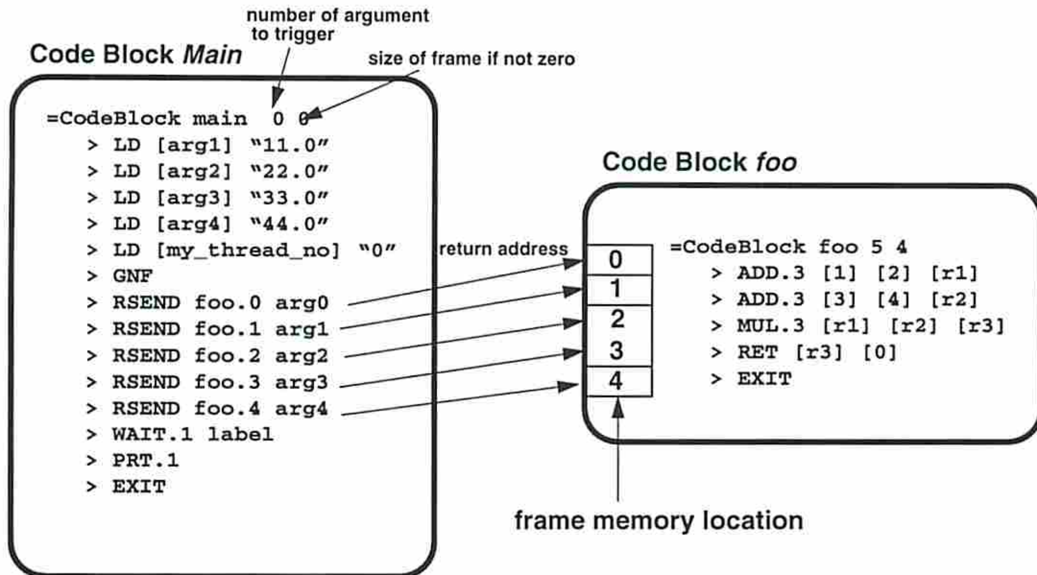

Figure 3.1: The GMT Function Calling Mechanism

3.1 Resource Diagram and Scheduling

Storage resource of GMT is classified into three category which are program memory, frame memory and heap memory 3.1. The program memory keeps the GMT program graph, which are represented by a set of code blocks. Codeblock is a sequence of instructions which are similar to RISC instructions. Explicit connection between code blocks is made by sending messages among instantiated codeblocks. At run-time, each codeblock is instantiated and associated with a frame block which provides the space for those operands of the instruction of the codeblocks. The instantiation of a codeblock with a frame block is called as an *activation*. Within an activation, a set of threads can be resident. The threads can be static with fixed synchronization identification. Alternatively, they can be created dynamically within the allocated local synchronization name. Each thread is associated with IP (Instruction Pointer to the codeblock) and two values (accumulator and auxiliary accumulator). The auxiliary accumulator is useful for receiving two data values into the waiting thread. It is comparable to a dyadic

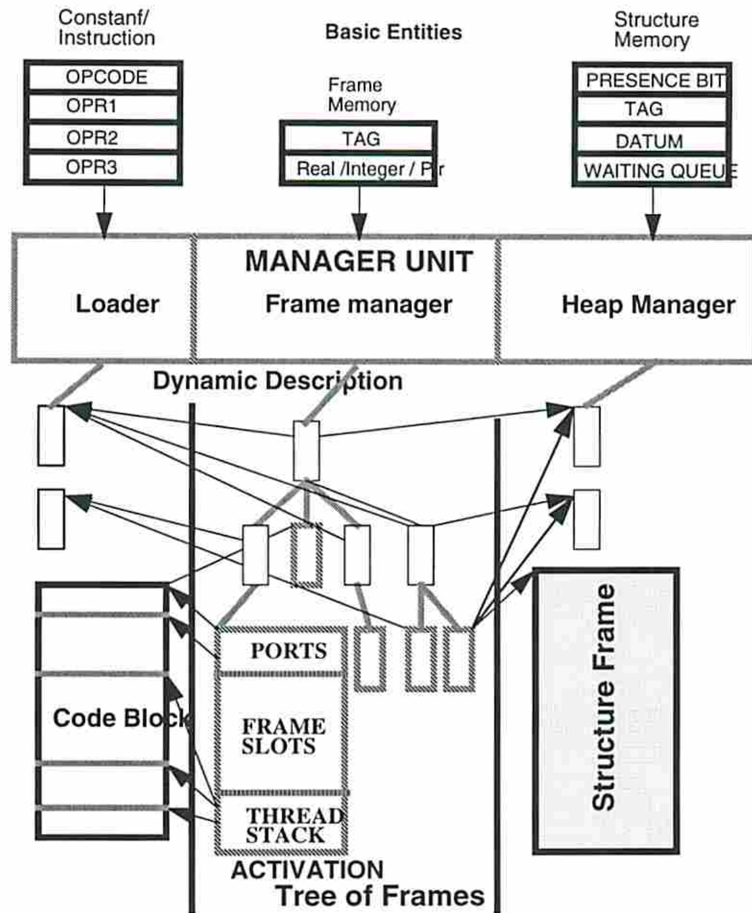

Figure 3.2: The GMT Resource Hierarchy

actor in the pure data-flow model. Each thread is tagged with a unique identification number and communication between activations can be made by referring to this thread and activation identification pair throughout the GMT system. Indeed, the thread name is globally identifiable in the GMT system. Vector type packets can be generated within an activation while an activation can receive the vector type packets. However, in order to support vector synchronization, the frame should be allocated accordingly. This means that the allocation of the frame should provide sufficient storage for the expected incoming vector tokens. Now we summarize the each data type as below:

- **Datum:** The basic data types in GMT are integer, real and array pointer. Further, it should be noted that data elements are tagged by their type.

- **Codeblock:** A code block represents a block of program code which includes some initial firing information to schedule itself as well as a block of micro instructions. A codeblock data structure contains the number of micro instructions, the list of micro instructions, the maximum size of thread stack to be provided, as well as an initial firing count. The initial firing count is the number of arguments to be received before the codeblock can be sent to the data processing unit.

- **Frame Block:** A frame is a collection of Datum which should be associated with an activation. A frame is managed by a manager unit.

- **Activation:** An activation is a unit of scheduling for the data processor. An activation includes a current codeblock pointer, a frame block pointer, and a thread queue to be scheduled when the activation is active in the data processing unit. An activation has a firing counter. When the firing counter is set to zero by receiving data from other activations, the activation is scheduled into the data processing unit. This firing counter should be set initially when the frame is initialized by the information of codeblock. An activation block also contains the caller's identification (color and codeblock number) in order to inform the return address of computed data by RET instruction. An activation is logically uniquely identified by its color value (context) and its codeblock number.

- **Thread:** A thread is a unit of scheduling within an activation. A thread is dynamically created by an activation. It has also an instruction pointer and a frame block pointer. It is associated with two data values, an accumulator

and auxiliary data. The data value is used like an accumulator. A thread can have one of three states:

  - Ready: Ready to run.
  - Waiting: Waiting to receive a signal or data from other threads.
  - Stopped: To be removed from the active thread queue of current activation.

- **Packet:** All communications between activations and global structure access are done by packets. The packet contains a value and activation identification to address. There are several packet types which can be addressed to an activation and to a structure as shown below:

  - FRAME_WRITE : This packet will write the value carried to the frame address and decrease the firing counter of the activation.
  - TRIGGER : This packet will trigger waiting thread with the data carried.
  - ARRAY_READ : Destined to the structure object to read a value.
  - ARRAY_WRITE : Destined to the structure object to write a value.

The scheduling hierarchy is shown in figure 3.1. When a program is in file system, it is not active and can be loaded into memory of the system bye system loader. When a program is active in a memory, we call it as task. The tasks can instantiate an activation by calling parallel function call. Those activation queues scheduling can be programmed by user program. When an activation forks a threads or a waiting threads received a datum from requested remote read, the thread becomes resident in processor memory and dynamically scheduled according to the data processor's availability. The active thread relinquish the data processor by voluntary STOP instruction or remote reads or remote resource manager call.

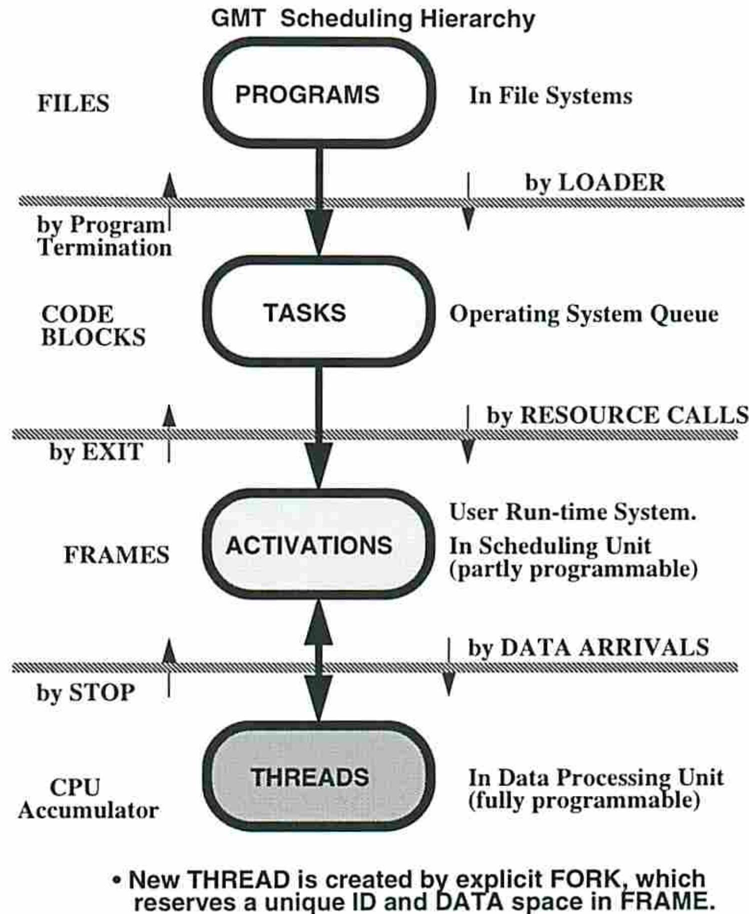

Figure 3.3: The GMT Scheduling Hierarchy

3.2 Processing Element Architecture

Our proposed architecture comprises four decoupled units: the scheduling processors, the data processors, the communication processors and the manager processors, as shown in figure 3.4.

All units are operating in a fully decoupled mode, since there are buffers between the units. Each unit has a local cache for its working space copied from the main frame memory area. However, the local caches can be controlled in a more predictive manner than conventional caches. Indeed, the packets in the queues

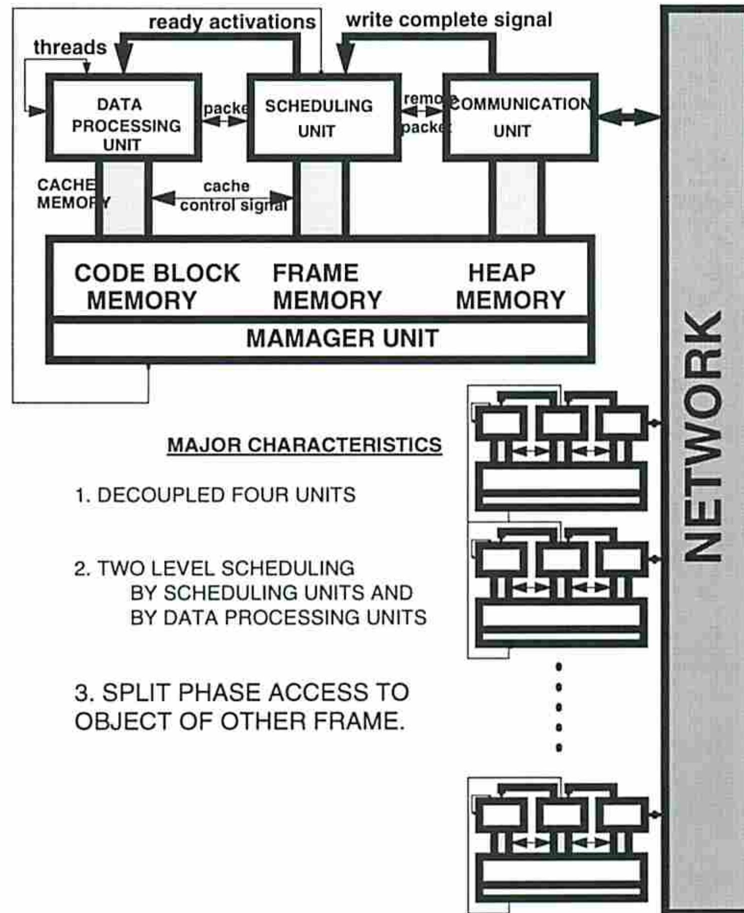

Figure 3.4: The GMT System Architecture

between units contain a pointer to those frame blocks which will be used next. This information can be used to prefetch these frame blocks.

A more detailed figure of the procssing element is shown in the figure 3.2.

- **Data Processors**: This unit performs the arithmetic or logic calculation on the active frame blocks. The ready activation is provided by the scheduler units. While a ready activation is processed in a data processor, many requesting packets could be generated by split-phased instructions. This unit may contain many functional units like floating-point processing and vector pipeline utilizing conventional processor technology.

