Computer Science
Technical Report

Code Generations, Evaluations, and Optimizations in Multithreaded Executions

Lucas J. Roh

April 28, 1995

Technical Report CS-95-102

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

DISSERTATION

CODE GENERATIONS, EVALUATIONS, AND OPTIMIZATIONS IN MULTITHREADED EXECUTIONS

Submitted by

Lucas J. Roh

Department of Computer Science

In partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 1995

ABSTRACT OF DISSERTATION

CODE GENERATIONS, EVALUATIONS, AND OPTIMIZATIONS IN MULTITHREADED EXECUTIONS

Efficient large-scale parallel processing can result only from proper handling of latency. Latency arises either from remote memory accesses or synchronizations. Multithreading is an execution model that can effectively deal with latency by switching among a set of ready threads. This model has been proposed in a variety of forms: a unit of storage can be based on either a collection of threads or a single thread, threads can be either blocking or non-blocking, and synchronization can be either implicit or explicit.

This dissertation describes research in the evaluation and optimization of various issues in multithreading. Issues of particular interest are the development of a multithreaded execution model to be used as a test-bed and a hybrid code generation scheme where threads are generated in a top-down manner and then optimized in a bottom-up fashion. Various forms of locality are also identified and studied. The results are used in designing an effective storage hierarchy that reduces the synchronization cost and improves the execution time. Other issues that are tackled include developing techniques that better manage various resources such as network and memory resources. The emphasis is placed on the performance evaluation of various design alternatives.

Lucas J. Roh
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 1995

ACKNOWLEDGEMENTS

There are many people that I would like to thank who have made the long, arduous journey to a milestone represented by this dissertation more palatable and perhaps even fun. Above all, I would like to thank my advisor, Walid Najjar, for all the guidance and support. His energy and ideas always kept me on my toes, and most of all, I have learned tremendously from him.

I would like to also thank Wim Bohm for the chance to work closely together in the Pebbles project. His insights, the ability to present things concisely, and his levity have made my stay at CSU all the more worthwhile.

To my cubicle-mate, Jeff Hammes, I thank you for all the fun discussions that we had. Also, I thank Bhanu Shankar for being part of the Pebbles project and with whom I have commiserated and laughed together.

I would like to thank all my relatives who have supported me, especially Aunt Mary who have always made me feel right at home in Colorado and Uncle Mike who have provided a good friendship and guidance. I would like to especially show a deep heartfelt gratitude to two people, Uncle Vicent and Uncle Joseph, who have made a huge impact on my life. Whether they know it or not, their wisdom and worldliness over the years have shaped my intellect and opened my mind. Their advice and wisdom will be forever with me.

DEDICATION

To my parents, who sacrificed everything for our education.

CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Many parallel processing models and paradigms have been introduced in the last three decades. A consensus does seem to be currently building around a system-wide organization of processing elements, as shown in Figure 1.1. Each node contains some local memory and a number of execution processors, and these nodes are interconnected via a high-speed communication network. The main advantage of this distributed memory parallel machine organization lies in its scalability. Two open questions are the subjects of active research: developing an efficient processor execution model which forms a building block of the distributed memory parallel machines, and developing a programming model that is both portable and scalable. The use of a shared address space, either via an explicit hardware support or through a run-time support, is helping to make the programming process easier.

Multithreading has been proposed as a promising processor execution model for building large scale parallel machines. It is based on the execution of threads of sequential code that cooperate to solve a given problem; the threads are dynamically created and asynchronously scheduled. The basic tenet of multithreading is latency hiding by switching among a set of ready threads, thus improving processor utilization. Because each processor has multiple threads of control, generating and coordinating these threads pose a significant challenge, and results in large hardware design space.

This thesis explores multithreading at the interface between hardware designs and code generators. The overall objective is the evaluation of various hardware and software alternatives based on their performance impacts. The next section discusses some of the important highlights in past attempts to design multiprocessors, focusing on their strengths and weaknesses. The discussion naturally leads to the motivation for multithreading. Then, some of the issues, and their variations, in multithreading are discussed. Different compilation strategies are then briefly discussed. Finally, the overview of the rest of this thesis is introduced.

1.1 Critique of Past Multiprocessors

One fundamental issue that must be addressed by any processor design concerns *program locality*. By properly exploiting locality through the use of a storage hierarchy (e.g., registers and caches), performance can be improved and cost lowered. Multiprocessing introduces two additional issues [AI87]: *latency* and *synchronization*. Latency arises from the need to communicate among physically separate processors (e.g., remote memory reads). Either this latency should be reduced, or masked by doing some other useful work. Synchronization is needed because the work associated with a given program must be broken down and distributed among processors, and there exists a need to coordinate these activities. In this section, the von Neumann and dataflow models are evaluated based on their handling of the above issues.

von Neumann Model. The implicit, sequential execution of the von Neumann model can effectively exploit program locality. In addition, sophisticated hardware mechanisms have been developed to dynamically exploit parallelism at the instruction level (e.g., pipelining, delay slots, cache prefetching, and multiple instruction issue) or at the data stream level in a limited way (e.g., vector processing). Some studies suggest a reasonable amount of significant instruction level parallelism may be available [Wal93].

A great variety of von Neumann-based multiprocessor execution models have been built or proposed. The SIMD model (such as MasPar, DAP, CM2, and IBM GF11) has one control unit which executes the program and issues commands to a number of identical data processors. Programming is relatively simple due to the synchronous nature of execution, and it is most suitable for data-parallel applications where 0 operations across whole sets of data are performed. In the traditional distributed memory MIMD model (such as Ncube and Intel Paragon), each processing node contains its own local memory which is inaccessible from

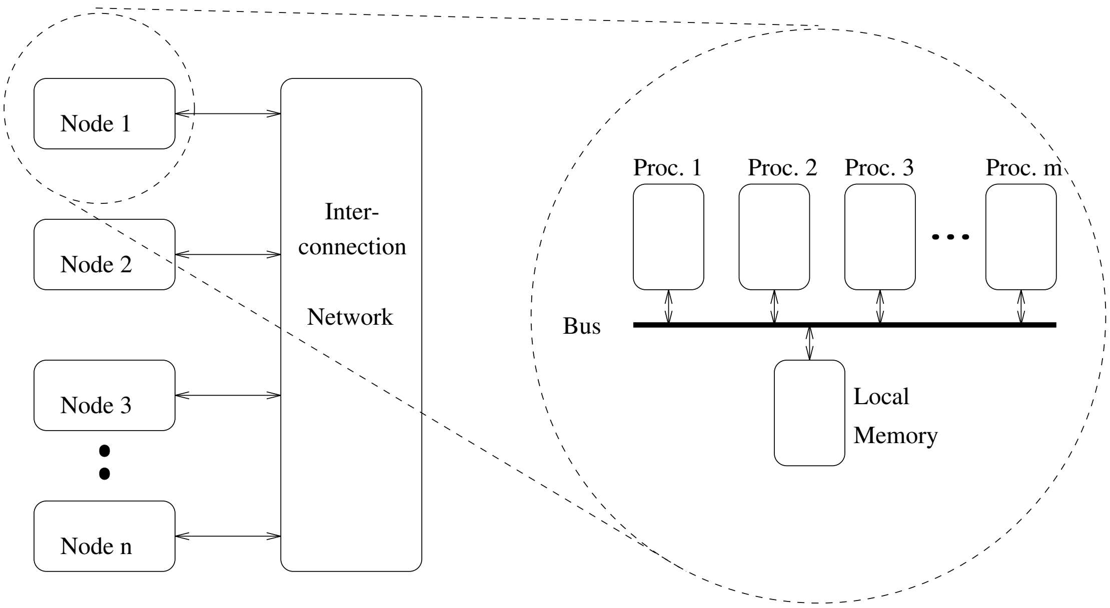


Figure 1.1: Generic MPP Architecture

other nodes, and each executes an independent instruction stream. It communicates via message passing, which has relatively high communication overhead and latency, so this model is most suitable for executing task-parallel or coarse-grain tasks.

The distributed memory MIMD with shared address space model (such as CM5, KSR1, T3D and IBM SP2) is gaining in popularity. This model provides a direct hardware method to access another node's local memory. Some of the newer machines such as T3D and IBM SP2 are characterized by *relatively* low network latency. The main advantage of these machines their ability to support a greater number of programming models compared to the other two von Neumann models.

In summary, the von Neumann execution paradigm is characterized by a single thread of control per processor. When an operation of potentially long and unpredictable latency takes place, such as a remote memory read or a synchronization, the processor pipeline halts and the processor remains idle while the operation completes. The traditional techniques to cover latency, such as the use of caches and out-of-order execution, fail in the presence of long and unpredictable latency. Switching to another task is usually cost-prohibitive for a simple remote memory read. However, in executing instruction sequences that have no long latencies, the von Neumann model's efficiency and use of locality gives it an important advantage.

Fine-grain Dataflow Model. Dataflow is an inherently parallel execution model which represents computations by directed graphs. Highlights of the model are only briefly treated here; for further details, see [AC86, GE91]. In the fine-grain dataflow model, nodes of the dataflow graph represent instructions and edges represent data paths through which operands flow. A node can execute when all its operands arrive and all such nodes can "fire" concurrently. The only constraint on the execution order of instructions is their true data dependencies; thus the model exposes maximal parallelism at all levels including function, loop, and instruction levels.

The dataflow architecture consists of a number of processing nodes that are interconnected via a network. Each processing node consists of an execution unit that executes the instructions and a matching unit that stores incoming operands and enables ready instructions to be executed later. The instruction-level scheduling decisions are left entirely to the run-time system and the operand arrival order. Operands are carried in packets known as tokens. In dynamic dataflow, each token carries its own context information. An instruction can be executed on any processor so the model is naturally scalable. Some of the more notable fine-grain dataflow designs include MIT's Tagged-Token Dataflow Architecture (TTDA) [ABM87, AN90], the Manchester Dataflow Machine [GKW85], and SIGMA-1 [YSHK85].

The problems associated with pure fine-grain dataflow have been well treated in the literature [Ian88, MNB91, Cul89, PT91]. Important points can be summarized as follows:

- Resource management is a problem due to large potential parallelism. Because maximal parallelism has been exposed, the machine is forced to handle a large peak demand for resources [Cul89]. For example, matching operations implicitly allocate resources within the matching store. A deadlock may occur if the resource requirement ever exceeds the capacity.
- Synchronization at the instruction level causes two main problems:
  - The overhead of token matching is incurred for every instruction. This overhead is especially noticeable in the execution of inherently sequential threads of code and in those cases where the compiler can statically schedule the instructions with only a small loss of parallelism.
  - The dynamic instruction selection tends to mask inherent instruction level locality [Ian88, MNB91]. This is a major drawback of the model because it cannot take advantage of the storage hierarchy.
- Application domains are limited because certain traditional concepts are difficult to represent. For example, critical sections (no interruptions) are necessary to support low-level operating systems and handle certain kinds of resource management, but they cannot easily be represented in a dataflow graph [PT91].