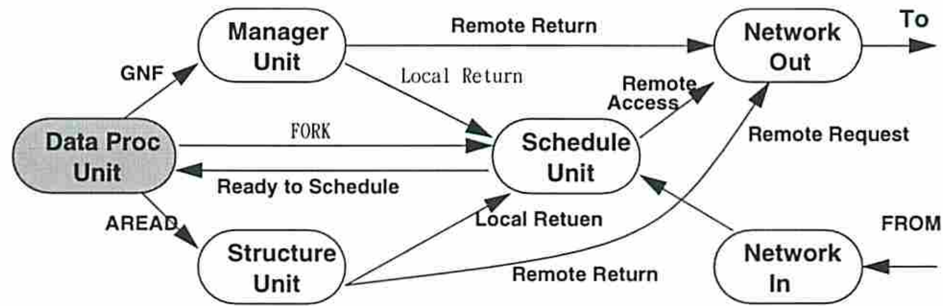

Figure 3.5: The GMT Processing Element

- **Scheduling Unit**: This unit performs the synchronization of the various packets arriving into an activation. This level of synchronization is done by data-flow principles. For all activations, the initial thread is to be scheduled if the pre-specified firing count is set to zero. If the firing counter is set to zero by receiving some data, the activation is scheduled into the queue between data processing unit and scheduling unit.

- **Communication Unit**: This unit performs the construction of packets out of or into a processing element and the appropriate routing. These communication Unit comprise two queue respectively for incoming packes and outgoing packets as shown in the figure 3.2.

- **Struture Unit**: This unit keeps global data such as Heaps and I-Structure and handles those accesses for creat/read/write. The incoming packet this unit can be waited in the unit.

- **Manager Unit**: This manager unit controls the tables of processor resource, such as frame memory, codeblock memory and heap memory. For example, this unit keeps the table of association between physical frame blocks and color identification. It also contains the list free blocks and load status information for load balancing purposes. According to the packet requesting a new color name or array, the manager unit allocates dynamically

the required object. Then, the manager unit sends the reply packet to the requesting activation.

3.3 Instruction Set Description

Instruction is composed by opcode and operands. The addressing mode of operands would be one of Accumulator of the current thread, Frame Direct, Frame Indirect, Immediate. An instruction has variable number of operands, which is specified by *opnum* within the instruction field. The type overloading is done in the simulator, i.e. the addition opcode is the same both for real addition and integer addition.

3.3.1 Load/Store Instruction Group

type checking of real or integer is performed too.

- `LD` : Load immediate value to the current accumulator if *opnum*=1. If *opnum*=2, load the immediate value into the the frame data location operand 1. Only this instruction supports immediate floating point data operand (constant).

- `MOV` : Move the current accumulator value into the designated frame address by operand 1 if *opnum*=1. If *opnum*=2, move (copy) frame data designated by operand 1 to the frame data location operand 2.

- `SWAP` : This instruction is swapping between two data of accumulator and auxiliary accumulator of the current thread.

3.3.2 Arithmetic/Logic Instruction Group

This group is executed solely by the data processing unit. This group is similar to conventional RISC instructions. However, operations are limited to the read/write

of the active frame block of the current activation. Indirect addressing through the frame data is allowed.

Depending upon *no_of_operand*, the execution is performed, as below. And type checking of real or integer is performed too.

- *opnum*=0 The result will doubled accumulator value of the current thread.

```
ADD.0;      v= v+v;
SUB.0;      v= v-v;
```

- *opnum*=1

```
ADD.1 opr1;     v= v+[opr1];
SUB.1 opr1;     v= v-[opr1];
```

- *opnum*=2

```
ADD.2 opr1,opr2;      v= [opr1]+[opr2];
SUB.2 opr1,opr2;      v= [opr1]-[opr2];
```

- *opnum*=3

```
ADD.3 opr1,opr2,opr3;      [opr3]= [opr1]+[opr2];
SUB.3 opr1,opr2,opr3;      [opr3]= [opr1]-[opr2];
```

The list arithmetic opcode is as below.

- `ADD` : Add. Upto 3 operands are supported.
- `SUB` : Subtract. Upto 3 operands are supported.
- `MUL` : Multiplication. Upto 3 operands are supported.
- `DIV` : Divide. Upto 3 operands are supported.
- `INC` : Increment. 1 operand is supported.
- `DEC` : Decrement. 1 operand is supported.

3.3.3 Thread Control Instruction Group

Within an activation, creating a new thread is done by FORK instructions. The input argument is the new IP to start with the same accumulator value of the current thread. The STOP instruction would stop the current thread and a new ready thread would be scheduled (selected from the local queue of the current activation). This group affects thread scheduling within data processor.

The JUMP instruction forces a new IP number to be executed next step. Note that this involves no thread creation: our JUMP is very much alike a conventional JUMP instruction.

The WAIT instruction suspends the current thread into the queue of the current activations.

- `JUMP`: Set instruction pointer of current thread by the label data of operand 1.

- `CBR`: Depending on the status value of accumulator of current thread Set instruction pointer of current thread by the label data of operand 2. The branch condition data is provided by operand 1 mnemonic as below.

  - EQ : equal zero.
  - NE : not equal zero.
  - GT : greater than zero.
  - LT : lesser than zero.
  - GE : greater or equal than zero.
  - LE : lesser or equal than zero.

- `EXIT`: Stop the current activation. The resources of current activation will be reclaimed by resource managed if necessary.

- **FORK:** Create new thread within a activation frame carrying the same data value of parent thread and instruction pointer by operand 1.

  FORK.1 is to generate new thread into the into the thread queue with SAME identification, while FORK.2 is generating new thread with NEW thread identification.

- **STOP:** Stop the current thread.

- **WAIT:** Put the current thread into the thread queue waiting for event from other activation. WAIT.1 is to wait for single data while WAIT.2 is to wait for two data to synchronization.

3.3.4 Communication Instruction Group

Communication within the same context (color) is done by SEND instructions. The operand of an instruction designates the destination codeblock name and the frame displacement to send to as well as the data to be sent. The communication to a codeblock of a different color is done by RSEND, in which the color of destination activation is assumed to be contained in the thread accumulator. This primitive is useful in the function calling mechanism: a block of frame data could be sent by BSEND instruction.

More specifically, the RET instruction is used for sending data back to the caller's activation. It is assumed that the caller's color name is sent into the current activation frame and is available. Further, the packet generated is of the TRIGGER type when there is a waiting thread in the destination activation rather than a FRAME_WRITE.

- **SEND:** This instruction generates a FRAME WRITE packet to the activation of the same color (context) addressed by operand 1 as the codeblock number and its frame address to write and operand 2 as datum to write.

- `RSEND:` This instruction generates a `FRAME WRITE` packet to the activation of the different color (context) named by the thread accumulator value. The rest is the same as SEND.

- `TRIG:` This instruction generate a `TRIGGER` packet to the activation of the parent color (context) addressed by operand 1 as datum to send and and operand 2 as the identification of waiting thread to trigger .

- `ACRE:` Array Create.

  Generate Packet to Structure manager to create an array object and wait for the identifier of created array. Operand 1 is the size of array. There is no differentiation of

3.3.5 Global Structure Access Instruction Group

An array is accessed by AREAD (Array Read) or AWRITE (Array write). This AREAD instruction is split-phased and after the AREAD instruction the current thread is suspended.

- `AREAD:` Array Read.

  Generate Packet to Structure manager to read an array object and wait for the identifier of created array. Operand 1 is the array name and operand 2 is index of array. There is no dimensional information.

- `AWRITE:` Array Write.

  Generate Packet to Structure manager to write an array object and wait for the identifier of created array. Operand 1 is the array name and operand 2 is index of array. Accumulator value of the thread is the datum to write.

3.3.6 Resource Control Instructions Group

There are two basic kinds of resource requesting instructions, Getting New Frame (GNF) and Getting New Array (GNA). These instructions are split-phased, i.e., the requesting packet is made to resource manager unit.

- `GNF.0 :` Get new color instantly. (static request) The new color identification will on the accumulator of the same thread.
- `GNF.1 :` Get new color by decentralized manager. (the same PE)
- `GNF.2 :` Get new color by centralized manager.
- `GNA.0 :` Create new array instantly. The array size is fp[oprd0]. The array identifier will be on the accumulator of the current thread.
- `GNA.2 :` Create new array instantly in the same way of GNA.0 and initialized by the the value fp[opr1].

Chapter 4

Resource Control in GMT

Based on the GMT model and its implementation in the previous chapter 3, we have tested various case of parallelism control such as the effect of decentralized frame memory allocator and the effect of the parallel associative access upon the system performance. Also we have studied the effect of data and program mapping for serial loop in the multithreaded machine hierarchy. From this simulated study, we observed that the multithreaded machine can be characterized by the average parallelism of program and the architectural parameters of communication delay.

4.1 Effect of Decentralized Frame Memory Allocator

We have performed a simulation study to investigate the effect of dynamic frame blocks management. We selected the TAK benchmark as shown in the figure 4.1, since it is frequently used in many Lisp evaluation situations[15] and because it is *function-call-heavy*. Indeed, it is a good test of function call mechanisms of intensive recursions [15].

We have examined three cases of frame memory manager: the static manager, the distributed manager, and the centralized manager. In the static manager

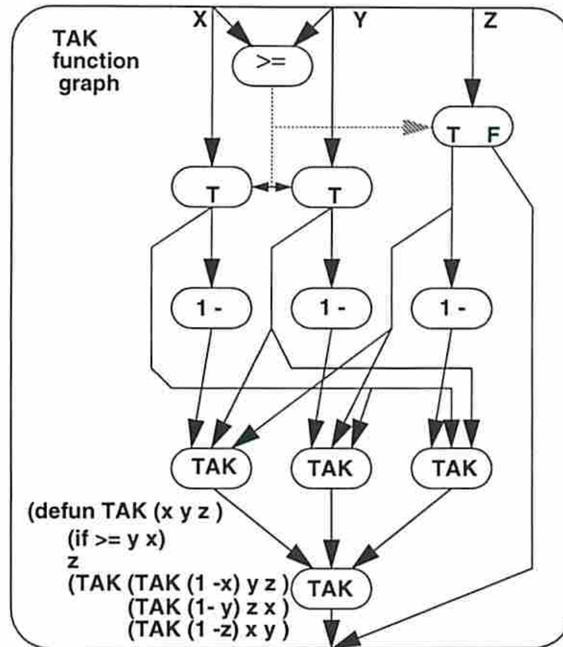

Figure 4.1: Data-Flow Program Graph for Tak

case, all frame names to be required is known prior to being loaded into the GMT system for execution. Therefore, there is no communication delay involved in getting a new frame.

In the dynamic manager case, the caller should get the name of the frame through frame managers. We have assumed two kinds of managers: the *distributed* kind where each local manager allocates frame names in a distributed manner, and the *centralized* kind, where frame names are globally managed by a single manager. In the latter case, creating a new frame involves more communication delays since many requesting packets can be involved.

Figure 4.3 shows that the static case performs best (38 times speed-up using 64 processors). In the static case, those frames are allocated at compile time and there is no resource manager correspondence during run-time.

Second best is the distributed manager case where the local manager can allocate the requested frame without remote packet access. As for the centralized

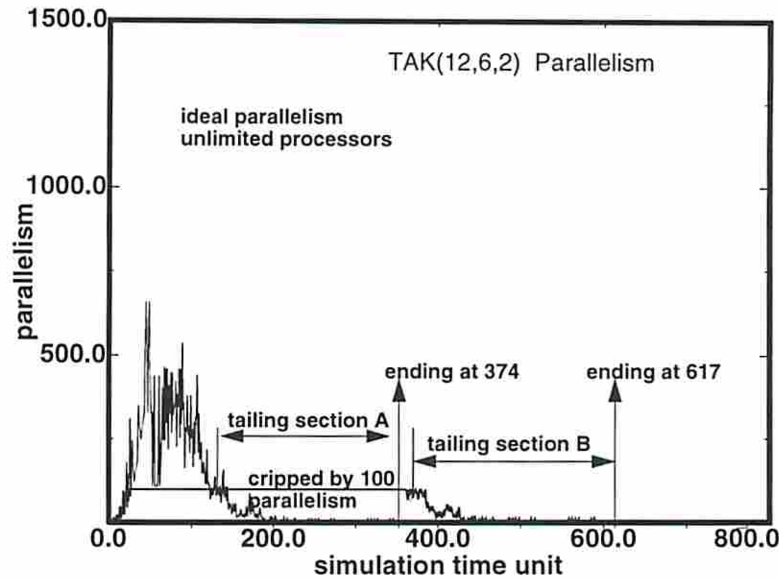

Figure 4.2: Parallelism profile for the TAK Benchmark

case, where one single manager processor does the frame mapping, the speed-up is extremely poor. This simulation result has shown that the compile time binding of frame reduces the execution time (about 30 % less than distributed manager case for 64 processors). In the case of the dynamic mapping, a distributed manager would be much preferable to a centralized one.

The ideal speed-up cannot be reached since the program has some inherent sequential sections as shown in the figure 4.2. One of the histograms shows full parallelism without any limitation on the resources while the clipped case displays the histogram of parallelism if only 100 execution units were available without any communication delay. In both cases, there are tailing sections of smaller parallelism and of similar shape These segments show that there are some serial sections which cannot be parallelized.

In order to reach ideal speed-up, we could have run larger programs so as to saturate communication resources and Processing Elements. In this simulation, however, we have concentrated on the effect of manager access and the static allocation effect.

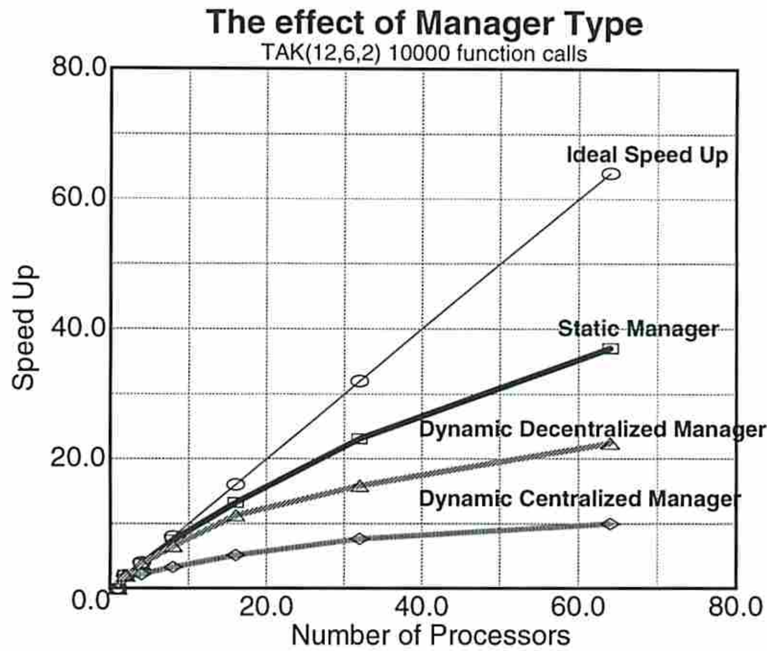

Figure 4.3: Speed-up for the TAK Benchmark

Now we discuss the effect of computational load of the frame manager itself. It has been reported that up to 50% of instructions are from manager calls of run time system for some applications in Monsoon system[17]. The manager calls are performed in a exclusive mode with which other computation threads cannot be mixed with. This manager call can disrupt severely the multithreaded pipeline like Monsoon. The dynamic manager is inevitable in a parallel machine because of the dynamic natures of programs and the synchronization requirement to be run in parallel. In reality the manager function itself compete with the original computation for a single CPU[8]. Then the serialization to the dynamic manager service would be a bottleneck in getting a good performance of multithreaded machine. There may be several solutions to this:

- Compile the program in a way to reduce the dynamic manager call. Determine the static allocation. But this is not always possible.
- Reduce the critical path of dynamic manager's service.

- Run multiple resource managers in a processor. (Multiple Service Queue) This may result low utilization if coordination between load migration between different manager is not allowed.

- Design a decoupled manager unit in hardware.

The algorithm of quick fit or buddy system would be implemented (programmed) in the separate decoupled manager unit. This separate implementation will speed up the dynamic manager's service without disrupting the other computation threads.

4.2 Effect of Associative Merge

For the iterative pattern test, a simple matrix multiplication has been used. The program structure is shown in figure 4.4. The range generator codeblock generates the index range for row vector of array A and column vector of array B to be vector-produced. The generated ranges are sent to the vector product calculation codeblock of the corresponding row and column. Now the following calculation is done in the vector-product macro-actor.

$$\texttt{VectorSum} = X_1Y_1 + X_2Y_2 + \ldots + X_NY_N$$

In the vector product actor, a simple scheme would be to send the request to the vector starting from index 1 to N serially. However, the vector summation is an associative operation. This means that the order of summation is unimportant. Thus, we can send array access requests for all indices at once and then the summation will be be done in the order of arrival of the requested data. The simulation has shown that the effect associative summation does not perform better that the serial case as in the following table 4.2). When the communication unit is enhanced (4 times and 10 times of instruction execution time per packet

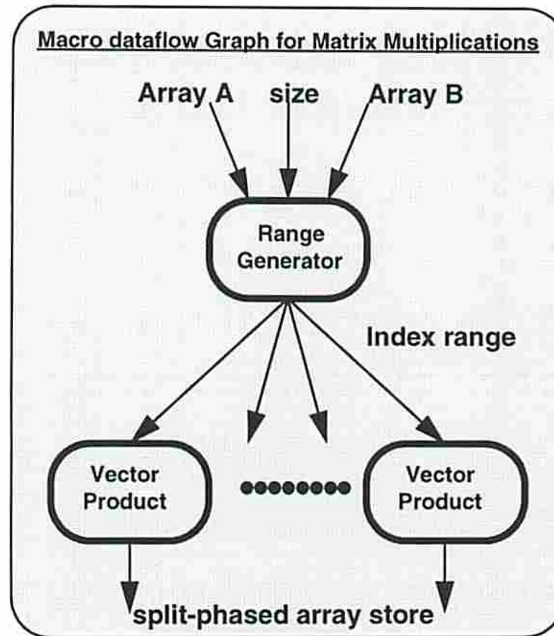

Figure 4.4: Data-Flow Program Graph for Matrix Multiplication

handling), then the associative merge cases are a little better than the serial array access as shown in the figure 4.5.

This poor result in the parallel vector product is due to the burst of packets from the vector product activation, which saturates the communication unit and the array access unit. Another reason is the overhead control instruction of parallel access to be synchronized within an activation. As shown in the single processor case, the total instruction count is 40 % more (14563) than in the serialized case (98563). This instruction overhead could not be offset by exploiting the parallelism in our simulation. A third reason is that the program is not big enough to utilize all 64 Processing Elements and communication nodes. Currently, the size of program to be run in software simulation is severely limited by the computing power of simulator. In order to run more big programs and resource access patterns, multiple trace-driven simulation is being studied.

It should be noted that the choice of serial array access rather than parallel array access will be another way of throttling the activities of parallelism in order

Processors	com=1.0 par	com=1.0 ser	com=4.0 par	com=4.0 ser	com=10.0 par	com=10.0 ser
1	140563	98563	140563	98563	140563	98563
2	73163	52163	73163	55396	139020	135110
4	39463	28963	39608	37940	96360	94350
8	22613	17363	27880	26276	65280	65480
16	14188	11563	20392	19892	47730	49670
32	9807	8547	15728	15936	37700	39730
64	7785	7155	13532	13776	32990	34400

Table 4.1: Matrix Multiplication - Execution Time Table

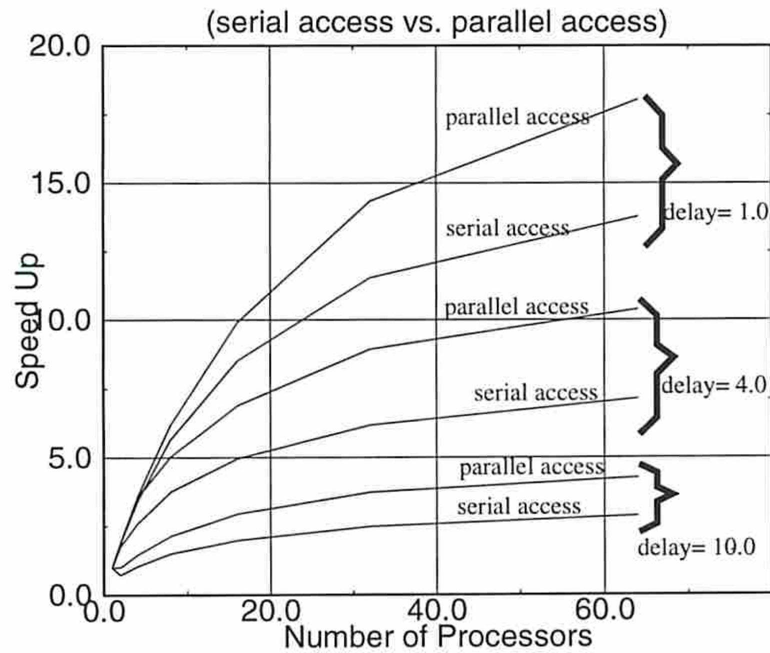

Figure 4.5: Speed-up Comparison of Parallel and Serial Access for Matrix Multiplication

to reduce the resource requirements not affecting the execution time. Associative array access in a parallel vector product is using N-time more synchronization name space than a serialized vector product loop, since there are N pending array access requests to the activation.

4.3 Utilization of Computation Unit for Loops

The utilization of computation unit is one of the important performance indices for the effective computation unit. The low utilization may be caused by lack of parallelism of application program or the imbalance of the program mapping between computation units and communication units. In this section, we try to investigate the fundamental limit of the utilization due to the program characteristics.

There are many levels of parallelism in a program: instruction level parallelism, block level parallelism or function level parallelism. The instruction level parallelism are mostly exploited by the instruction pipeline using the techniques of prefetching or multiple issuance of instruction in a cycle (superscaler machine). However, it has been reported[34] there is some limit to the available parallelism for a single threaded programming even if perfect condition of such techniques are assumed. Thus the parallelism of functional or block level is important in a large scale parallel machine. In a multithreaded machine, the concurrent threads of communication and computation are the key to the exploitation of parallelism of program. Hence the limit of the available parallelism in a multithreaded machine must be considered in the context of overlapping communication threads (array accesses) and computation threads rather than by the instruction level parallelism. The available parallelism is calculated by the critical path analysis of the program and the total number of instructions. The critical path in a loop execution should

be calculated by *the path of longest array accesses* not by the longest path of general instruction sequences. Usually the array access is split-phased which takes non-deterministic time depending upon the system load. However, we may assume an average time of array accesses and the maximum number of sequential array accesses in the critical path of the loop body will determine the loop execution time as below.

$$\text{LoopExecuctionTime} = \text{MAX}\{\text{critical path of maximum array accesses * average access time}\}$$

The logical parallelism of *FORALL* loop is not limited by the loop execution time, as loops can be instantiated as many as possible since there is no data dependency between the loops. However, the serial loop with some data dependency between loops, there is a limit to the available parallelism. In SISAL language, the dependency is explicitly represented as *old* key word in the loop construct. This serialized portion can be a bottleneck limiting the available parallelism of a program.

In the Livermore Loop kernels in SISAL [14], the nested loop construct of *FORALL* and *WHILE* is frequent as in Loop2, Loop5, Loop6, Loop13, Loop17 , Loop20, Loop19, Loop23. Usually the *FORALL* loop in the inner loop has decreasing size of iteration as the iteration number of the outer loop increases. Usually, the nested loop represents a decreasing parallelism profile. therefore, it would be useful to find out a bound of processor utilization depending on the type of decreasing parallelism profile.

FORALL loop construct represents a rectangular parallelism profile. However, in a real machine, the slope of increase of parallelism must be limited by the rate of creating loop instances. Let's assume the issue rate $\alpha = \frac{M'}{t}$. Then the rectangle should be skewed by this rate as shown in figure 4.6. This profile can be regarded as a set of one rectangular parallelism of smaller maximum parallelism M' and two

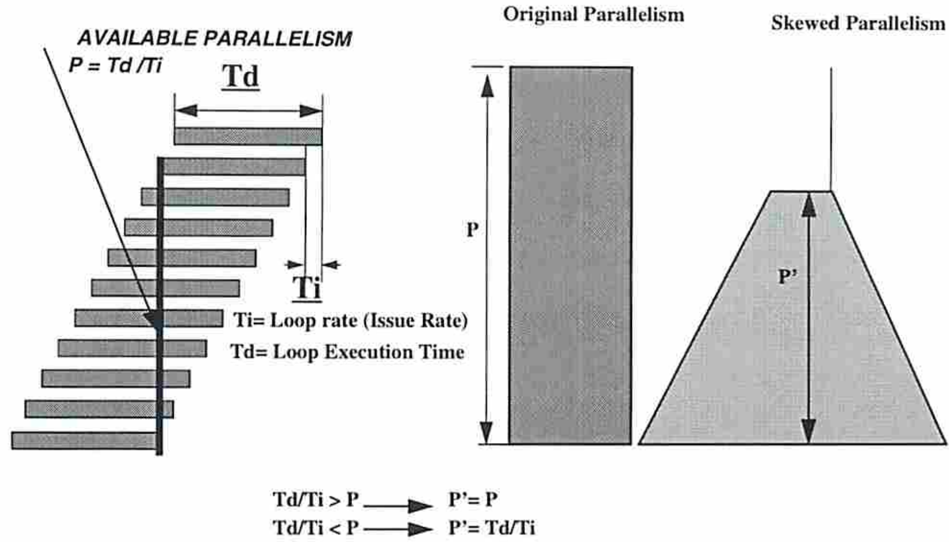

Figure 4.6: Parallelism Profile change when Loop Rate considered

triangular parallelism, where the modified maximum parallelism M' is calculated from the above figure and the following equations:

$$M' = \frac{T}{T_i} \tag{4.1}$$

$$\frac{M'}{t} = \frac{1}{T_i} \tag{4.2}$$

$$t = T \tag{4.3}$$

$$M'(T' - t) = MT \tag{4.4}$$

$$T' = T_i M + T \tag{4.5}$$

$$T' = T(1 + \alpha M) \quad where \quad \alpha = \frac{T_i}{T} \tag{4.6}$$

In the above equation α is the ratio between loop length and the preceding overhead time and the M is comparable to the number of iterations.

The triangular parallelism is frequent in the nested loops. There may be two kinds of triangular parallelism, Type I and Type II as in the following figure 4.7.

By careful investigation, the Triangular type I is a special case of Type II when $T_1 = 0$ and $T_2 = t_2 = t$.

So we are going to calculate the utilization of processors for the case of Type II assuming that the number of processors are limited and the work loads are perfectly balanced. When the number of processors is limited, the restricted parallelism profile is shown in figure 4.7. In order to find the processor utilization, there are two conditions to consider: one is that the total number of operations are the same for the both shape and the tailing (or preceding) triangles of the limited parallelism are of the same as the original profile. The we can obtained the following equations.