Several designs improve upon the first-generation fine-grain dataflow machines; for example, RMIT [AE90] borrows features of static dataflow to eliminate unnecessary tagging operations, and EPSILON-1 [GDHH89] provides direct matching in order to reduce the matching store resource requirements. However, these are evolutionary improvements that do not directly address the main shortcomings listed above.

In spite of the shortcomings, however, the dataflow model has an important advantage over the von Neumann model in that it addresses the two key multiprocessor issues well: the dataflow model tolerates latency by switching among a set of ready instructions, and it has relatively cheap fine-grain synchronization. The latency tolerance is possible due to the dataflow model's ability to expose maximal parallelism at the function, loop and instruction levels.

1.2 Multithreaded Execution Model

To summarize the discussion so far, the key advantage of the dataflow execution is the ability to provide fine-grain level synchronization (as opposed to higher level synchronization such as barriers) and task-switching at a low cost so that latency can be usefully masked. The key advantage of the von Neumann execution is the ability to exploit locality via storage hierarchy through compile-time, static scheduling. *Multithreading* attempts to combine the two approaches to reap the benefits of each. Some multithreading approaches start with a pure dataflow model, extends it so that the granularity of tasks is increased from one to many instructions, and statically schedules the instructions within each task (or a thread). Threads themselves can be dynamically scheduled as in the fine-grain dataflow model. Thus, the benefits of sequential execution (e.g., taking advantage of locality, reducing overhead, and the ability to write critical sections) can be reaped within a thread and cheap fine-grain synchronization among threads can exploit useful parallelism.

The guiding principle of the above approach, also referred to as a *von Neumann-dataflow hybrid*, is a shifting of responsibility from the hardware to the compiler. By giving low level scheduling decisions to the compiler, the hardware is simplified. However, the compiler has the burden of providing a proper mapping to the hardware so that latency can be masked while at the same time locality is exploited.

The chief feature of multithreading, its ability to cover latency by switching among a set of ready threads, provides another view of multithreading: one that starts with a conventional von Neumann processor and adds the capability of inexpensive context switches. In this view, there exist a number of ready tasks that execute in an interleaved fashion. For example, when a currently executing task makes a remote access, the processor switches to another ready task, saving the current task somewhere. This process requires some hardware support and involves replicating resources, such as register sets.

It is of a paramount importance to multithreading that the cost of thread switching be much smaller than the latency. Only when the cost of thread generation and switching is small can the increased parallelism usefully mask latencies with other computations. It is possible to emulate multithreading on the traditional von Neumann multiprocessor models discussed in the previous section (á la TAM); however, without efficient hardware support the thread switching and creation cost may defeat the usefulness of multithreading.

1.2.1 Variants and Issues in Multithreading

In this section, various approaches to multithreading are classified in several ways.

The Storage Model. Most multithreaded models associate a basic unit of storage (often called a frame) with an activation of a thread or a group of related threads. The frame can contain slots to perform synchronizations, to store input values, and to store temporary values. The allocation of slots in a frame is usually statically determined.

This storage unit can be compared to the traditional von Neumann stack frames associated with function activations; however, whereas the von Neumann calling structure is such that only the top frame is active while the rest lie dormant, the multithreading "calling" structure has the form of a tree, with all the "frames" concurrently active. The inherent ability for a given activation to generate descendent activations all of which may be concurrently active gives rise to the development of a powerful compilation model that can exploit parallelism. However, the challenge for compiler writers is finding a proper mapping from the programming model to this execution model. In most models, a frame does not usually span across multiple processors, although some may allow frames to migrate for a load balancing purpose.

Models such as TAM [CSS$^+$91], *T [NPA92], EM-4 [SYH$^+$89a, SYH$^+$91], and the Iannucci's Hybrid Architecture (IHA) [Ian88] use activation frames that are based on code blocks. A code block is associated with a program structure such as a function or a loop body. For example, when a function is invoked, an activation frame is allocated explicitly as part of the calling convention. In general a code block may consist of several threads; however, a thread is never larger than a code block.

Instead of associating a frame with an activation of a code block, it maybe associated with an activation of a single thread instead. In this storage model, the frame size would typically be much smaller but require more frequent allocation and deallocation operations. The Pebbles model developed in this thesis uses this thread-based frames.

The state of computations is stored in various storage elements including registers, caches, frame memory, and global heap memory. The globally distributed heap space is used for long term storage needs (e.g., data with lifetimes greater than a single frame activation), which typically consist of data structures such as arrays which maybe accessed by several threads located in different processors. The processor state is usually linked to an activation frame. In general, a smaller activation frame increases the amount of synchronizations but decreases the synchronization cost.

Blocking versus Non-blocking Thread Models. Broadly, the proposed multithreaded models can be classified into blocking and non-blocking variants. The classification is based on how thread switching and synchronizations are reconciled. In the blocking thread model, a thread can be suspended (or blocked) and then resumed later, requiring a mechanism to save states and to move threads between the different states. A thread may be suspended when either a remote memory request or a synchronizing operation is made. The hardware complexity and cost may range from simply queueing the suspended thread in a hardware queue to saving an entire set of registers. The size of blocking threads may range from tens to millions of instructions. Examples include HEP [Smi81], Tera MTA [ACC$^+$90], Dash [WG89], IHA [Ian88], and J-Machine [DFK$^+$92].

In the non-blocking thread model, once a thread starts executing, it runs to completion without blocking. This model therefore places two restrictions on each thread: all its inputs need to be present before the execution starts, and all instructions must take a bounded amount of time. The second requirement implies the use of split-phase transactions for long latency operations; for example, the request and its reply represent separate transactions. The non-blocking model requires a simpler processor architecture but the thread size is necessarily smaller than in the blocking model; examples include Monsoon [PC90], *T [NPA92], EM-4 [SYH$^+$89a, SYH$^+$91], TAM [CSS$^+$91], EARTH [HG93, HTG94], S-TAM [Vas94], and Pebbles.

A machine that supports the blocking thread model obviously can support the non-blocking threads, albeit not as efficiently as another machine which is specifically designed to run non-blocking threads. Since the non-blocking threads are smaller and also more numerous, a different hardware requirements exist; for example, a larger queue may be needed to handle a larger number of ready threads. Similarly a non-blocking machine can support blocking threads via a programming method: a set of non-blocking threads can be considered to be one longer blocking thread (e.g., EM-4) with "suspensions" between the boundaries. However, the software must explicitly manage such a "macro-thread." The choice of a thread model also interacts closely with the chosen storage model. Typically, values in registers are not carried across threads. In some cases, notably IHA, registers are not even valid across potentially suspensive points within a thread. Also, the expected lifetime of frames may affect the design of the frame memory. In general, it is expected that the lifetime of a frame is longest for the blocking, code block-based frame model, the shortest for the nonblocking, thread-based frame model, and somewhere in the middle for the other two combinations.

Synchronization. A synchronization requires a meeting point such as a register, a memory word, an interrupt level, etc. The number of these meeting points and their costs greatly affect the ability to tolerate latency through task switching. Most von Neumann machines either provide a limited number of these meeting points (e.g., the registers) or their implementation is expensive (e.g., software-implemented semaphores). In contrast, multithreaded machines provide virtually unlimited meeting points since any memory location can be used as a meeting point. The cost is also low for both explicit and implicit synchronizations that the multithreaded machines provide.

In the implicit case, a synchronization operation is performed as part of a normal instruction execution sequence. A fine-grain dataflow is an extreme example of this: every operation requires an implicit synchronization. Most often, the implicit synchronizations are implemented by encoding synchronization functions in the tag bits associated with each memory word. For example, when an instruction accesses a word which is not yet written, as indicated by its tag bits, the synchronization fails and either the thread terminates or it becomes blocked. Examples include Monsoon, IHA, and Tera.

In the explicit synchronization case, the instruction set contains a set of specific synchronization instructions[1] to perform the operations. Usually, specific memory locations are reserved as meeting points (typically within an activation frame or the heap memory) and the synchronization instructions manipulate their contents. In general, they operate as counters; for example, when a certain number of synchronization events have occurred as indicated by the count, an associated thread can be enabled. Examples include P-RISC, TAM, and *T.

Synchronization involves first associating the "signal" with the appropriate thread, then enabling (or creating) it, and later executing it. The cost of the first two can be reduced by special hardware such as synchronization units and hardware queues. In addition to cheap synchronizations, a cheap dynamic thread creation (forking) mechanism is also important. In general, a thread is created as a result of certain synchronization operations or through a specific call. However, the actual creation could also be handled implicitly or explicitly. In some cases, such as Monsoon, the thread is created automatically as part of the execution of a single instruction, and in other cases, such as P-RISC and IHA, explicit instructions are executed to generate and enable threads.

Scheduling. Given a set of ready threads in the system, an interesting challenge lies in selecting which thread to execute next. The choice could affect the performance greatly and involves several tradeoffs. The scheduling aspect of multithreading has received scant attention so far. This is probably due to the difficulty in implementing a proper scheduling policy since scheduling is controlled locally but has global affects. In general, a global knowledge about the program and system component interaction is required.

Consider an example of such a problem: it may seem sensible to give high priority to a thread that lies along the program's critical path (if that path can be determined) since many other threads may depend it. On the other hand, its execution might disrupt the existing locality so much that little or no overall benefit results. Since optimal scheduling is an NP-complete problem, a set of good heuristics which work well over a range of problems are the best that we can hope for.

[1] In some cases, a synchronization instruction may actually be composed of a sequence of fundamental operations, as is the case in EARTH and TAM.

Models	Storage	Thread	Synch.	Forking	Scheduling
Monsoon	Code block	Nonblocking	Implicit	Implicit	FIFO
IHA	Code block	Blocking	Implicit	Explicit	FIFO
TAM	Code block	Nonblocking	Explicit	Explicit	Quantum
EM-4	Function	Nonblocking	Implicit	Implicit	FIFO
P-RISC	Code block	Nonblocking	Explicit	Explicit	FIFO
J-Machine	Code block	Blocking	Implicit	Explicit	FIFO
*T	Code block	Nonblocking	Explicit	Explicit	FIFO
Tera	Function	Blocking	Implicit	Explicit	FIFO
EARTH	Function	Nonblocking	Explicit	Explicit	FIFO
Pebbles	Thread	Nonblocking	Implicit	Implicit	FIFO/Depth-first

Table 1.1: Classification of Proposed or Existing Models

A related difficult question is when to do scheduling: at run-time, compile-time, or a combination of the two. It would seem that a combination of the compile-time and run-time scheduling makes the most sense but much research needs to be done.