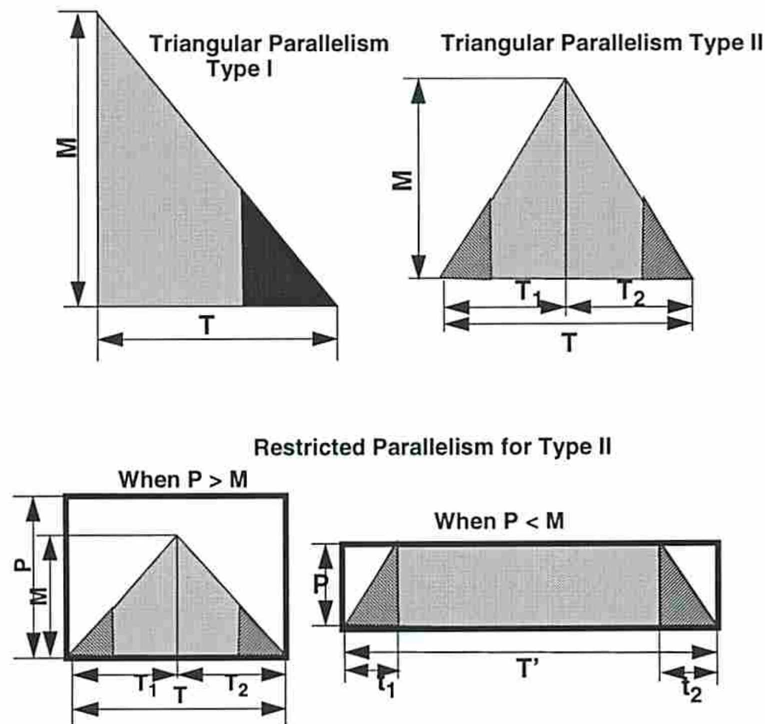

Figure 4.7: Triangular Parallelism I

$$U = 1 - \frac{1}{\frac{M^2}{P^2} + 1} \quad \text{When } P < M \tag{4.7}$$

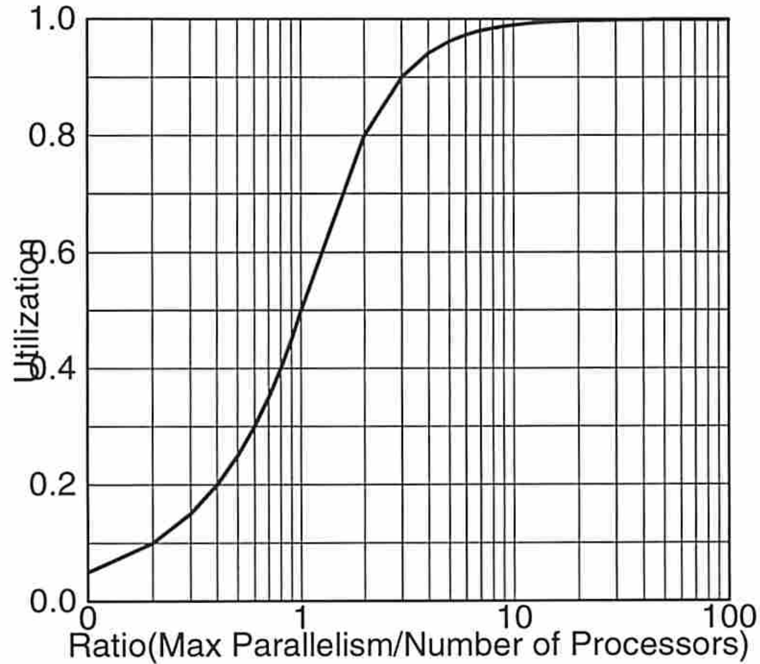

Figure 4.8: Bound of Parallelism for Triangular Parallel Loop

When communication delays are introduced, the critical path is lengthened. As far as the communication delays are not localized the overall shape of parallelism profile will not be changed. But due to the communication delays, the averaged parallelism will be sparse.

The modified critical path T_m is the sum of the original path without communication delays T_o, the added delays due to the increase of service time (slower communication element) T_c and the queuing delays due to various resource contentions T_q. T_c can be determined statically and T_q can be decided by the population traffic. So far we assume that the instantiation of a loop is without delay

and the tasks are evenly distributed without causing a bottleneck of hot point. In real application, the instantiation is not so instant and the instant increase of the logical parallelism can not be obtained. We may assume that T_l time adds to the loop execution path as the overheads of instantiating this loop. In K-bounded loop, this overhead can be amortized by the reuse of this loop instance by the factor of K.

Then we may calculate the utilization limit using the following T_m

$$T_m = T_o + T_c + T_q + \frac{T_l}{K} \tag{4.9}$$

Then we can calculate the average parallelism divided by the factor of $\frac{T_m}{T_o}$ while keeping the same parallelism shape. We assumed that the mappings are well balanced and there is no contention delay.

4.4 Complementary Mapping for Loops of Triangular Parallelism

As explained in the previous section, a nested loop of decreasing parallelism is typical in scientific computation. One example of loop6 is shown above. Inside the *WHILE* loop new array W created every iterations. However, by careful analysis of array W, it can be shown that array W is only updated once as shown figure 4.9. By creating two versions of array W, we can envision a way of parallelizing the loops in multithreaded way, in which the array accesses are totally asynchronous and overlapped with other computation.

If we map the inner loop in the way of *modulo* mapping of iteration number over processing element number, it results an unbalanced computation load at each loop as shown figure 4.10. One way to solve this situation is to apply

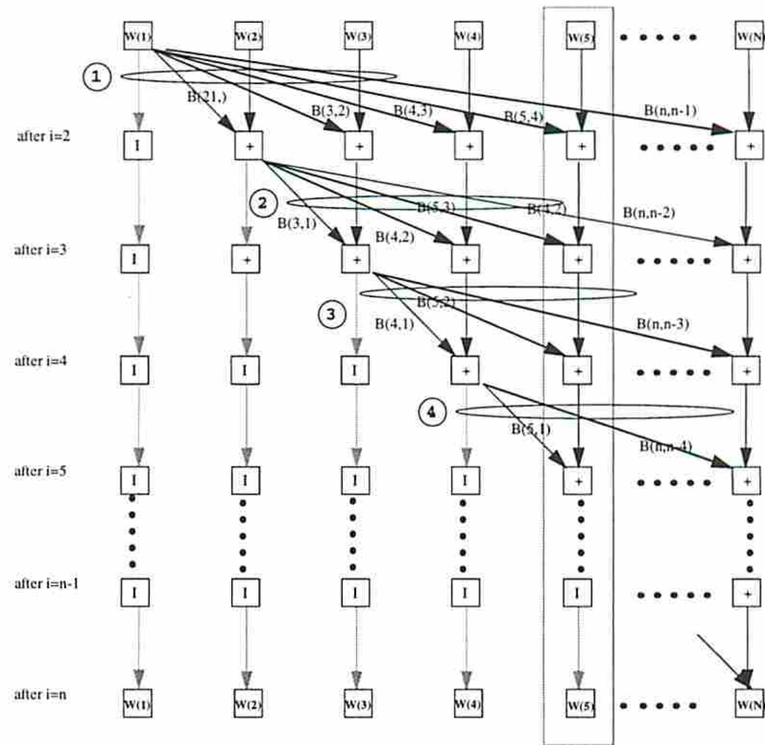

Figure 4.9: Array access pattern in Livermore Loop6

different mapping function for this iteration loop as below, which we call it as *complementary mapping.*

$$\text{Let} \qquad X = \text{LoopIndex} \quad mod \quad (2 * \text{NofPE}) \tag{4.10}$$

$$\text{PE(index)} = \begin{cases} X & \text{if} \quad X < \text{NofPE} \\ 2 * \text{NofPE} - X & \text{otherwise} \end{cases} \tag{4.11}$$

The following figure 4.10 shows the work balance of each loops between the modular mapping and the complementary mapping.

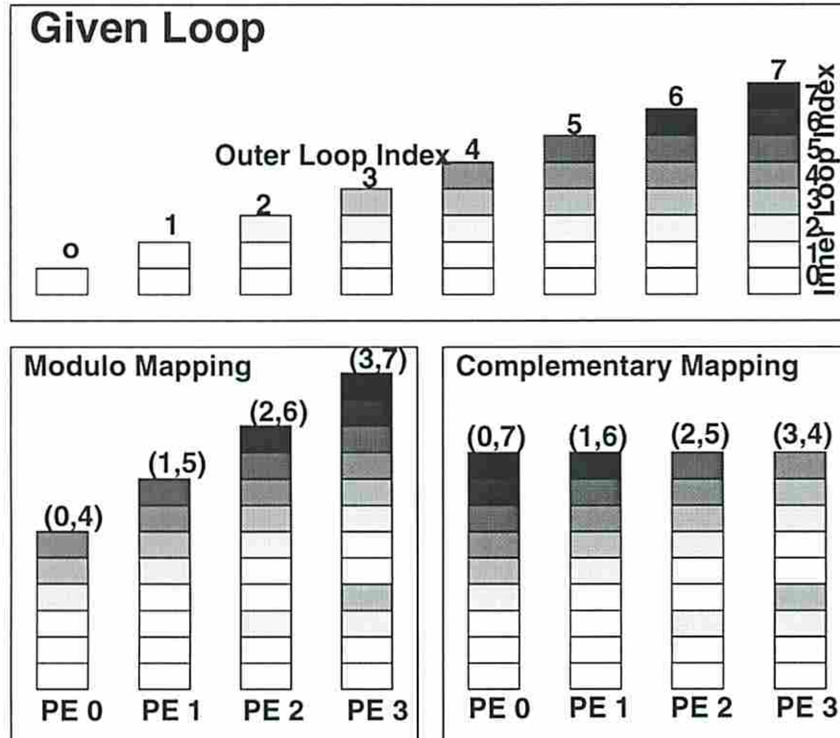

Figure 4.10: Work Balance Diagram of Complementary Mapping

We have performed a simulation of this Loop6 with two strategy of loop mapping. The advantage was clear in case that the system is loaded lightly as shown in the figure 4.11. However, the advantage is not high in case of a highly loaded situation because the other effects such as communication latency and waiting time is more dominant in determining the utilization of system. Nevertheless, this mapping strategy can be utilized by inserting the mapping code without causing any additional overheads compared to the *modulo* mapping.

From this result we conclude the following thing:

- Overall, the utilization is governed by the program's parallelism limit. However, a slight enhancement is observed by using the complementary mapping to the conventional mapping.

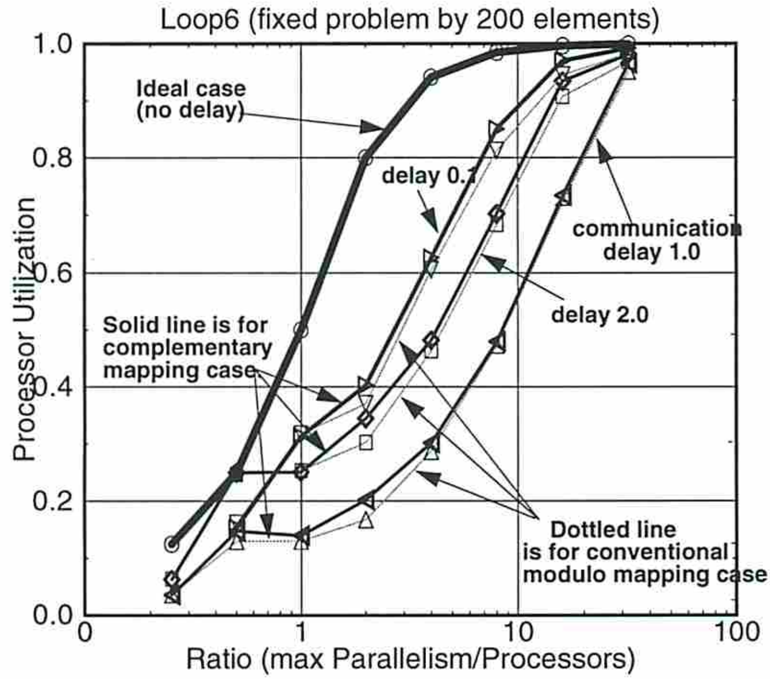

Figure 4.11: The Effect of Complementary Mapping (Simulation Results)

- The multithreaded nature of dynamic execution tolerate the mapping mismatch.

Chapter 5

Balance Equation Theory

In chapter 4, the GMT model and its simulation results indicates that the program characteristics of the average parallelism and the remote access pattern are the key factors to the execution time of programs. in this chapter, we established a performance model between the architectural parameters and the program's workload characteristics. Two major factors are considered: one is the average parameters of program runlength and the other is the average latency of accessing remote elements.

As shown in the figure 5.1, we have partitioned our performance model into three stages: Workload Modeling, System Identification and Balance Formula Setup. In the first stage (1), we describe a workload model of characterizing the program graph. We choose the average runlength of the kernel part of programs. In the second stage (2), we identify the system load and remote load latency. One of the identification is to model the system as queuing networks which provides a relationship between the network traffic (population of packets in the system) and the latency time (3). If this analytic model is hard to be obtained, we can run a synthetic program to measure the relationship among the runlength, the concurrency and the latency. The (RL,K,LAT) table can be obtained by running a synthetic experimental loop (2).

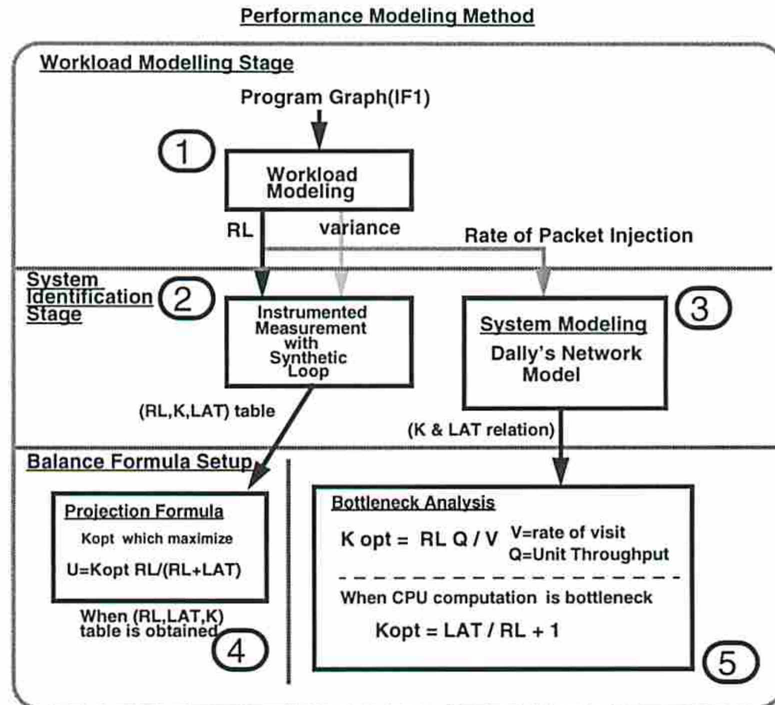

Figure 5.1: Overview to Performance Modeling Method

In the third stage, we set up the formula of matching those parameters. In the projection method (4), we plot the inverse of utilization graph obtained from the (RL,K,LAT) graph and chose the concurrency point of the smallest execution time. Those steps (1), (2) and (4) is termed as as *Projection Method.*

This approach is based upon that the workloads are distributed in a balanced way and the performance can be represented by those *average* parameter. However, if the system is not running at saturating mode, we try to set the concurrency to balance between capacity and the rate of input to the unit.

We will described each numbered stage in the following sections.

5.1 Workload model in multithreaded machine system : the average runlength

A multithreaded program can be modeled by a partially ordered set of threads communicating each other. A thread is scheduled upon receiving a datum and is suspended when it needs to send a remote read/write packet.

Rather than considering general program structures, we consider in this study only independent loops accessing a global array. We believe that this program structure represents the demand of most scientific and engineering computations. The workload can be characterized by a statistical distribution of the run-length size of computation thread. We will describe the threads by their average length. In a loops, the average runlength, R_a, can be determined by :

$$R_a = \frac{R_t}{N_s}$$

where R_t is the total run length of instructions in a loop including all control instructions for synchronization and N_s is the total number of split phased instructions in the loop.

The procedure of finding the average latency from IF1 graph and setting the bound is as follows:

1. (Stage Partitioning) From the IF1 graph of the main loop body, generate a reduced graph merging all non-array access node in the figure 5.2.

   By topologically sorting the reduced graph, tag the stage number to those array accesses nodes. Between the two adjacent stages, calculate the average thread length. In this way, we generate the partitioning for the group of array accesses to be in parallel.

2. Insert the control instructions of generating remote accesses in parallel and synchronizing them.

3. Count the total instruction number as R_t for each stages and the number of remote accesses in each stage as N_s.

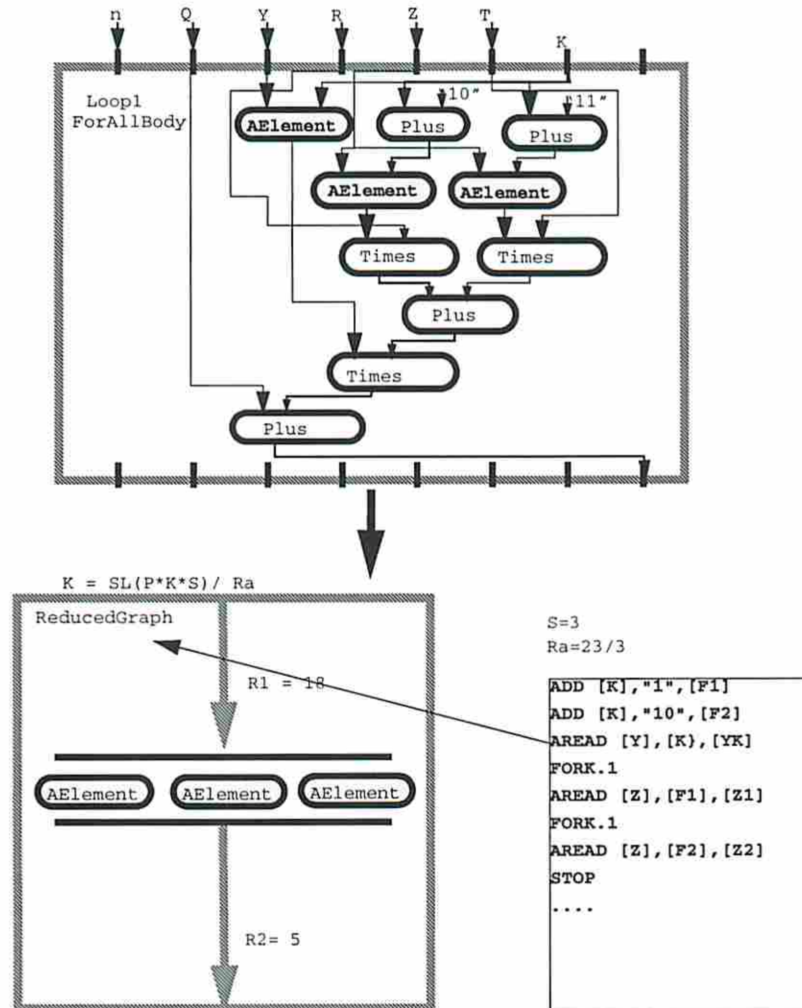

Figure 5.2: Example of Loop1 to thread generation

The above procedure is only applicable to the local analysis of the loop kernels of *loop generate* and *loop gather* of IF1.

5.2 How to obtain the average latency from a network model

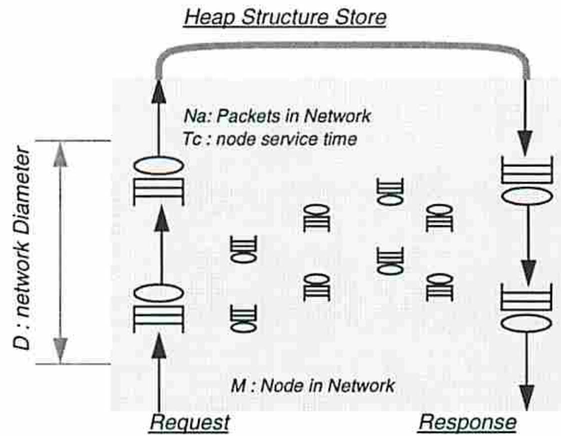

Figure 5.3: Latency Graph and Latency Model

A general network model is shown in figure 5.3. Let D be the average hop distance between any two nodes. Let L_0 be the average no-load latency between any two nodes and T_c be the node routing time.

$$L_0 = 2\ D\ T_c$$

In a lightly loaded network, L_0 is the same as L_a, the average network latency. However, as indicated in the analysis of Dally [9], in a network utilized at 20% of its capacity, the latency increases to twice L_0. We should add this term scaled by a traffic factor function Ftn. The destination processor for read/write packets also involves a delay L_{cpu}. In summary, the average latency would be approximately modeled as follows:

$$L_a \quad = \quad L_0 * Ftn(K) + L_{cpu}$$

$$Ftn(K) = \frac{1}{1 - \frac{N_a}{N_t}}$$

where N_a : Number of active packets in system

N_t : Total Packet capacity of network system

The form of traffic function Ftn is adopted from Dally's analysis [9]. [1]

This function value increases sharply as the N_a approach the N_t. The N_a can be determined by the concurrency factor K.

5.3 Balance equation set by bottleneck analysis

When the workload is unbalanced in using the system resources, system's performance is governed by the unit of hot spot which is usually the slowest device in the system. A good system architect should design the parallel machine to balance among computation units, network units and memory units. However, if any of the system unit is running at this saturating mode, we should engage a different equation to predict the system performance and set the right concurrency level accordingly. We describe this situation by the following equation, where we choose the runlength (of computation thread between context switch in a processor) R and the concurrency by balancing the demand for a particular unit and its throughput:

$$\text{Visit \# per thread for Unit } i \quad : \quad V_1, V_2, \ldots V_N \tag{5.1}$$

$$\text{Unit throughput} \quad : \quad Q_1, Q_2, \ldots Q_N \tag{5.2}$$

$$\text{Balancing K for unit } i \quad : \quad K_1, K_2, \ldots K_N \tag{5.3}$$

[1] Dally's analysis is based upon k-ary n-cube networks. However it can be extended to other network topologies.

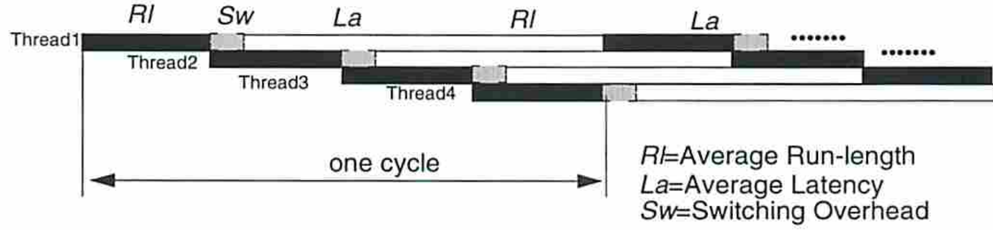

Figure 5.4: Overlapping communication and computation threads

$$K_i \frac{V_i}{R} \quad \rightarrow \quad K_i = \frac{RQ_i}{V_i} \tag{5.4}$$

$$K_{opt} = \min(K_i) \qquad \text{for all } K_i \quad i = [1, N] \tag{5.5}$$

The *Visit Ratio* V_i is the frequency at which a thread requests accesses to the subsystem i between two consecutive execution of that thread at the processor. From the above equation, we can expect that the optimum point is approximately proportional to the run-length of threads, if the bottleneck device is other than the computation unit. [2]. This trend is observed from our EM-4 experimentation which is detailed in the next chapter of this paper.

5.4 Balance equation when the computation unit is the bottleneck

In this case, the run-length R and the visit ratio V_{cpu} are correlated. We setup the balance equation differently from the previous case. We define the optimal point of concurrency as the point where, on the average, during one remote accesses, all the other concurrent threads can be made to run one after the other (see figure 5.4). [3]

[2] R is independent from V_i

[3] We argue that this balance point is where the system is utilized best just as it was determined that the best multiprogramming level would be reached when the page swapping time was the same as the average run-length between page faults.

This optimal concurrency is reached for $K = K_{opt}$ when the average communication latency is "covered" by the K_{opt} computation threads. During one cycle of thread run (R_l), switching (S_w) and waiting for the response (L_a), the minimum number of threads which can be overlapped is represented by the fraction of the cycle time and CPU time ($R_l + S_w$), as shown below:

$$
\begin{aligned}
K_{opt} &= \left\lceil \frac{R_l + S_w + L_a}{R_l + S_w} \right\rceil \\
&= \left\lceil \frac{L_a}{R_l + S_w} + 1 \right\rceil
\end{aligned}
$$

The above equation shows that the optimal point K_{opt} is inversely proportional to the runlength of the computation thread.

5.5 Projection Method

In many real systems, the *LAT*, *RL* and *K* are related non-lineraly due to the complexity of system network and the dynamic nature of program. The finite size of network buffer and the cache effect make the analytic solution of the latency even impossible. Furthermore, the remote memory read involves the computation time of the destination processor (as in EM-4). In this case, the *RL* is correlated the *LAT* and it is hard to obtain the closed form of *LAT*.

Thus, we proposed a method of determining the relationship of *RL* , *LAT* and *K* by instrumented execution of a synthetic loop. From the table obtained from the experimentation, we can project the utilization of processors and the best concurrency from projection of *(RL,LAT,K)* table. The methodology of our projection is as follwoings:

1. Program a small synthetic loop with a known runlength size.

2. The latency of remote accesses should be measure at only one PE node in order not to disturb the system traffic by the measurement itself. We should be using a timer inside PE and accumulate the latency in memory not causing external I/O.

3. Measure those latencies at all range of RL and K, until non-linear rate change is observed.

4. Keep those data points as three dimensional array.

From the table of *(RL,LAT,K)*, we apply the following equation for projecting the utilization.

$$Util = \frac{RL * K}{RL + LAT} = \frac{K}{1 + \frac{LAT}{RL}}$$

We can plot the utilization from the experimented table of (RL,LAT,K). Given a program of which average thread run-length is RL, we can suggest the best concurrency point of K from the plot of this utilization projection.

Although this projection is very practical in estimating the system performance, the accuracy depends upon the followings.

- How accurately we can measure the table of *(RL, LAT,K)* from the real system? In many system, the measurement itself perturb the system's performance. Very careful instrumentation is required. We will discuss this matter in the following EM-4 experimentation (chapter 6.

- The workload on the system is only described by concurrency K of uniform latency RL. In real application, the RL is not uniform and the deviation of the RL would effect the real performance.