Most multithreaded models introduced so far rely on strict FIFO run-time scheduling. Examples include *T, Monsoon, J-Machine, and Tera. Some models do use a simple priority-based scheme. For example, TAM tries to schedule together threads that belong to the same activation frame in order to exploit locality. Even for this fairly simple local scheduling policy, the implementation cost maybe such that a simple FIFO scheduling policy might end up being better [NWD93].

Implementation. In Table 1.1, various proposed and actual models are classified according to the above criteria. Just about every design alternative discussed can be implemented in either hardware, software, or some combination of both. The tradeoff lies mainly in implementation cost versus speed. However, there should be some optimal price/performance point and this is the subject of current research. For example, EARTH uses dedicated hardware to perform synchronization, whereas *T uses the main execution processor for the duty. As another example, some have hardware scheduling queues (e.g., *T) whereas others implement queues via software schemes (e.g., TAM). Each tradeoff may affect the goals and strategies of the code generation significantly, by providing different cost figures.

In Figure 1.2, the variations in the multithreading model are (roughly) mapped into the design space. Along the x axis lie the two thread models: blocking versus nonblocking. Along the y axis, the models are classified according to the cost of doing various multithreading operations such as synchronization and thread creation. As more hardware support is provided, the run-time cost decreases.

1.3 Compiling for Multithreaded Models

Compiling for a multithreaded processor is a challenge especially since the multithreaded processor affords a wide range of design parameters as described in the previous section. The goal is to generate code that can make effective use of the resources of a given multithreaded machine.

The most significant aspect of code generation for any multiprocessor involves proper partitioning of programs so that the hardware overhead for operations such as synchronization is minimized and the ability to exploit parallelism is maximized. Partitioning determines the unit of computations (tasks or threads) in a program. For a given multiprocessor architecture, there should be some optimal granularity. Generally, the partitioning process generates threads during compile time and the scheduling process determines how to schedule these threads at either run-time or compile-time.

Several methods that generate multithreaded code have been developed by various researchers [GN88, EG91, Ian88, SCvE91, HDGS92, Tra91, RNB93]. These methods generally fall into two categories. In the *top-down* strategy, codes are generated directly from program data-dependence graphs. In the *bottom-up* strategy, fine-grain dataflow graphs are first generated, which are then partitioned into a set of threads. The

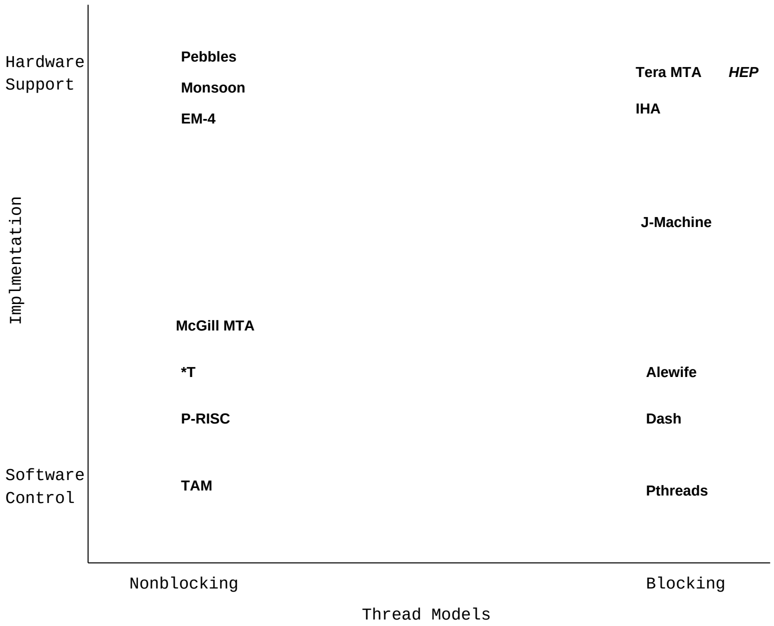


Figure 1.2: Mapping of Existing Multithreaded Models

top-down method is traditionally used to compile programs for traditional multiprocessors [CF89]; the usual goal is to minimize the overhead by making tasks fairly large but small enough so that some parallelism could fruitfully be used. Loop slices usually represent the minimum task size. In contrast, the bottom-up method is usually used to generate code for the hybrid or multithreaded machines. Here, the thread size is usually small, so the stated goal of most of these methods is to create as large a thread as possible, as long as it satisfies certain conditions.

1.4 Overview of Thesis

The objective of this thesis is not only to present another new multithreaded model but also to examine what features are useful and what issues are important in the design of multithreaded processors and code generators. The overriding goal is to identify what can be done to improve the performance through either hardware or software techniques. Therefore, this thesis focuses on the study of "what-ifs" by implementing different techniques and then evaluating them.

In Chapter 2, background materials on the existing multithreaded models and code generation schemes are surveyed. In Chapter 3, the research on code generation for the von Neumann/dataflow hybrid execution model is presented. Here, different bottom-up techniques are applied in order to generate threads or "clusters" of instructions from fine-grain dataflow graphs. The main challenges in partitioning include minimizing the synchronization cost, masking the latency, and exploiting instruction level locality. Along with code generation, the research focuses on the quantitative evaluations of cluster executions and how they compare to the fine-grain dataflow executions.

The pure bottom up thread generation suffers from the lack of knowledge of program structures, thereby limiting the thread size. In Chapter 4, another thread generation scheme is evaluated: instead of starting from the dataflow graph, code is directly generated from the program data-dependence graph and then further optimized via global, bottom-up techniques. The hope is to create larger threads by taking advantage of the program's semantic information. Along with hybrid code generation, a baseline multithreaded execution model called Pebbles and a parallel machine language called MIDC is developed. They serve as a compilation target and are used to further evaluate issues in multithreading.

In building truly general-purpose multiprocessors, there are many issues that must be tackled besides the locality, latency, and synchronization issues discussed previously. All of these issues must be properly balanced; for example, extensive multithreading in order to cover latency runs counter to the underlying storage hierarchy [CSvE93]. Some of the important issues that multithreaded models must deal with include the storage hierarchy, scheduling, parallelism management, resource management, and communication issues. Chapter 5 analyzes these issues and, based upon the analysis, various techniques that reduce overhead and improve overall performance are presented:

- Due to the sequential execution of threads, the locality of accesses *within* an executing thread can be exploited using registers and caches. At the *inter-thread* level, however, the locality of accesses is not yet well understood. An analysis of inter-thread locality and the design of the storage hierarchy that exploits this locality are therefore studied and presented in Section 5.1.
- In Section 5.2, inter-processor communication is evaluated and the techniques to reduce communication traffic are studied. Issues of particular interests include different sources of message generation and the locality of these messages. The techniques include grouping related threads to the same processor in order to confine communications between related threads within a processor, and "bundling" of several short messages into one longer message in order to reduce the message processing overhead.
- Multithreading relies on program parallelism in order to mask latency. In some cases, it may be desirable to control the parallelism in an explicit manner. Moreover, some forms of parallelism may be further exploited in a way that improves the run-time performance. These issues are studied in Section 5.3.

Chapter 2

BACKGROUND

In this chapter, several notable von-Neumann/dataflow hybrid and multithreaded architectures are first surveyed. Then, approaches to generating code for these architectures are examined. Finally, some important issues that must be addressed in multithreading are examined.

2.1 A Survey of Multithreaded Architectures

There exist a growing number of von-Neumann/dataflow hybrid and multithreaded machines that have been either built or proposed. These machines are mostly distinguished by their processor architectures rather than by their interconnection schemes.

2.1.1 Basic Terminology and Concepts

The following definitions and concepts are common to several models discussed later.

A *continuation* identifies a point in computation. It usually consists of a pointer to the instruction (IP) and a pointer to its associated activation frame (FP). A continuation can be associated with a thread or with just the next instruction to be executed; the essential quality is that it contains all context information for that computation.

A *token* contains a continuation and zero or more values. Most network traffic is messages consisting of just tokens. In addition, most transactions across the network are split-phase (Tera is a notable exception).

I-structures [AT80, ANP89] implement non-strict arrays. Each memory cell in an I-structure memory has an associated presence bit used to tag whether the memory cell contains a valid value or not. This mechanism allows synchronizations to be performed at the array element level, rather than at the array level, by allowing access to an array element before the entire array is constructed. Structure elements are accessed in a split-phase fashion: a processor sends a request for an array element, and if the requested element value is present, it is returned in a separate transaction. A read request to an absent element is deferred until the value is stored. A strict split-phase operation for every access to I-structures may be unnecessarily general; in many cases, compile time analysis can determine the order of production and consumption of array elements.

Active Messages [vECGS92] is a communication mechanism which can efficiently integrate messages into ongoing computations. Basically, the head of each message contains the address of a specific message handler. Rather than using a generic message handler that accepts any type of message and decodes it, an active message handler is designed to handle one particular type of message. Hence the active message handler knows the layout of the message and how to respond. Overall, the handler extracts the message payload from the network and either stores the data into preallocated storage or responds to the message directly. The key is that no buffering takes place.

2.1.2 Monsoon

Monsoon [Pap88, PC90, PT91] was originally developed as an outgrowth of the MIT Tagged-Token Dataflow Architecture (TTDA) [ABM87] with explicit token storage; later, the thread-running capability was added.