	Sakane's	Projection	Newarkars's
Objective	Find # of threads for highest processor utilization depending on the grain size of threads (runlength)		
Benchmark Program	MP3D	Synthetic Loop	Synthetic Loop
Approach	Synthetic Workload Experimentation Curve	Latency Projection, architectural factors, runtime overheads are aggregated into the projection model	Closed form Analysis by AMVA
Validation	EM-x RTL simulation	EM-4 Benchmarking	Earth-MANNA benchmarking

Table 5.1: Comparison Method Average Methods

5.6 Comparison with other approaches

We have outline other performance models of multithreaded machines in chapter 2. Sakane et.al. investigated the multithreading characteristics on the EM-X multiprocessor system.[16] Nemawarkar et.al. applied an analytic model to analyze the performance of the EARTH-MANNA multithreaded multiprocessor system[29]. We compared our projection method in the following table 5.1.

Chapter 6

Experimental Result in EM-4

We experimented the projection method in EM-4 machine of 80 processors, which is a prototype multithreading machine developed by ETL in Japan. The experimental result shows the the projection method can match roughly the actual performance graph, from which we can determine the proper concurrency level. This method can be used for general multithreaded machine, because we do not assume any specific machine type and we based upon the projection from the actual measurement of the *(RL,LAT,K)* table.

6.1 EM-4 Architecture

EM-4 is a highly parallel multithreaded machine, which supports fine-grained parallel computation like pure data-flow and also supports the thread block of *strongly connected arc.*

Therefore, the threaded library supports a multithreaded programming model with shared memory. The system is configured as 16 groups of 5 processor boards. Its network topology is a processor-connected omega network, The processor pipeline is composed of four units: the SU (Switching unit), the IBU (Input Buffer Unit), the FMU (Fetch and Match Unit) and the EXU (Execution Unit). There are 16 registers in the EXU and 32 words in the FIFO buffer of the IBU.

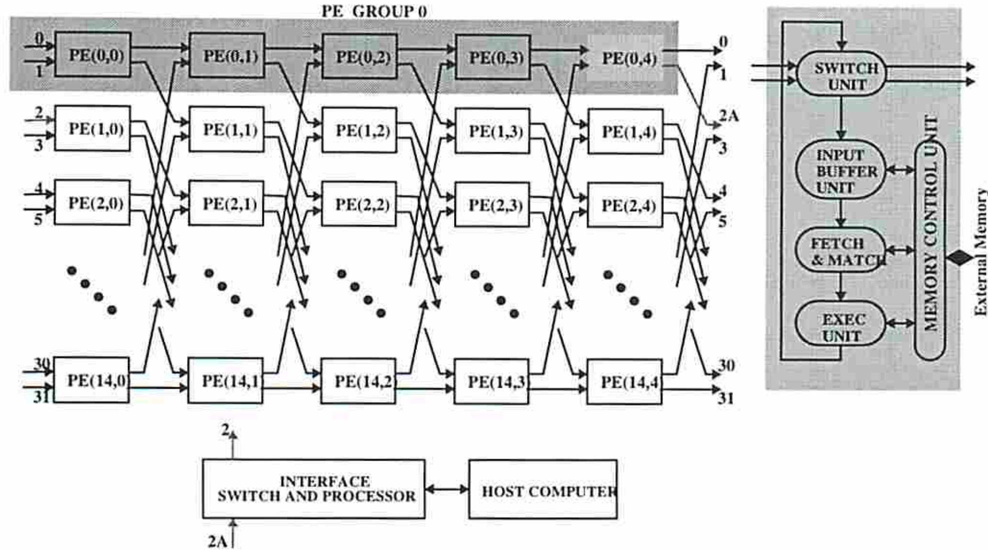

Figure 6.1: EM-4 System Architecture

the SU has *three buffered banks* in order to avoid deadlock. Further monitoring (measuring time) can be performed only by one processor, the IFSW (Interface switch unit).

EM-4 has an extremely efficient communication instruction of sending/receiving packets. Compared to other machines such as CM5, DASH and Monsoon, the overhead of sending a packet is just consuming two CPU cycles and the routing step at each router in the network is just two steps. The comparison table showing the cycle time of sending a packet, and the bit size of packet, the CPU over overhead in sending a packet in cpu cycle, the routing steps in each router is shown in table 6.1.

6.2 RL-LAT-K table in EM-4

We programmed a synthetic loop whose thread run-length can be adjusted by changing the parameter Nd:

```
t = utime();
for(i = 0; i < N_ITER; i++)
```

Machine	cycle	packet width	cpu overhead [cycles]	routing overhead
CM5	25ns	4	3600	8
CM5 (Active Method)	25ns	4	132	8
DASH	30ns	16	10	2
Monsoon	20ns	16	10	2
EM-4	80ns	44	2	2

Table 6.1: Communication Overheads comparison in EM-4

```
for(j = 0; j < N_PE; j++)
  {
      x = mem_read(GLOBAL_ADDR(pe_tbl[j],test_mem));
      for(d=0;d < Nd;d++);
  }
mem_read_utime = utime() - t;
```

Each thread in a processor accesses other processors memory location *testmem* iteratively as shown in the figure 6.2. We can calculate the thread length of this synthetic loop by counting the number of instruction from the compiled code. We can determine that the initial overhead is 11 instructions while the basic run-length is 13 instructions. For each increment of Nd, we added to 7 more instructions to the run-length as the following equation.

$$R_l = N_d * 7 + 11 + 13 \quad \text{Run-length in number of instructions}$$

$$R_l' = 3N_d + 10.37 \quad \text{Run-length in micro second}$$

Now we have run a fixed load synthetic loops of various thread run-length and concurrency level.

Measuring the latency uses the `utime()` library routines which employ the hardware timer in the IFSW PC. The measurement is performed by only one PE

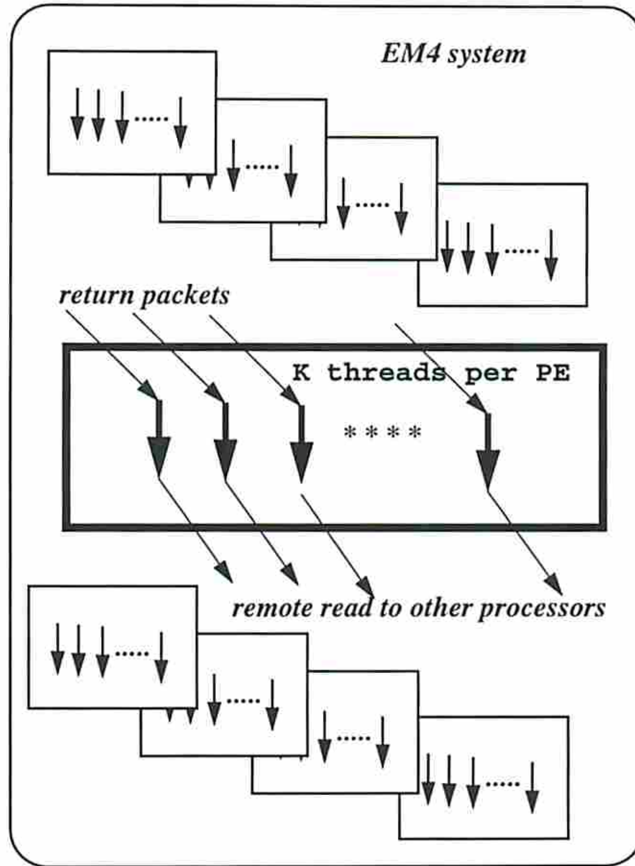

Figure 6.2: EM-4 Experiment Set-up

rather than all PEs, in order not to disturb the system's network demand. [1] So we should have to invoke one measurement thread while there are many threads in a PE making traffics to access other PEs.

In this way, we have measured the tuple relationship of concurrency of threads/PE, remote read latency, and the thread run-length as shown in figure 6.3. We can observe that the latency is linearly increase as the concurrency increase. And there is a change of increase rate around K=7, which is caused by the limited size of EM4 match memory.

[1]There is only one timer in system (IFSW PE). If we allow all threads to access the timer switch, what we measure is the queuing delay at the timer service rather than the latency time of communication.

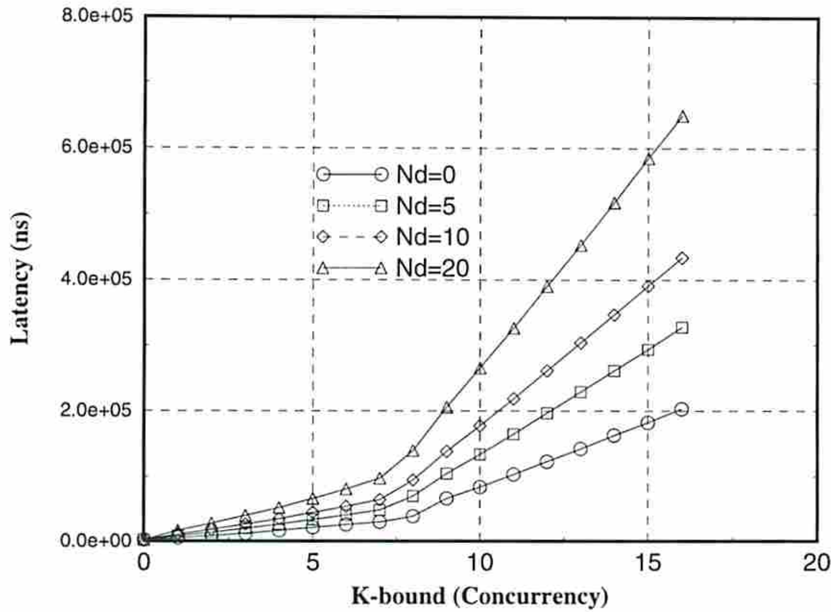

Figure 6.3: EM-4 Latency vs Run-Length vs Concurrency

6.3 Projection Method in EM-4

From the RL-LAT-K table, we can project the utilization curve (figure 6.4) from the derived equation from the balance equation in the previous chapter 5.

From the experimentation we have the following execution graph in figure 6.5, which accords to the projected figure 6.4 from the balance equation. This shows that valley shape and the lowest execution point of for given loop is around 7 or 8 concurrency level.

From the experiment, we plotted the trend of the optimum point for each thread length is shown in figure 6.5. (A) corresponds to the base execution time when there is only one thread in a PE, (B) corresponds to the lowest execution time and (C) corresponds to the points where the execution time is the same as that of single thread. If we trace (A), (B) and (C) for all possible run-length in the EM-4 machine, we can obtain figure 6.5.

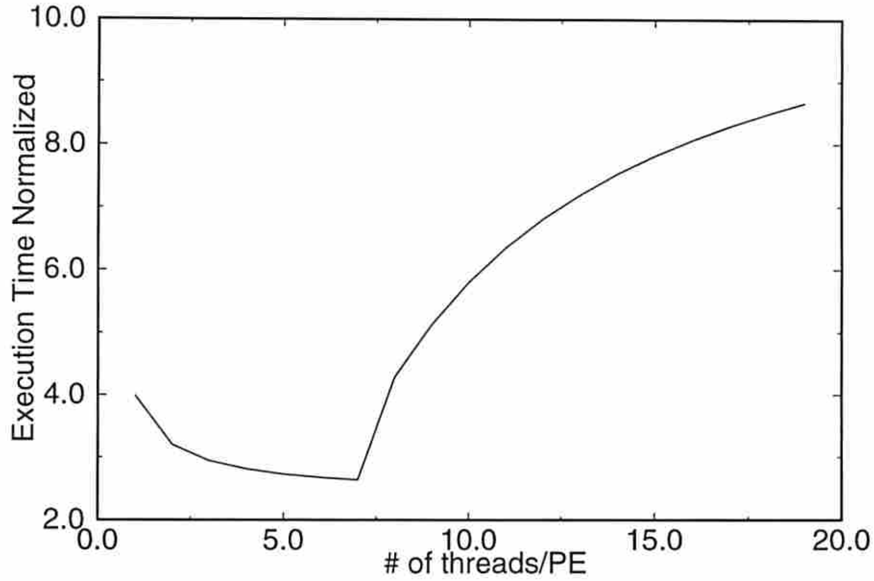

Figure 6.4: EM-4 Normalized Execution Time Projected

Clearly, we have observed that the optimum # of threads increases as the thread run-length increases which correspond well to the bottleneck analysis in the previous section.

We have plotted the theoretic projection of the inverse of utilization graph obtained from the (RL,LAT,K) table and the actual execution curve as shown in figure 6.6. The theoretic graph shows the sharp increase of the execution time after the (A) vertical line but the actual execution increase after the (B) line. However, we can choose the K=7 for minimum concurrency point from the theoretic optimum point of which execution time is almost comparable to that of the lowest execution time as shown in the figure 6.5.