The process involved in the initiation of each instruction is similar to fine-grain dataflow: a synchronization is first performed. If it is successful, the operation corresponding to the instruction is performed and a result can be sent to one or two destinations. The synchronization operations include various types of semantics not available in a pure dataflow machine; each frame slot has programmable presence bits specifying

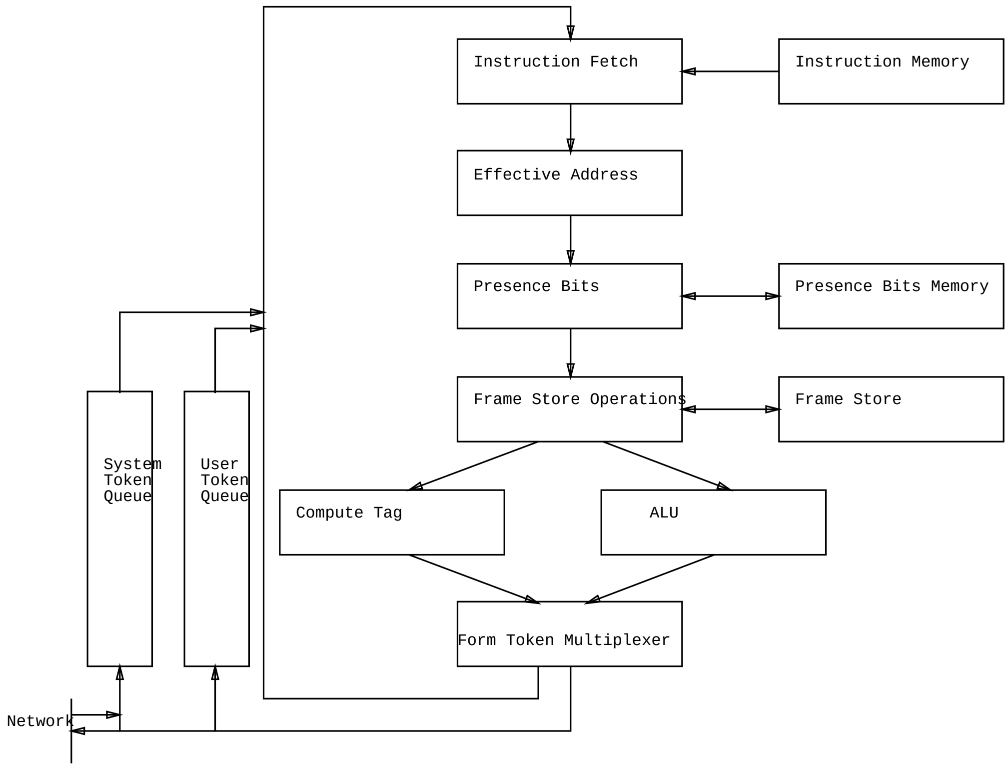


Figure 2.1: Monsoon Processor

the state of the frame slot. A common operation works as follows: if the slot is empty, the value is stored, otherwise the value in the slot is extracted and the corresponding operation performed. In addition, the machine may support loop constants, I-structures, and accumulators. Simple von Neumann-style imperative threads can be supported by simply ignoring the presence bits and sending a single result to the immediate successor instruction. Notice that each token arrival triggers the retrieval of the instruction that resides in the instruction memory which may be aborted in the middle of the pipeline.

Monsoon consists of a number of pipelined processors and I-structure modules connected via a multi-stage network. The architecture of a single processor is shown in Figure 2.1. Each processor has an eight stage pipeline; a token may enter the pipeline in each cycle and zero to two tokens may emerge out of the pipeline. A token value is 64 bits and may be an integer, double, structure store pointer, bit field/boolean or a tag. A tag is also 64 bits and is of the form PE:FP.IP where PE specifies the processing element. Both the tag and value contain an additional 8 bits of type information. Altogether, a token contains 144 bits. All activation frame references are local. A data structure pointer specifies either a PE or an I-structure module, and a local address within it. In addition, it specifies interleave information that tells how a data structure is spread over several PE/I-structure modules.

There is a facility that allows one result token to be immediately fed back to the top of the pipeline; this is used to implement a running thread. At any time, a given thread occupies one stage of the pipeline. Hence, the pipeline depth of eight provides up to 8 independent threads that can be executing in an interleaved fashion; hardware allows up to 32,000 threads that are ready to execute. The state of a thread is captured by the *computation descriptor* which has a number of registers including the continuation register C, the value register V, and three temporary registers. To be able to make use of temporary registers, recirculating token paths of the Monsoon are used to make sure that the thread is not switched out. The compiler can make sure this happens through the control of successor instructions.

Even though there are three temporary registers, the instruction set resembles an accumulator-based machine instruction set with V as the accumulator. For example, the following expression

```
e := a + b;
```

might be compiled into the following sequential thread:

```
MOV    [FP+a], V
FADD   V, [FP+b]
MOV    V, [FP+e]
```

The result of the add instruction is implicitly stored in the V register, which is actually carried in a token. In a thread, the instruction sequencing is automatically sequential. The sequence can be altered by regular RISC control flow operators or by forks and joins. The `fork` instruction enables a new thread in addition to the current thread. In actual compiled code, the fork instruction is combined with the predecessor instruction since any instruction can perform a fork in addition to a regular operation. The `join` is also handled implicitly through the Monsoon matching mechanism. There is a `stop` instruction which ends the current thread. The `start` instruction and its derivatives are used to enter a new thread in a different activation. These are used, for example, in procedure calls.

Evaluation. An 8-processor Monsoon have been built running Id programs. Compared to the original dataflow incarnation, Monsoon's performance is improved through the use of a small number of registers and by taking advantage of 8-fold parallelism available through the pipeline. There must be enough program parallelism, however, to keep each processor's pipeline full.

In general, it takes more Monsoon instructions than 3-address von Neumann instructions to perform a task due to Monsoon's accumulator-based model. Also, the relatively small number of registers limits the amount of locality that can be exploited. Another problem is the lack of a memory hierarchy. In essence, the entire frame store lies on the same level as the registers, making it necessary to slow down the processor clock cycle.

Another drawback of this model is the fact that the pipeline is inefficiently utilized since most instructions do not use all the stages. For example, a synchronization need not be performed for every instruction but it occupies two stages of the pipeline. Also, each join operation causes a pipeline bubble since the first arrival causes the corresponding instruction to be aborted.

2.1.3 Iannucci's Hybrid Architecture (IHA)

In contrast to Monsoon which starts with a dataflow paradigm and adds support for static scheduling, IHA [Ian88] is based on a von Neumann paradigm with support for dynamic scheduling and fine-grain synchronization. Code for this architecture is generated by partitioning dataflow graphs into a set of Scheduling Quanta (SQ). The compiler schedules instructions within a SQ, and determines the dependencies among SQ's. IHA is based on the code block storage model.

The architecture consists of an array of processors and I-structure modules connected via an interconnection network. Each processor has a pipelined datapath, local memory, and a set of registers. The processor architecture is shown in Figure 2.2. Each frame location has presence bits used for synchronization.

Each processor contains several queues of continuations corresponding to different states: ready, running, blocked, etc. When a thread requests a remote memory access, it becomes blocked. When the thread blocks, its continuation is stored in a frame slot and the hardware then picks the next available thread (continuation) from the ready queue and starts executing it. When reading from a frame location, a thread blocks if the location is empty, thus providing an implicit synchronization. The instruction set also provides means to synchronize explicitly; for example, a new thread can be explicitly forked. A binary operation can read from two frame slots, either of which can cause a suspension. An IHA instruction is lower powered than a dataflow instruction. It uses an explicit instruction to fork a new thread.

Registers may be used to improve efficiency; however, they are only valid from the time a thread starts until it blocks or terminates. This means the compiler must know where possible suspensions can occur and must avoid using registers across them.

Evaluation. Benchmark results indicate that the average number of instructions executed between suspensions ranges from 2.4 to 6.5. Also, dynamic instruction counts between the TTDA and hybrid model seem to be about the same, indicating that performing synchronizations only when necessary makes up for

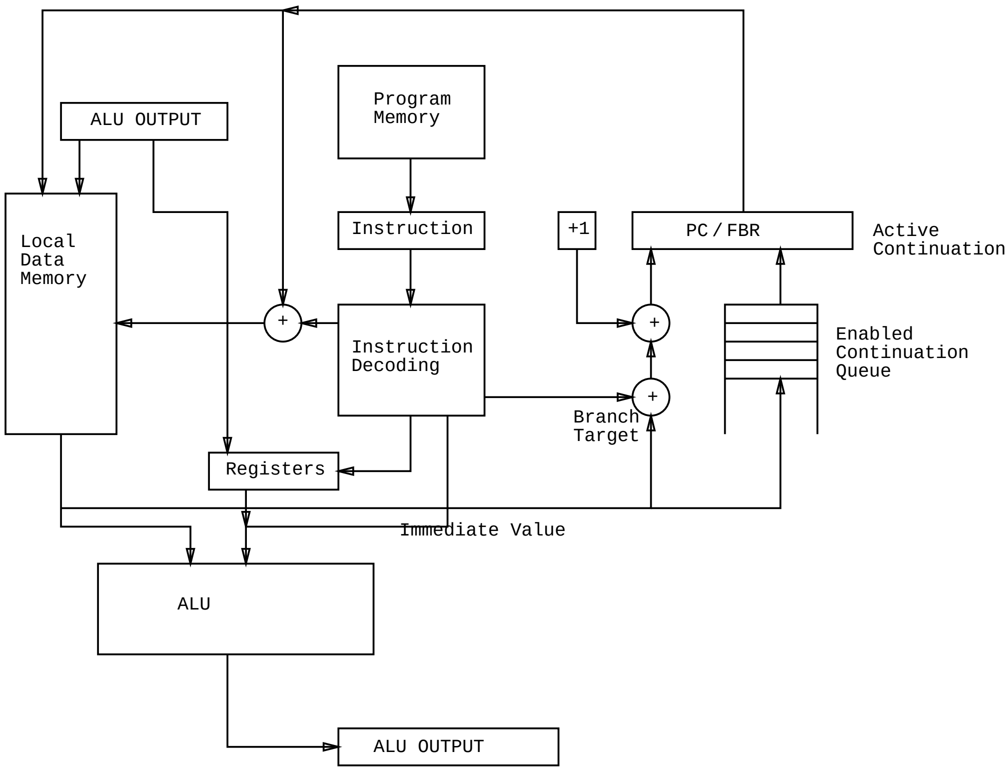


Figure 2.2: Iannucci's Hybrid Processor

the lower powered hybrid instructions. Overall, the hybrid model seems to tolerate latency well. Possibly the biggest criticism of the hybrid model is that its instruction set is not load/store. Each instruction can read from up to two frame locations and perform an operation. The use of multiported cache can reduce the latency and improve the throughput; however, cache access is still more costly than accessing the registers.

2.1.4 TAM

The Threaded Abstract Machine (TAM) [SCvE91, CSS+91] is an execution model which exposes synchronization and thread scheduling to the compiler. It provides a low-level compilation framework, consisting of TAM primitives, for compiling parallel languages such as Id. The TAM primitives can then be targeted to conventional von Neumann parallel machines. TAM provides primitives to send and receive messages, switch threads, do remote memory reads, etc.

In order to reduce the cost of software synchronization, TAM uses several different forms of synchronization which vary in cost. It tries to use the cheapest form of synchronization whenever possible by taking advantage of the underlying storage hierarchy and the locality present in programs. TAM uses active messages to reduce the overhead involved in network interfacing. It uses *inlets* which can directly deliver a specific message from the network to the place it is needed.

TAM uses a code block-based storage model. A frame slot can be used as a caller argument depository, a synchronizing point or a local variable. Each frame also has space to hold a list of enabled threads within that frame activation. Structures are heap allocated data objects with I-structure semantics.

TAM uses three-address instructions where operands can refer to registers or to frame slots. Threads are non-blocking and may be either synchronizing or non-synchronizing. A synchronizing thread uses a frame slot as a barrier. A thread can perform a conditional fork of another thread by decrementing a counter stored in the frame slot. When the counter reaches zero, the corresponding thread is enabled. An enabled thread is identified by a continuation (i.e., the instruction pointer and the frame pointer). This explicit synchronization approach is in contrast to both Monsoon and IHA, which use presence bits. In IHA a thread blocks when the slot is empty; in Monsoon, an instruction in a thread is aborted when reading an empty slot.

Active message handlers called *inlets* provide the external interface to the frame and are the means of communication with other frames. The inlets receive the argument values for the activation and are the targets of I-structure fetches. An inlet is a thread consisting of several instructions which decompose messages, store them in the frame slots, and enable threads.

Given a tree of activations as in Monsoon and IHA, TAM recognizes the fact that not all activations may reside at the top level of storage hierarchy. Therefore, this requires the ability to move activations between storage levels. The storage hierarchy is made visible to the TAM compiler, which generates code that explicitly move certain activations up and down the storage hierarchy. TAM's scheduling discipline reflects this storage hierarchy. An activation must be *resident* at the top level of the hierarchy in order to be executed. Once a thread in a resident activation starts executing, it has access to registers and TAM continues to schedule all enabled threads from that activation until there are no more enabled threads. A *quantum* represents the set of threads executed during one such interval. At the start and end of a quantum, an initialization and a completion thread associated with the code block are executed, respectively. The initialization thread sets up several things, including resetting counters for synchronization purposes. The completion thread grabs an enabled activation from the pool, makes it resident, and starts it. Processor registers may have a lifetime of a single quantum. However, register allocation can be much more difficult since a quantum is dynamic: the set of threads in an interval varies on different activations of the same code block. Register save and restore are most likely to be necessary for inter-thread register allocation.

Evaluation. Since thread-to-thread and activation-to-activation transitions are explicit, TAM executes more instructions than do implicitly synchronizing models. For example, the dynamic instruction count is roughly 50% larger than the TTDA. The average thread length is 3-4 instructions. Due to the fairly small thread size, the intra-thread locality does not provide much opportunity for exploiting locality. At the quantum level, which typically has around 10 threads, locality can be exploited via registers. However, the dynamic quantum boundaries make it difficult for the compiler to perform register allocation across threads.

Although TAM exposes and provides several forms of synchronization to the compiler, the cost nevertheless can be quite high. In fact, the results reported for the CM-5 implementation of TAM [CGSvE93] indicate that half to three-quarters of the execution time is spent in processing messages, and another 10 to 30 percent of the time is spent scheduling threads. Overall, a significant amount of processor time is spent performing non-compute activities. In many ways this is due to a large number of synchronization activities that cannot use the cheapest form. Although the overall performance results indicate that TAM is scalable, its total reliance on software scheduling and synchronization may become a bottleneck in achieving high performance.

2.1.5 EM-4

The design of EM-4 [SYH+89b, SKS+92] is derived from the SIGMA-1 dataflow processor [YSHK85] with several major improvements. It is designed to be highly scalable with more than a thousand nodes. Each node contains a single-chip processor called EMC-R and some local memory; nodes are connected via Omega network. The EMC-R integrates a matching unit, a network interface, and an execution unit organized in a pipelined fashion. The processor architecture uses strongly-connected blocks (SCB) to implement static scheduling and uses direct matching to minimize the matching cost. There are no specific structure store modules.

EM-4 can execute a form of dataflow graph in which arcs are classified into *strongly-connected* and normal arcs. The strongly-connected arcs dictate static scheduling in that a processor executes nodes connected via these arcs successively without matching. A set of nodes connected via strongly-connected arcs form a SCB. Hence, a SCB is likened to a nonblocking thread; once started, it executes to completion. A SCB may also take advantage of registers.

As in Monsoon, tokens are the primary means of communication. As tokens arrive from the network, they are queued in the input buffer. The matching unit takes one token at a time and processes it. Matching is implemented using ordinary memories and simple logic. For each function invocation, an instance of storage is allocated to a processing node called an operand segment. The operand segment is used to store the incoming operands for the matching and for temporary storage. Matching is performed by checking for the stored data in the match location and storing the new data if its partner is absent. If the partner is already present, the corresponding SCB is enabled. The template segment contains the code for the function. At the top of the operand segment, the beginning address of the corresponding template segment is stored. There is a one-to-one correspondence between the operand segment and the template segment so that the match address uniquely identifies the SCB to invoke upon a match. Therefore tokens contain only an operand segment address and not an instruction pointer as in Monsoon.

Tokens also contain a type field which can encode a number of different special tokens aside from the regular token. There exists a corresponding SCB for each type of special tokens. Unlike TAM, where each message contains the address of the handler, that address is implicit in the type field of a special token. The operand segment address field of the token can then hold a second value. These tokens can be used to implement I-structure emulations, remote memory reads, etc.

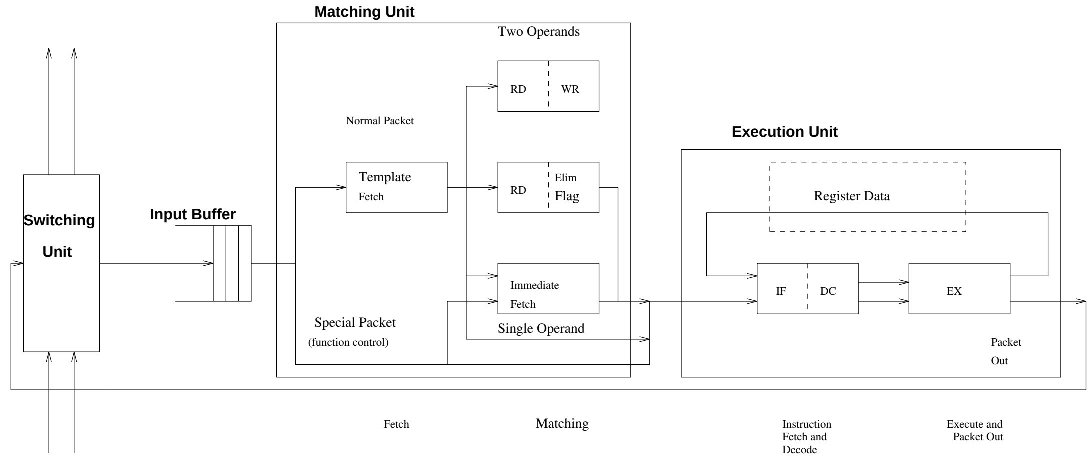


Figure 2.3: The EMC-R pipeline organization

The pipeline organization of a processing node is depicted in Figure 2.3. In addition to the main circular pipeline that includes the matching unit, a smaller circular pipeline going just around the ALU unit is used to implement SCB execution. A potential bottleneck arises because different functional units of EMC-R share a single port to the local memory. A prototype machine containing 80 processing nodes on 16 processor boards has been constructed for performance evaluation.

Evaluation. The SCB is based on dataflow partitioning; a node and its direct descendents are collected into a SCB. This means that a SCB cannot have more than two inputs, thereby limiting the potential sizes of SCB's. The requirement for one-to-one correspondence between the operand segment and the template segment may lead to wasted storage space. Most likely there exist holes in the operand segment which do not serve any purpose.

2.1.6 P-RISC

P-RISC [NA89, Nik93] is an abstract model based on the idea of adding dataflow features to conventional RISC processing, somewhat similar to IHA. A crucial difference with IHA is the use of explicit synchronizing instructions rather than implicit synchronizations that might occur with executing regular instructions in IHA. Hence, there are no presence bits associated with each memory location. Instead, specific locations are reserved for counter-based synchronizations. In this respect, P-RISC is similar to TAM.

It uses a message queue which may be either implemented in hardware or managed in software using local memory. When a message arrives at a particular node, a general interrupt handler moves the message into the message queue. The head of each message points to the entry thread. When invoked, it moves the payload into registers first and later stores it in frame locations (in some cases, the latter step may be skipped). During execution, a thread may fork other threads. The continuation of each forked thread

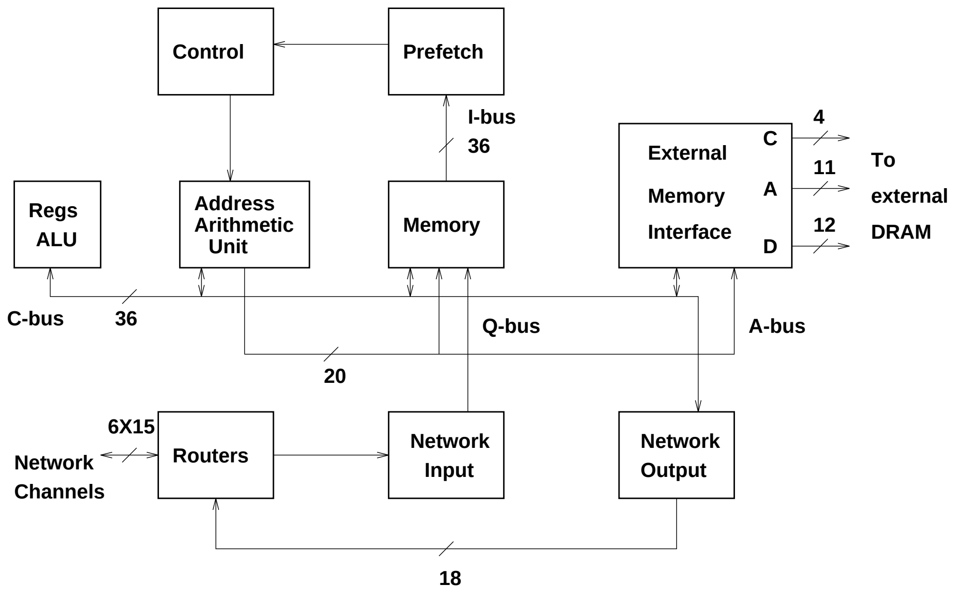


Figure 2.4: The J-Machine Message-Driven Processor

is stored in the scheduling stack. As each thread finishes, it pops the next continuation off the stack and executes it. When there are no more forked threads to execute, it grabs the next message off the message queue and starts the whole process again. The execution is therefore data-driven. It is assumed that the number of threads executed (and therefore the number of instructions executed) is limited by the amount of state carried by the incoming message. Preliminary results indicate between 20-50 instructions executed per incoming message.