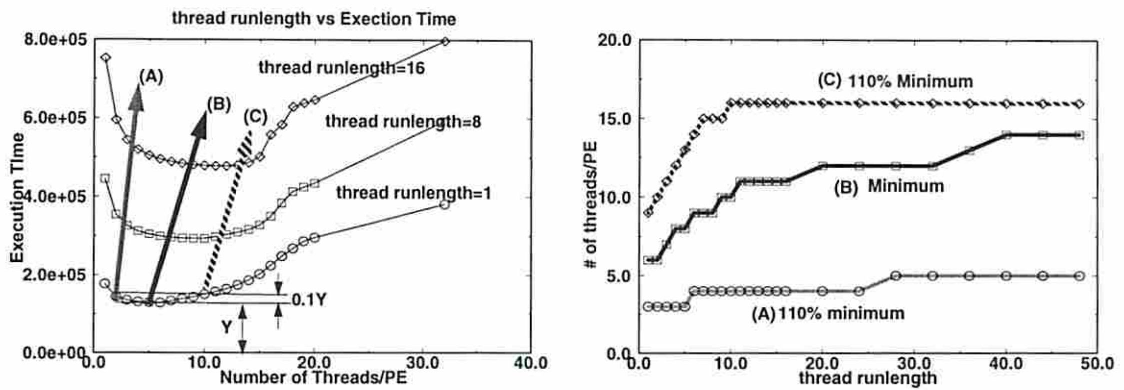

Figure 6.5: EM-4 : The optimum thread number change vs thread run-length

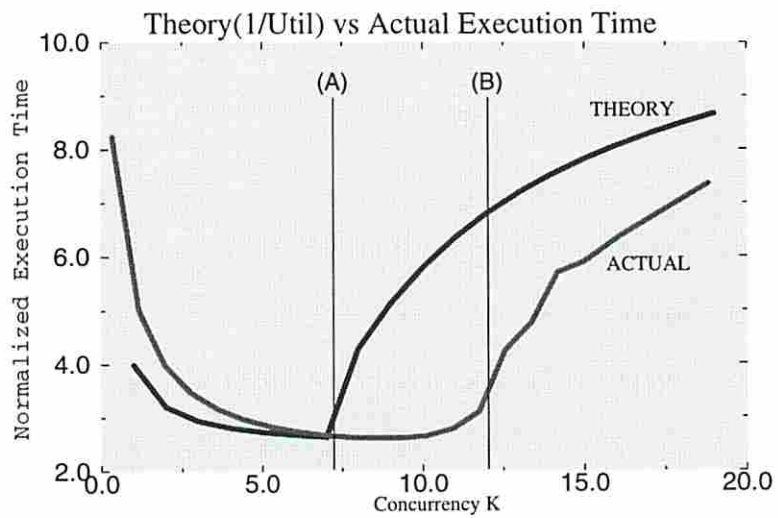

Figure 6.6: EM-4 Experiment Match Curve

Chapter 7

Conclusions and Future Research

This chapter summarizes contributions of this research and suggests future research issues.

We provided a framework for setting the parallelism control in multithreaded computing. Our work was motivated the belief that the overall performance can be determined by the average effect of key parameters such as the run-length of computation threads, the average latency of remote access and the concurrency of threads in a PE.

We have defined a virtual model of multithreaded machine, GMT and implemented its simulator. From the implementation of the GMT, we have tested the effect of centralized and decentralized frame manager. From the test, we observed that the decentralized/static manager design performs better. We also tested parallel and serial access of associative merge in array access and found that the parallel accesses is good while it requires the same amount of storage of thread structure. We presented a method of finding a utilization bound for triangular parallel loop, which is commonly found in Livermore loop kernel. For this kind of loop, we proposed the complementary mapping. The performance advantage is observed, though minimal. This rather small gain shows that the overall performance is governed by the limit of parallelism of given loop and the multithreaded machine offsets the mapping mismatch.

We proposed a simple balance equation set for setting the parallelism. We described two case of relation between the optimum concurrency and the thread run-length: one is proportional and the other is inversely proportional. We suggested method of determining of the latency LAT from the system network analysis and a method of determining of the run-length RL from IF1 graph.

Finally, we proposed a method of determining the best concurrency from projection of *(RL,LAT,K)* table which can be obtained by instrumented execution of a synthetic loop. This projection method is experimented in EM-4 system. The experimental result shows the the projection method can match roughly the actual performance graph, This method can be used for general multithreaded machine, because we do not assume any specific machine type and we based our projection upon the actual measurement of the *(RL,LAT,K)* table.

7.1 Future Research Issues

Future area of this research will be as follows:

- EM-X, which is the next enhancement to the existing EM-4 is under development[35]. This machine has different method of handling remote read access in the destination processor by bypassing the processor pipeline for the simple remote read service. This scheme would reduce the latency of remote read greatly. Even with this variation of the new system, we can apply the projection method in the same way just by measuring new *(RL,LAT,K)* map.

- Effects of variance of average run-length and latency to be studied for synthetic loop of various thread run-length RL in a loop.

- The methodology of finding the average run-length of real application other that loop kernel with conditional branch can be studied.

Bibliography

[1] Arvind and K. P. Gostelow. The U-Interpreter. *IEEE Computer*, pages 42–49, Febrary 1982.

[2] Arvind and R. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[3] Micah Beck and Keshav K. Pingali. Static Scheduling for Dynamic Dataflow Machines. *Journal of Parallel and Distributed Computing*, 10:279–288, 1990.

[4] Gregory T. Byrd and Mark A. Holliday. Multithreaded processor architecture. *IEEE Spectrum*, pages 38–46, August 1995.

[5] C. A. Ruggiero and J. Sargeant. Control of parallelism in the Manchester Dataflow computer. In *Lecture Note in Computer Science No. 274*, pages 1–15, 1987.

[6] D. E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM - A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, June 1993.

[7] D. E. Culler, A. S., K. E. Schauser, T. Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 164–175, Santa Clara, California, April 1991.

[8] D. E. Culler, K. E. Schauser, and T. Eicken. Two Fundamental Limits on Dataflow Multiprocessing. In *The IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 153–164, Orlando Fl., January 1993.

[9] William J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[10] David E. Culler and Arvind. Resource Requirements of Dataflow Program. In *Proceedings of the* 15th *Annual International Symposium on Computer Architecture*, pages 141–150, 1988.

[11] David E. Culler and Gregory M. Papadoupoulos. The Explict Token Store. *Journal of Parallel and Distributed Computing*, 10:289–308, 1990.

[12] Derek L. Eager and John Zahorjan and Edward D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, C-38(3):408–423, March 1989.

[13] P. Evripidou and J.-L. Gaudiot. The USC Decoupled Multilevel Data-flow Execution Model. In J.-L. Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-flow Computing*, pages 347–380. Prentice Hall, 1991.

[14] John Feo. The Livermore Loops in Sisal. Technical Report UCID-21159, Lawrence Livermore National Laboratory, 1987.

[15] Richard P. Gabriel. *Performance Evaluation of Lisp Systems*, chapter 3, pages 81–92. The MIT Press, 1985.

[16] H. Sakane and M. Sato and Y. Kodama and H. Yamana and S. Sakai and Y. Yamaguchi. Dynamic Characteritics of Multithreaded Execution in the EM-X Multiprocessor. In *International Workshop on Computer Performance Measurement and Analysis*, 1995.

[17] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance Studies of the Monsoon Dataflow Processor. *Journal of Parallel and Distributed Computing*, June 1993.

[18] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

[19] R. A. Iannucci, G.R. Gao, R. H. Halstead Jr., and B. Smith. *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 1. The Kluwer Academic Publishers, 1994.

[20] J. -L. Gaudiot and W. Najjar. Macro-actor execution on Multilevel Data-driven Architecture. In *Proceedings of the Working Conference, Parallel Processing, IFiP Pisa, Italy*, April 1988.

[21] J. R. Gurd and C. C. Kirkham and I. Watson. The Manchester Data-Flow Prototype. *Communications of the ACM*, 28(1): 34–52, January 1985.

[22] M. H. MacDougall. *Simulating Computer Systems, Techniques and Tools.* MIT Press, 1987.

[23] N. Yoo and J-L. Gaudiot. The USC macro-data-flow simulator. Technical Report CENG-89-27, USC, October 1989.

[24] N. Yoo and J.-L. Gaudiot. The USC Threaded Data-Flow Machine Simulator. Technical Report CENG-91-xx, USC, July 1991.

[25] G. M. Papadopolous and K. M. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the* 18th *Annual International Symposium on Computer Architecture*, pages 342–351, 1991.

[26] R. S. Nikhil and G. M. Papadopoulos. *T: a Killer Micro for a Brave New World. Technical Report CSGM 325, MIT, Laboratory for Computer Science, January 1991.

[27] Robert A. Iannucci. Toward A Dataflow/Von Neumann Hybrid Architecture. In *Proceedings of the* 15th *Annual International Symposium on Computer Architecture*, pages 131–140, 1988.

[28] R.S.Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the* 16th *Annual International Symposium on Computer Architecture*, pages 262–272. IEEE, 1989.

[29] S. S. Nemarwarkar and G. R. Gao. Measurement and Modeling of EARTH-MANNA Multithreaded Architecture. In *Proceedings of the 4th International Workshop on Modelling, Analysis and Simulation of Computer and Telecommuncation Systems (MASCOTS'96)*, pages 109–114, 1996.

[30] S. Sakai, Y. Kodama, and Y. Yamaguchi. Prototype implementation of a highly parallel dataflow machine em-4. In *Proceedings of the Fifth International Parallel Processing Symposium*, pages 278–286, Anaheim, 1991.

[31] T. Shimada, K. Hiraki, and S. Sekiguchi. The SIGMA-1 project. In *Some Where ??*, pages 3–9, 1988.

[32] A. Sohn, Sakai, N. Yoo, and J-L. Gaudiot. Effects of Multithreading on Data and Workload Distribution for Distributed-Memory Multiprocessors. In *Proceedings of the Nineth International Parallel Processing Symposium*, 1996.

[33] V. G. Grafe and G. S. Davidson and J. E. Hoch and V. P. Holmes. The epsilon Dataflow Processor. In *Proceedings of the* 16th *Annual International Symposium on Computer Architecture*, pages 36–45, Sandia National Laboratories Albuquerque, New Mexico, 1989.

[34] David W. Wall. Limit of Instruction-Level Parallelism. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, California, April 1991.

[35] Y. Kodama and H. Sakane and M. Sato and H. Yamana and S. Sakai and Y. Yamaguchi. The EM-X Parallel Computer: Architecture and Basic Performance. In *Proceedings of the* 22nd *Annual International Symposium on Computer Architecture*, pages 14–23, 1995.

Appendix A

GMT Simulator Implementation

A.1 SMPL

We have implemented the GMT model using the SMPL library[22], which is a collection of routines that can be used to implement discrete event simulators. Users need to write their own main program, where all the facilities in the system and actions for all the events are defined.

This library provide the basic model of discrete model of service, queue handling and statistic measurement. Programmer is repsonsible for designing the basic model of his system describing the service and routing of the customers (tokens).

Fundamental SMPL service calls are described as below:

- `facility(NAME,NUM)`: The call create a facility named NAME and returns the facility number which have NUM servers.
- `cause(EVENT, TOKEN)` : This call retrieves the current event structure, EVENT, and the token pointer, POINTER from the head of global event queue.
- `schedule(EVENT,FACILITY,TOKEN)`: This call schedules EVENT to the facilty FACILITY with the token structure of TOKEN.
- `request(FACILTY,TOKEN,TIME)` : This call requests the service of facility after the amount of TIME from the current time. If the FACILITY is idle, this call returns TRUE and the facility is tagged as occupied.
- `release(FACILTY,TOKEN)` : This call releases the token from FACILITY. After this release, the FACILITY is available for use.
- `simtime()`: This function returns the current simulation time.

Using these basic calls, we present the pseudo code of the GMT simulator's main loops as below.

```
   do{
cause(&event_case, &tkn);
SimulationTime= simtime();
switch(event_case){
        case NULL:
            // Simulation abort

case SCHEDULER_ARRIVAL:
    // if toekn does not reach to the destination processor
            //   schedule for the NETWORK_OUT service.
            // or
    //   perform the SCHEDULER service.
            //   schedule the SCHEDULE_DEPARTURE

case SCHEDULER_DEPARTURE:
            // release the scheduler facility
            // release the structure of toekn and packet

case NETWORK_OUT_ARRIVAL:
            // find the destination processor's facility number
            // simulate the network delay
            // schedule NETWORK_IN_ARRIVAL and NETWORK_OUT_DEPATURE

case NETWORK_OUT_DEPARTURE:
            // release NETWORK_OUT facility of the current processor.

case NETWORK_IN_ARRIVAL:
            // request the NETWORK_IN service in current processor
            // if not queued
            //    schedule the NETWORL_IN_DEPATURE
            //    if current  packet is ARRAY_ACCESS
            //           schedule STUCTURAL_ARRIVAL
            //    if current packet is FRAME memory asking
            //           schedule MANGER_ARRIVAL
            //    otherwise
            //           schedule the SCHEDULER service.
            //


case NETWORK_IN_DEPARTURE:
            // release the current NETWORK_IN facility

case EXECUTE_ARRIVAL:
             // get the current activation
             // request the current processing element
             // if not queued
             //    execute the activation
```

```
    case EXECUTE_DEPARTURE:
                // release the current activation and facility

    case STRUCTURE_ARRIVAL:
                // if the destination pprcessor is not reached
                //    schedule NETWORK_OUT_ARRIVAL
                //
                // else
                //    request the current STRUCTURE
                //      if not QUEUED
                //         service the array access


    case STRUCTURE_DEPARTURE:
                // release the current STRUCTURE facility

       case MANAGER_ARRIVAL:
                // if the destination pprcessor is not reached
                //    schedule NETWORK_OUT_ARRIVAL
                //    request the current MANAGER
                //      if not QUEUED
                //         service the manager access.


    case MANAGER_DEPARTURE:
                // release the current MANAGER facility
          };

   } while(event_case!=END_SIM);
   backclock(); // back to the last event.
```

A.2 Simulation Interface

A.2.1 Batch Command Parameters

- `-pe #` : Set the number of processing element to simulate.
- `-prog filename` : Load program named `filename`.
- `-trace #` : Set the trace mode to `#`.
- `-histo filename #` : Set the histogram file to `filename` and set the histogram sensitivity to `#`.

A.2.2 Interactive Commands

When you run GMT, you will be prompted by `GMT>` . then you can load programs or run program by the following commands.