2.1.7 J-Machine

The J-Machine [DFK+92, NWD93] (Jellybean Machine) is based on a processor building block consisting of a small processor, an integrated network interface, and a relatively small amount of memory. The idea is that the entire building block can be implemented cheaply, so large-scale systems can be with a good cost/performance ratio. The processing nodes, each containing a Message-Driven Processor (MDP) and 1 Mbyte of DRAM, are interconnected in a k-ary 3-cube topology. The J-Machine uses deterministic dimensional-ordered, wormhole routing with two-way channels. The key feature of the J-Machine is an integrated network interface that provides low cost message handling and a low latency, high throughput network. The hardware support is provided for end-to-end message delivery including message formatting, injection, buffer allocation and task creation. Messages are injected into the network using a series of SEND instructions at a rate of up to 2 words per cycle. The mechanism therefore efficiently supports threads that communicate frequently using small messages.

Figure 2.4 shows a block diagram of the MDP which includes a CPU, a small on-chip memory, and a network router/interface. When a message arrives at a node, the hardware queues the message in one of two queues corresponding to two priority levels. Each message is processed by dispatching a task that takes four cycles. The address of the task is stored at the head of each message (i.e., active message based).

In addition to the usual set of conventional microprocessor instructions, the MDP instructions also provide support for communications, synchronizations, and naming. There are three register sets: two for each priority level and a third to run a background task whenever the other two queues are empty. A register set includes four data registers and four address registers. A higher priority message can interrupt the processing of a lower priority message.

Each data word contains 4 presence bits which can encode a variety of synchronization functions, including futures. When attempting to read a slot before its value has been written, a processor exception occurs. A system routine would then suspends the thread until the value is written. The hardware uses a segmented memory system where virtual addresses of global objects are mapped to a particular memory

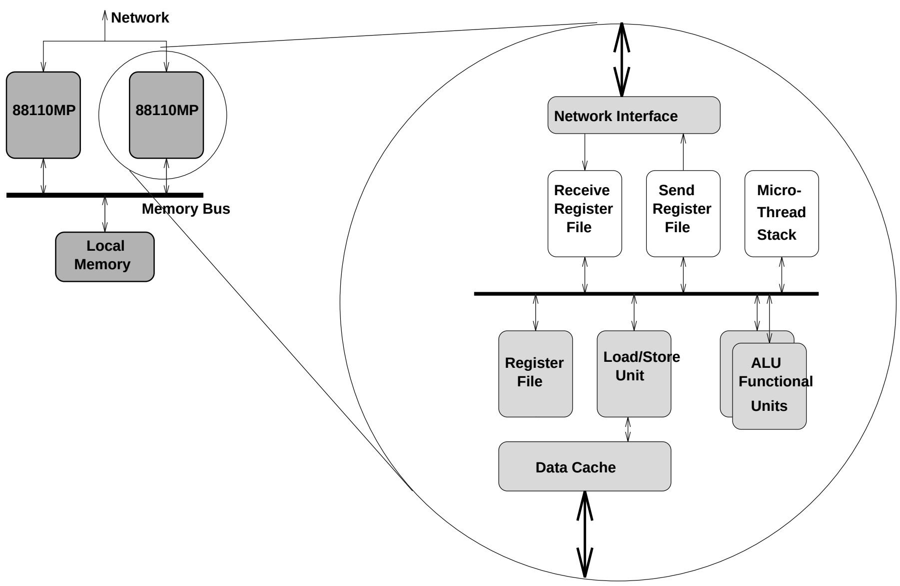


Figure 2.5: The MIT/Motorola *T Node

segment of a processing node. This mapping is handled using explicit instructions with the help of a hardware name-translation table. Before a global object can be accessed, the name-translation table must first be consulted to get its physical address.

Evaluation. The bandwidth of the local memory (DRAMs) and the network are mismatched since the network can deliver data at a rate three times the DRAM bandwidth rate. Also, a considerable amount of time can be spent explicitly translating and managing the global names. Another drawback is the machine's relatively small register set which limits the number of single-cycle operand accesses.

Currently, a prototype 1024-node J-Machine is operational at MIT. The per message overhead is up to two orders of magnitude smaller than compared to other traditional machines such as CM-5 and Delta.

2.1.8 *T

The *T project is a joint effort between MIT and Motorola. Although it represents a successor to the Monsoon project, *T is quite different from Monsoon. Rather, it builds on the ideas behind P-RISC and TAM to provide fairly cheap, explicit thread creation and synchronization using a conventional microprocessor enhanced with message handling capability.

The processors are interconnected by a fat-tree interconnection network. The network operates with two priority levels for the user and operating system programs. The network switch is a cut-through router with intelligent buffering which allows bypassing of blocked messages. The router uses store-and-forward during high network utilization, and uses virtual cut-through at other times. *T also has a high performance I/O system with two independent HIPPI channels. The network provides some protection mechanisms against errant user-mode control of the network (e.g., the use of process id for each message).

The network communication is based on active messages with each message consisting of a continuation and one or more values. All transactions are split phase and the network interface is well integrated with the processor pipeline. There are no specific I-structure modules; instead, global heap memory is distributed among the nodes.

The logical model of a *T node consists of three RISC processors:

- The data processor dP runs the computation threads. It obtains a thread to run from the continuation queue and starts executing it. Each thread runs to completion without blocking. Registers do not live across threads.

- The synchronization coprocessor sP runs short synchronization threads. It receives load responses or synchronization messages and executes the corresponding handler thread. The handler thread may load message arguments into the frame slots and may enable a thread by pushing its continuation into the continuation queue so that dP can process it. The handler thread functions similar to the TAM's inlets.
- Remote memory request coprocessor rMEM is very similar to sP and is dedicated to serving incoming memory retrieval and storage from the network.

The actual design multiplexes the functions of dP and sP onto a single customized Motorola 88110 processor (88110MP) which has a network message interface and synchronization unit (MSU) built into the chip. The MSU unit contains a set of message transmit registers, a set of receive registers, and a microstack that stores thread continuations. In addition, there is another set of instructions to manipulate the MSU unit. The implementation takes advantage of the dual-issue capability of 88110 and a wide datapath between the message registers and data registers. For example, eight words can be moved from the data registers into the transmit registers in a single cycle. A second 88110MP implements the rMEM function. Figure 2.5 shows the diagram of a single node.

Instead of polling, the network receiver can be configured so that interrupts occur when a message arrives. For a long computation thread, polling instructions should be embedded in the middle to ensure that the network is not clogged with un-delivered messages. In this case, there might be a need to save and restore contexts.

A computation thread (a microthread in *T terminology) is created by *fork* which pushes its continuation onto the microthread stack. The last instruction of each thread is *sched*, which schedules the next ready thread. The *sched* instruction gets the next thread to execute either from a waiting message or the top of the microstack, and stores the continuation into a register. The *jmp* instruction then jumps to it. Therefore, message handlers have the priority of over the computation threads.

Evaluation. *T supports low cost message transmission and handling that is tightly integrated with a superscalar processor core. It can format and send a simple message in six cycles which compares well with the J-Machine. However, the code must explicitly test to make sure the message is indeed accepted by the network interface. It might retry sending a few times before trapping to a software routine that saves the message into the local memory to be retried at a later time.

Forking a thread takes only a few cycles. Synchronization is performed via the use of synchronization counters managed in software. It uses *cpost* which checks the counter and conditionally posts a new microthread. rMEM implements I-structure semantics in software since the memory does not contain presence bits. The scheduling behavior is more or less fixed due to the use of the stack, though it is possible to control it somewhat at a cost.

The 32-entry microstack might be too small, causing excessive overflows and requiring software trap handling. Unfortunately, no performance data is available to evaluate the model on many issues including the stack overflow.

At the time of this writing, *T has been scrapped in favor of development of its successor called *T-NG. *T-NG incorporates global caches and uses unmodified PowerPC 620 instead of 88110MP. Since thread creation and synchronization is not as efficient as *T, global caches are used to help reduce the need to task switch as often.

2.1.9 Tera MTA

The Tera MTA system [ACC+90, ACC+91] is a descendent of the Horizon and HEP multiprocessors. It uses multi-stream processors and memory units interconnected via a modified 3D mesh network. The network provides sufficient bandwidth to allow each processor to send and receive a small packet in each network cycle. The network is designed so that only some of the network nodes contain resources such as processing nodes, memory units, cache units, and I/O processors. The rest of the network nodes provide bandwidth and routing capability. For example, a 256 processor system has 4096 network nodes, of which 1280 have resources attached. The network is structured so that in alternating layers along the z-dimension, either the x-dimension or y-dimension links are missing.

Each processor has hardware support for 128 threads. Each thread's context is represented by a set of general-purpose registers, a set of target registers, and a status word containing the instruction pointer and status bits. There are 128 of these hardware contexts allowing rapid context switching. The target registers are used as branch targets, thus separating the branch condition test from the target address generation. It interleaves these threads so that a context switch can occur every cycle. The system can guarantee that the results of arithmetic and conditional operations are available to the next instruction in the thread through this interleaving. Each instruction can pack three operations: a memory reference, an ALU operation, and either a flow-control operation or a simple ALU operation to perform an address calculation.

Assuming an average memory latency of 70 clock cycles, 70 threads are needed in each processor in order to mask the latency. Instead of interleaving at each instruction, consecutive instructions from the same thread can be issued by using an explicit lookahead field embedded in each instruction. The 3-bit lookahead tells the minimum distance between the current instruction and the instruction that depends on it. With a 3-bit field, the maximum distance is 8 instructions; therefore, at most eight instructions can be issued consecutively from a single thread.

Tera MTA supports multiple processes through the use of protection domains. Each thread is associated with a single protection domain. There are 16 protection domains allowing up to 15 user tasks to be executing concurrently without interference from each other.

Each memory word has four tag bits comprising two data trap bits, a full/empty bit, and a forward bit. When a thread accesses a word with a data trap bit set, a processor exception occurs. The data trap bits can be used to implement breakpoints, for example. The forward bit is used to implement an indirect access: when the forward bit is set, the word is interpreted as a pointer to another word and the hardware automatically dereferences it. The full/empty bit can be used to implement *futures* and synchronized variables. When a *future* word to be read is empty, the thread busy-waits until the word is written. The hardware can be set so that after a certain number of retries, an exception occurs allowing a software routine to save the thread in a defer queue to be tried later. The synchronized variables function like semaphores: when a full word is read, it is reset to empty thus allowing mutual exclusions.

Evaluation. The hardware limitation of 128 threads poses a significant restriction that the compiler must handle, i.e., the compiler must be careful not to create more threads than there are available hardware contexts, so a significant amount of time may be spent in resource management. Another performance hit is caused by busy-waiting when reading an empty word. A study, however, is needed to find the frequency and the cost of busy-waiting in a real application.

In the Tera MTA system, the amount of parallelism required to keep the processors busy can be significant: up to 17920 (256x70) threads are necessary to keep all 256 processors busy, although in practice the use of lookahead fields reduces this figure. Hence a heavy burden falls on the compiler to extract enough parallelism. Even when you have enough parallelism, the interleaving of threads implies a loss of locality (causing longer memory access latency) and results in a need to have even more threads to mask the memory access latency. During periods of limited parallelism and single thread executions, performance may not fare well. Overall, Tera claims a peak performance of 256 Gigaflops on a 256-node system.

2.1.10 McGill/Concordia EARTH

A primary feature of the EARTH model [HG93, HTG94] (formerly known as MTA) being developed at McGill and Concordia is the use of an off-the-shelf microprocessor as an execution processor. Each node contains a microprocessor and an external synchronization unit that performs the synchronizations and interfaces to the network.

Unlike *T, the synchronization unit in EARTH is not user-programmable. The synchronization unit (SU) and the execution unit (EU) are linked together via queues. Figure 2.6 shows the architectural model of a node. Both SU and EU share access to the local memory via the same cache. EU retrieves the thread continuations from the ready queue. SU puts thread continuations into the ready queue but EU cannot directly write to the ready queue. EU communicates to SU via the event queue. The event queue also gets messages from the network.

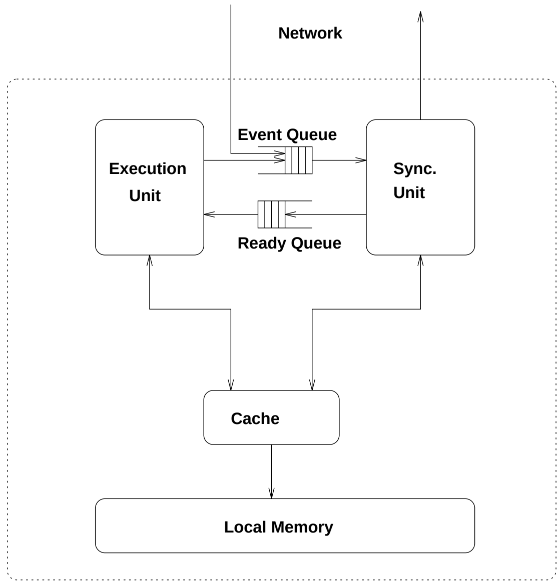


Figure 2.6: McGill/Concordia EARTH

An activation frame is associated only with a function and not a loop, thus limiting the amount of parallelism that can be exploited. Each frame contains synchronization slots, local variables, and the linkage information to its caller.

When a function is invoked, the first thread of that function is automatically enabled. In general, the execution unit may enable a thread by posting either a *spawn* or *sync* message to the event queue. The SU processes *spawn* by posting the indicated thread in the ready queue. It processes *sync* by decrementing a synchronization slot. Each slot contains three pieces of information: a current count, the reset count, and the pointer to the thread to be enabled when the count reaches zero. If the synchronization slot value reaches zero, the count is reset and the corresponding thread is posted to the ready queue.

The execution unit may enable a remote thread or perform a remote synchronization by posting either a *rspawn* or *rsync* message. When such a request is seen, the SU packages an appropriate message and sends it out to the network. The remote SU receives the message and performs the required action.

The execution unit fetches the next thread from the ready queue by executing *fetch_next*. Since existing microprocessors do not contain the above native EARTH instructions, each EARTH instruction is replaced with a macro made up of microprocessors' native instructions.

This model proposes an external *Register Use Cache* in order to take advantage of the register windows in some microprocessors. This external cache maintains mappings between register sets and frames. The synchronization unit can take advantage of this cache in the following way: when a ready thread is about to be posted, the cache is searched associatively to a find a match. If the thread's activation frame is already in the register windows (i.e., there's a match), then the thread is given a higher priority. Perhaps the highest priority can be given to a thread whose frame is the same as the currently executing thread. However, the exact mechanism of how this prioritizing is accomplished is still unclear at this point.

EARTH's developers have created an emulation model running on top of the MANNA multiprocessor. MANNA contains sixteen nodes interconnected via a crossbar. Each node of MANNA contains two 50MHz Intel i860XP processors, allowing one to serve as SU and the other as EU, and 32 MBytes of DRAM memory.

Evaluation. Recall that *T uses one processor to handle synchronizations and executions, and uses another processor to handle remote memory requests. EARTH uses one processor to handle the execution and a non-programmable ASIC device to handle synchronizations and remote memory requests.

A threaded version of C has been developed where the programmer explicitly specifies the thread boundaries, which serves as useful targets for more implicitly programmed languages. The results based on MANNA indicate that various EARTH operations are relatively costly: *sync* and *spawn* can take up to several hundred cycles. The large overhead is due in part to the slow interface between EU and SU, as implemented on MANNA. The ready and event queues are implemented using local memory whose read accesses take 10 cycles.

A lot of useful parallelism arises at the loop level but the programming model does not allow granularities smaller than a function to be spread across processors. Hence, the function must be explicitly broken down into sub-functions to exploit loop-level parallelism.

2.2 Code Generation

Traditionally, threads are generated either in a top-down or bottom-up manner. In the top-down approach, the code generator directly creates threads from the program dependence graph. The advantage is relatively easy thread generation since the code generator works with the higher source level constructs. On the other hand, the bottom-up approach takes a fine grain dataflow graph and attempts to coalesce instructions into threads. While the top-down code generator has a relatively narrow view of the program graph at any given time, the bottom-up generator has a view that can potentially encompass the whole graph. Thus it need not be constrained by the original higher-level source constructs; this can be especially valuable during optimizations. However, the bottom-up approach requires that a dataflow graph be generated first, which practically confines it to the functional language domain. In general, pure top-down schemes create larger threads than pure bottom-up schemes [BNSR93]. This thesis explores an approach that combines the two approaches.

2.2.1 Top-down Code Generation

Several researchers have worked on developing top-down approaches to generating threads from functional languages [CF89, Sar87, BNR89, GP92]. In most of these instances the threads block for remote memory accesses, and their approach is targeted for conventional MIMD multiprocessors or vector processors.

Most partitioning methods are based on the macro-dataflow program model and the general MIMD machine model. Macro-dataflow represents programs as acyclic dataflow graphs, organized in a hierarchical fashion where a node may contain one or more graphs. An example of macro-dataflow is IF1 [SG85a] which is used as an intermediate form for the Sisal [MSA+85] compiler.

Sarkar [Sar87] describes a general partitioning and scheduling technique which uses a table containing the parameters of a particular machine. The parameters include the number of processors, various operation latencies, communications delay, and task switching overhead. It uses macro-dataflow where each node is annotated with estimated execution cost which comes from profiling. He describes both run-time and compile-time scheduling techniques. For run-time scheduling, a set of heuristics is applied to partition the graph. The granularity depends on the trade-off between the overhead and parallelism. For compile-time scheduling, a further analysis is performed including code expansion in order to generate enough parallelism. The related tasks are then pooled together and assigned to processors. A special consideration is given to FORALL-type loops. Cann [CF89] uses a simpler heuristic to generate tasks and is mainly interested in the loop slice level.

A non-strict macro-dataflow model is described in [GN88]. A variable resolution actor model is proposed in USC Decoupled Execution Model [EG91].

A partitioning method for NUMA multiprocessors is described by Wolski and Feo [WF92]. Their graph model is annotated with variable communication costs in addition to execution costs to be used by the mapping heuristics.

Girkar and Polychronopoulos [GP92] describe a method to exploit parallelism across the loop and function boundaries through the use of a hierarchical intermediate-level graph.

A thread-based programming model for EM-4 [SKS+92] implements a thread as a set of SCB's that are sequentially ordered. Sequential threads are explicitly written by the programmer in C. A "thread" is therefore more like a code-block in TAM with each SCB corresponding to a TAM thread, although the order of execution of SCB's in a thread are statically determined whereas in TAM the order is non-deterministic. In this method, both synchronization and parallelism are expressed explicitly through a set of library functions. The functions include thread forking and buffered message passing, among others. Also, registers are spilled into memory at the end of an SCB. The programming method therefore hides the non-blocking nature of EM-4. The overall effectiveness of this approach remains to be demonstrated.

2.2.2 Bottom-up Code Generation

Many bottom-up code generation strategies based on partitioning of dataflow graphs have been developed [Ian88, SCvE91, HDGS92, Tra91]. Most of these schemes generate threads for programs written in Id [NA92], a non-strict language.

Iannucci [Ian88] describes the *dependence sets* method of graph partitioning as well as the conditions that must be satisfied for a correct and efficient partitioning of a dataflow graph. In his scheme a dataflow graph is partitioned into a set of Scheduling Quanta (SQ). His bottom-up partitioning strategy is guided by:

- Maximizing the parallelism: do not obscure the procedural or loop parallelism.
- Maximizing the run length (the critical path length) of SQ's.
- Minimizing the dynamic synchronization (inter-SQ arcs).
- Avoiding deadlocks.
- Maximizing machine utilization: this is hard to measure and not very useful.

There are two problems that might arise in a partitioned graph that could cause deadlocks: static and dynamic cycles. In a static cycle, a cycle of SQ's exists such that the program does not terminate due to cyclic dependences. A dynamic cycle involves reading and writing the same remote memory locations that otherwise have no static cycles. Static cycles have been well treated by others such as Sarkar and Hennessy.

The dynamic cycle is resolved by the method of dependence sets. In this method, each input instruction is associated with a unique name. An input instruction is the target of any remote read operations. The dependence set of an instruction is the set of all names associated with inputs that can reach the instruction without going through other inputs. With each unique set, a partition is associated. This method is sufficient to guarantee no deadlock.

Other researchers have extended Iannucci's definition to create a more general partitioning scheme (rather than just a scheme to ensure a deadlock-free execution). A version of the dependence sets algorithm along with some optimizations are used to generate code for TAM [SCvE91] from Id90 [Nik90]. The optimizations include methods to combine nearby threads to form larger clusters through the merge-up and merge-down methods. Since control is explicit in TAM, a new intermediate graph called a dual graph is used in which both control and data flows are explicit.