- `!` : run unix command.
  `!dir` : ls *.l
  `!asm fname` : cc -E fname.asm ¿ fname.l
- `help` : this message will be printed.
- `cl` : clear all loaded programs.
- `reset` : reset smpl kernel for rerun.
- `load [filename]` : load program file.
- `command [filename]` : load command file.
- `list [cbno]` : list names of loaded code block or list the disassembled code block if [cbno] is provided
- `set [p|t|c] [#]` : set # of some hard ware parameter. (defaults=1) (p:processor #) (c:ratio of communication/process time) (t:TraceMode)
- `histo [histogram_file_name] [histogram_sensitivity]` :
  example : `histo FILE1 100`
- `status` : show simulator status
- `run` : run the loaded program.
- `report` : report smpl statistics.
- `br [c|b|i|x|a] [#]` : set break point at codeblock [cbno] run.
  (c:color) (b:cbno) (i:pc) (a:array id)
  (x:clear break point at current cbno)
- `pr [a|f|q|t|i|s|p|v] [#]` :
  print Activation—Frame—Frame[#] which is debugged.
  (a:activation) (f:frame) (q:thread queue) (t:thread) (i:single frame)
  (s:stopped array) (p:stopped packet) (v:array #)
- `pr s [color] [cbno]` : set current activation for print.
- `c` : continue until next activation break.
- `n` : next instruction.
- `q` : quit this program.

A.3 BNF Definition of the GMT assembly language

⟨digit ⟩ →**0** |**1** |**2** |**3** |**4** |**5** |**6** |**7** |**8** |**9**

⟨identifier ⟩ → ⟨non-number ⟩ ⟨non-blank-character ⟩

⟨integer ⟩ → ⟨digits ⟩

⟨real ⟩ → ⟨integer ⟩. ⟨integer ⟩

⟨relational-operator ⟩ →**EQ** |**LE** |**LT** |**GT** |**LT** |**GE**

⟨label-identifier ⟩ → **:** ⟨identifier ⟩

⟨immediate-data ⟩ → " ⟨real-string ⟩ " | " ⟨integer ⟩ "

⟨frame-identifier ⟩ → [⟨identifier ⟩] | [⟨digit ⟩ ⟨identifier ⟩]

⟨array-identifier ⟩ → ⟨frame-identifier ⟩

⟨codeblock-identifer ⟩ → ⟨codeblock-name ⟩ **.** ⟨port ⟩

⟨codeblock-name ⟩ → ⟨identifier ⟩

⟨port ⟩ → ⟨integer ⟩

⟨var-operand ⟩ → **.0** |
  .1 ⟨frame-identifier ⟩ |
  .2 ⟨frame-identifier ⟩ ⟨frame-identifier ⟩ |
  .3 ⟨frame-identifier ⟩ ⟨frame-identifier ⟩ ⟨frame-identifier ⟩

⟨codeblock ⟩ → =CodeBlock ⟨no-of-firing ⟩ ⟨new-line ⟩ ⟨list-of-instructions
⟩

⟨no-of-firing ⟩ → ⟨integer ⟩

⟨list-of-instructions ⟩ → ⟨instruction ⟩ ⟨list-of-instructions ⟩

⟨instruction ⟩ → > ⟨instruction ⟩

⟨instruction ⟩ →
  NOP.1 |
  MOV.1 ⟨frame-identifier ⟩ |
  MOV.2 ⟨frame-identifier ⟩ ⟨frame-identifier ⟩ |
  LD.1 ⟨immediate-data ⟩ |

LD.2 ⟨frame-identifier ⟩ ⟨immediate-data ⟩ |
SWAP.0 |
ADD ⟨var-operands ⟩ |
SUB ⟨var-operands ⟩ |
MUL ⟨var-operands ⟩ |
DIV ⟨var-operands ⟩ |
INC ⟨var-operands ⟩ |
DEC ⟨var-operands ⟩ |
JUMP ⟨label-identifier ⟩ |
CBR ⟨relational-operator ⟩ ⟨label-identifier ⟩ |
STOP |
START ⟨label-identifier ⟩ |
FORK ⟨label-identifier ⟩ |
JOIN |
WAIT.[0 |1 |2] |
AREAD ⟨array-identifier ⟩ ⟨frame-identifier ⟩ |
AWRITE ⟨array-identifier ⟩ ⟨frame-identifier ⟩ |
SEND ⟨codeblock-identifer ⟩ ⟨frame-identifier ⟩ |
RSEND ⟨codeblock-identifer ⟩ ⟨frame-identifier ⟩ |
RET |
TRIG |
GNF.[0 |1 |2] |
GNA.[0 |2] ⟨frame-for-array-size ⟩ |
PRT |
EXIT |
DEBUG_POINT

A.4 Sample Programs

A.4.1 Function Calls

Fundamentally the caller should new context for the called function name. This is done by `GNF` instruction in split phase. After receiving the new color name then argument sending is done by `RSEND` instruction. And the sending thread identification should be sent as an zero argument to the caller activation. After sending the argument the current thread is waiting to receive the return value.

Then called function block can access the frame data by accessing the addressed frame location. After processing the value the return data will be sent back to the caller by `TRIG` instruction.

The fibonacci sample program is listed as below;

A.4.2 Fibonacci - Recursive Function Call

```
%
% =====
% define fibo
%
% function fibo (n:integer returns integer)
%      if n < 0 then
%         0
%      elseif n = 1 then
%         1
%      else
%         fibo (n-1) + fibo (n-2)
%      end if
% end function
%
%

#define VAL "16"

#define CALL(function, arg0, arg1, label) \
           > GNF.0 \
           > RSEND function.0 arg0 \
           > RSEND function.1 arg1 \
           > WAIT.1 label \
           :label

=CodeBlock main 0 10
           > LD.2 [x] VAL
           > LD.2 [thread0] "0"
           CALL (fibo, [thread0], [x], RET_POINT)
           > PRT.0                % Print return value.
           > EXIT

=CodeBlock fibo 2 10
    > MOV.2 [1] [n]
    > LD.1 [n]
    > CBR LE RET1
    > DEC.0
    > CBR EQ RET2
    > LD [my_thread_id] "0"
    > DEC.1 [n]
    > GNF.0              % First Call
    > RSEND fibo.0 [my_thread_id]
    > RSEND fibo.1 [n]
    > DEC.1 [n]
    > GNF.0           % Second Call
    > RSEND fibo.0 [my_thread_id]
    > RSEND fibo.1 [n]
    > WAIT.2 JOIN
:JOIN
    > ADD.0 % Auxiliary and Accummulator for this thread are added together.
    > MOV.1 [result]
    > RET [result] [0]
    > EXIT
:RET1
    > LD.2 [result] "0"
    > RET [result] [0]
    > EXIT
:RET2
    > LD.2 [result] "1"
    > RET [result] [0]
    > EXIT
```

A.4.3 Tak - Recursive Function Call

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% SAMPLE PROGRAM for GMT
% programmed by Namhoon Yoo
%
% tak.asm : Function Call Management
% Sample Program of TAK..
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%              2 3 4    = 4
%              5 4 2    = 4
%              6 4 2    = 3
%              8 6 2    = 3
%              10 6 2    = 3 (1734 calls)
%              12 6 2    = 3 (10000 calls)
%              18 12 6   = 3 (63609 calls)
#define XX "12"
#define YY "6"
#define ZZ "2"


#define CALL(function, arg0, arg1, arg2, arg3,label) \
           GNF.0 \
           > RSEND function.0 arg0 \
           > RSEND function.1 arg1 \
           > RSEND function.2 arg2 \
           > RSEND function.3 arg3 \
           > WAIT.1 label \
           :label


=CodeBlock main 0 100
           > LD.2 [thread0] "0"
           > LD.2 [x] XX
           > LD.2 [y] YY
           > LD.2 [z] ZZ
           CALL (tak, [thread0], [x], [y], [z], RET_POINT)
           > PRT                % Print return value.
           > EXIT

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ( defun tak ( x y z)
%     (if (>= y x)
%         z
%         (tak (tak (1 - x) y z)
%              (tak (1 - y) z x)
%              (tak (1 - z) x y))))
%
%
=CodeBlock tak 4 100
           > MOV.2 [0] [return_thread_no]
           > MOV.2 [1] [x]
           > MOV.2 [2] [y]
           > MOV.2 [3] [z]
           > SUB.2  [y] [x]
           > CBR GE QUICK_END
           > LD.2 [one] "1"
           > SUB.3 [x] [one] [x1]
           > SUB.3 [y] [one] [y1]
           > SUB.3 [z] [one] [z1]
```

```
            > LD.2 [counter] "3"  % for thread counting

            > FORK.2 CALL1
            > LD.2 [thread1] "1"
            > FORK.2 CALL2
            > LD.2 [thread2] "2"
            > FORK.2 CALL3
            > LD.2 [thread3] "3"
            > STOP
:QUICK_END
            > RET [z] [0]   % Return Trigger
            > EXIT
:CALL1
           CALL (tak,[thread1],[x1],[y],[z],RET1)
            > MOV.1 [xxx]
            > JUMP JOIN
:CALL2
           CALL (tak,[thread2],[y1],[z],[x],RET2)
            > MOV.1 [yyy]
            > JUMP JOIN
:CALL3
           CALL (tak,[thread3],[z1],[x],[y],RET3)
            > MOV.1 [zzz]
:JOIN                                  % Join Location
            > DEC.1 [counter]
            > LD.1 [counter]
            > CBR EQ NEW_CALL
            > STOP

:NEW_CALL
            > FORK.2 NEW_CALL2
            > LD.2 [thread4] "4"
            > STOP
:NEW_CALL2
            CALL(tak,[thread4],[xxx],[yyy],[zzz],END)
            > MOV.1 [final]           % acc  final
            > RET [final] [return_thread_no]         % Return Trigger
            > EXIT
```

A.4.4 Matrix Multiplication (MMUL) : Array Access and Iteration

```
%
% SAMPLE PROGRAM for GMT
% programmed by Namhoon Yoo
%
% Array Multiplication : Paralle
%

#define ARRAY_L_SIZE "10"
#define ARRAY_M_SIZE "10"
#define ARRAY_N_SIZE "10"

=CodeBlock create_array_A_B 0
    > LD.2 [l_size] ARRAY_L_SIZE
    > LD.2 [m_size] ARRAY_M_SIZE
    > LD.2 [n_size] ARRAY_N_SIZE
    > LD.2 [one] "1.0"
    > MUL.3 [l_size] [m_size] [a_size]
    > MUL.3 [n_size] [m_size] [b_size]
    > GNA.2 [a_size] [one]
    > MOV.1 [array_name_A]
    > GNA.2 [b_size] [one]
    > MOV.1 [array_name_B]
    > SEND array_mult.0 [array_name_A]
    > SEND array_mult.1 [array_name_B]
    > EXIT

=CodeBlock array_mult 2 0
    > MOV.2  [0] [array_name_A]
    > MOV.2  [1] [array_name_B]
    > LD.2 [one] "1"
    > LD.2 [l_size]  ARRAY_L_SIZE
    > LD.2 [m_size]  ARRAY_M_SIZE
    > LD.2 [n_size]  ARRAY_N_SIZE
    > MOV.2 [l_size] [ctr1]
    > LD.2 [ixa0] "0"
    > SUB.3 [ixa0] [m_size] [ixa0]
:LOOP_1
    > ADD.3 [ixa0] [m_size] [ixa0]
    > LD.2 [ixb0] "0"
    > DEC.1 [ixb0]
    > MOV.2 [n_size] [ctr2]
:LOOP_2
    > INC.1 [ixb0]
    > FORK.2 LOOP_CHECK
    > GNF.0
    > RSEND par_vec_prod.0 [array_name_A]
    > RSEND par_vec_prod.1 [array_name_B]
    > RSEND par_vec_prod.2 [ixa0]
    > RSEND par_vec_prod.3 [one]
    > RSEND par_vec_prod.4 [ixb0]
    > RSEND par_vec_prod.5 [n_size]
    > RSEND par_vec_prod.6 [m_size]
    > STOP
:LOOP_CHECK
    > DEC.1 [ctr2]
    > LD.1 [ctr2]
    > CBR GT LOOP_2
    > DEC.1 [ctr1]
    > LD.1 [ctr1]
    > CBR GT LOOP_1
```

```
    > EXIT


=CodeBlock par_vec_prod  7 0
    > MOV.2 [0] [Avec]
    > MOV.2 [1] [Bvec]
    > MOV.2 [2] [ixa0]
    > MOV.2 [3] [ixad]
    > MOV.2 [4] [ixb0]
    > MOV.2 [5] [ixbd]
    > MOV.2 [6] [iter]
    > MOV.2 [iter] [cntr]
    > LD.2 [sum] "0.0"
    > SUB.3 [ixa0] [ixad] [ixa]
    > SUB.3 [ixb0] [ixbd] [ixb]
:LOOP
    > ADD.3 [ixa] [ixad] [ixa]
    > ADD.3 [ixb] [ixbd] [ixb]
    > DEC.1 [iter]
    > LD.1 [iter]
    > CBR LT STOP
    > FORK.2 LOOP
    > AREAD [Avec] [ixa]
    > AREAD [Bvec] [ixb]
    > WAIT.2 JOIN
:STOP
    > STOP
:JOIN
    > MUL.0
    > ADD.1 [sum]
    > MOV.1 [sum]
    > DEC.1 [cntr]
    > LD.1 [cntr]
    > CBR EQ EXIT
    > STOP
:EXIT
    > EXIT
```