Attempts to create larger threads through a combination of dependence and demand sets algorithms along with some form of global analysis is described by Traub *et al.* [TCS92]. A *demand set* is similar to a dependence set except that a partition is based on a common set of outputs rather than inputs. An output is any instruction that sends a result to a non-local destination, such as a remote read or message send. Their algorithm distinguishes between certain and potential dependences. For example, there is a potential dependence between a write to an I-structure and an instruction that reads from the I-structure, since in general it cannot be discovered at compile-time which write instruction goes with which read instruction (and for non-strict programs this can depend on inputs). The general idea behind their partitioning strategy is that through an analysis, some of the potential dependences can be converted into certain (although indirect) dependences thereby providing more information to the partitioning process.

Their strategy depends on the concept of congruence, which is used to indicate commonality among inputs and outputs. For example, it is useful to know that two particular inputs depend on the same set of outputs (though they may unknown). This knowledge is used in an inter-procedural analysis where information gained from the analysis of a callee site (the block of code that is called) could be used to optimize the caller site (the block of code contains the call) and vice versa. Their method uses a combination of the dependence sets and demand sets, along with inter-procedural analysis. This procedure is repeated until there is no change in the partitioning. The extent to which this kind of analysis improves the partitioning of dataflow graphs generated from a non-strict language is uncertain.

In general, the non-strict semantics of Id requires a more careful partitioning strategy than would be required under the strict semantics of Sisal. Therefore, a strict language should be more amenable to compile-time analysis and will impart more knowledge to the partitioner.

2.3 Other Issues in Multithreading

In this section, some of the issues addressed in the previous chapter are expanded.

Locality and Storage Hierarchy. Culler *et al.* [CSvE93] argue that the existence of storage hierarchy fundamentally limits the amount of multithreading that can be easily supported, thus limiting the latency that can be tolerated.

The idea of memory hierarchy rests on the principle that fast memories are small and expensive while slow memories are large and inexpensive. The observation is that switching is cheap only for those threads residing in the top part of hierarchy. Hence, only a limited number of threads may be switched inexpensively. A scheduling policy that favors threads that already lie in the top part of hierarchy would be preferred. An open question in parallel architecture is the problem of organizing the storage hierarchy to operate in concert with the way computations are scheduled in the parallel programs. One simple approach is TAM's scheduling policy which favors threads within the currently executing quantum.

Cache is an integral part of any storage hierarchy. The basic idea of a cache is to reduce the overall access time to the main memory. The research on incorporating caches into multithreaded executions and measuring their effectiveness is ongoing. Cache in a dataflow model is discussed in [Tak92] for DFM-II [Tak87]. This model is designed for a fine-grained dataflow machine and must therefore take into account the explosion of parallelism that is typical in these machines [Cul89]. The model is evaluated using a relatively small set of hand-coded benchmarks. The caching mechanism attempts to preserve the working set of the program in the cache.

Because processor speeds are improving at a much faster rate than memory speeds in the past decade, the design of on-chip caches has become more crucial. The details of various issues in cache designs are treated elsewhere [HP90]. Next, a basic framework for a high-performance cache design is briefly described next.

A direct-mapped cache has the advantage of a shorter critical path length than a set-associative cache, at the expense of higher conflict misses. There have been several designs which try to combine the advantages of direct-mapped caches with those of set-associative caches such as the victim-cache [Jou90], MRU cache [SR86], hash-rehash cache [AHH88], and half-half cache [THG95]. In general, they split the cache into two parts: one is direct-mapped while the other is set-associative. The idea is to simultaneously send the desired address to both parts of the cache and to *assume* that the direct-mapped portion contains the data. If the assumption turns out to be false, the set-associative portion of the cache provides the data (if available) at a slightly greater cost. If there is sufficient locality, this method could result in a lower average access time than either a pure direct-mapped or a pure set-associative cache of the same size. In the model described later in this thesis, this form will be extended such that the main memory can be treated as the set-associative part of the cache.

Scheduling. TAM uses storage-directed scheduling whereby threads within a currently executing quantum get a higher priority. EARTH can also be designed to prioritize threads, if microprocessors with register windows are used, with the help of the external *Register Use Cache*. The synchronization unit will prioritize threads based on whether or not the register set associated with a particular thread's frame lives in the register window.

A study of co-locating threads based on the amount of sharing between threads is discussed in [TE94]. This should reduce compulsory and invalidation cache misses and thus improve the performance. However, their results suggest that load balancing is the most important determinant of the performance. In part, their results are due to the large threads they used (ranging in upwards of a million instructions).

A local scheduling policy is prone to anomalies [CSvE93]: when there are n tasks, one of which spawns another set of n tasks, the tasks may not be spawned until most of the original tasks are completed. Thus the utilization can be poor.

A certain amount of global strategy and view is needed. For example, if a processor executes a short task and a long task where the short task spawns off work for the other processors, it is advantageous to schedule the short task first.

Resource requirements are also vastly different depending on the algorithm: depth first tree search performs well under unfair (e.g., priority-based) scheduling but performs poor under fair (e.g., FIFO-based) scheduling. The case is opposite for traversing the linked list. In some cases, idling a processor may be better than switching to another unrelated task.

Overall, most current multithreading proposals use scheduling queues (either implicitly or explicitly) which cannot easily incorporate a scheduling policy that is directed by a higher level program understanding.

Communications. The communication cost in a large-scale multiprocessor can be reduced by exploiting the *communication locality.* In general, this implies the need to place communicating threads closer together or to reduce the overhead.

Multithreading models using code block-based frames in essence exploit communication locality by keeping local the communication among threads within a code block. For models that use thread-based frames, there must be a conscious effort to group these threads together. However, this effort must be balanced against a load balancing problem that might arise.

Parallelism. Fine-grain dataflow can potentially generate a large amount of parallelism which might cause resource deadlocks. Therefore, controlling parallelism becomes important for dataflow execution. Dataflow-derived multithreaded models can also suffer from too much parallelism, so they also need to control it.

Throttling [RS87, Sne93] is a pure run-time method of controlling parallelism. New activation requests may be suspended if the run-time mechanism decides there is too much parallelism based on the availability of resources. The suspended process is reactivated some time later when sufficient resources become available. When an active process finishes, its context can be used by another process. Processes have to be fairly large, otherwise they lead to too much throttle overhead; however, processes that are too large could have large internal parallelism and cause resource deadlocks. It is therefore important to find a proper balance.

K-bounding [Cul85] is a method that controls loop parallelism in Id. As originally designed, the compiler analyzes the code and determines the maximum resource usage for a loop cycle. At run-time the hardware decides the number of loop cycles that can be allowed to execute in parallel based on the activity level of the machine and the static information of maximal resource usage. Machine resources are recycled and reused. However, the bounds are statically determined in the current implementation.

Chapter 3

BOTTOM-UP THREAD GENERATION AND EVALUATION OF HYBRID EXECUTION MODEL

In this chapter, several issues related to medium grain hybrid execution are explored. Central to this study is the concept of *clusters* of instructions that become the schedulable units of execution. In this model, clusters can be seen as the dataflow equivalent of basic blocks in sequential programs. This chapter describes two bottom-up algorithms that generate these clusters from fine grain dataflow code as well as a set of optimizations for these clusters. The two algorithms are the *basic block algorithm* [NRB93] and the *dependence sets algorithm* [Ian88, SCvE91]. These algorithms are applied to a set of benchmark programs and the resulting code characteristics are presented in detail. In addition, performance figures are derived for a variety of simulated machine models. The performance figures are used to identify bottlenecks and gauge the effectiveness of various architectural features in terms of scalability, tolerance to network latency and cost/performance tradeoffs.

The results indicate that despite relatively small grains (on average between two and four instructions per grain) that were generated, the medium grain model outperforms the fine grain model under a wide range of machine models. Further, it is demonstrated that the medium grain model, by exploiting instruction level locality and hence reducing communication and synchronization overheads, is better able to tolerate increases in network latency and synchronization costs than the fine grain model.

In the following sections, the research on the partitioning strategies and the evaluation of these strategies is presented.

3.1 The Cluster Model

In this section, the cluster-based medium grain dataflow model is presented in detail. First, the fine grain model is briefly reviewed and then the definitions related to medium grain execution are discussed.

Fine grain execution model: In dynamic, pure dataflow, execution of an instruction is triggered by the arrival of input values via tokens. Each instruction is hence independently scheduled dynamically.

Definition 1 Well-behaved dataflow graphs: *A graph is said to be well-behaved if and only if no token is left in the graph after the last instruction has executed. This property of dataflow graphs implies that every generated token will eventually be consumed by the execution of some instruction. Note that well-behavedness is a property of the* whole *graph and is not tied to any specific construct in that graph. This property is used by partitioning algorithms to ensure deadlock-free medium grain execution.*

Definition 2 Cluster: *A cluster is a connected, directed, and non-blocking dataflow graph of instructions. It is scheduled as a unit and the graph imposes a partial order on instructions executing within it.*

Medium grain execution model: In a medium grain dataflow graph, execution of a cluster starts when all input data is available. The order of execution preserves the partial order as defined by the cluster graph. The medium grain model can execute in one of two modes: **blocking** or **non-blocking.** A cluster can block when executing an instruction that has unbounded latency such as a remote memory access or a function call, and other instructions in the same cluster depend on the results of the unbounded latency instruction. A context switch to other ready clusters can then occur. In the non-blocking mode, a cluster is constructed such that it can run to completion without blocking. An advantage of this is that the overhead of context saving is eliminated. However, cluster sizes may be smaller than those of blocking clusters. This work is confined to non-blocking clusters. Therefore, a cluster is a piece of code that can be executed on a processor with bounded latency.

Threads versus clusters: A thread is an instantiation of a cluster. Internal cluster parallelism can be exploited by superscalar, superpiplined, and VLIW processors which can take advantage of the instruction-level parallelism. It is relatively easy to order the instructions within the thread that will maximize the utilization of a given processor since the dependence graph of instructions within the cluster is known. The scheduling techniques for various types of processors are described in [CCMH91, Gro83, Lam88, Muc88, WS87]. The execution of a cluster can be summarized by the following set of rules:

- Execution starts when *all* the input tokens to the cluster are available. This property of cluster execution will be referred as *strict* cluster execution. A non-strict model would allow the execution to start when a predetermined subset of the input tokens is available.
- Once started, the execution of a cluster continues to completion without blocking and has a bounded execution time. This implies that an instruction that issues a memory request cannot be in the same cluster as the instructions that use the value returned by the split-phase read.
- The cluster based execution of a program must be deadlock free. This implies that the structure of clusters within a program will guarantee that all the inputs of every cluster will become available and hence allow a strict execution.

Note that the strict execution and deadlock-free rules are not unrelated. It can be shown that if every cluster in a graph can always execute strictly (meaning that all its input will become available for every instantiation) then the whole graph will always execute without deadlock.

The move from fine-grain execution, as in the traditional dataflow architectures where instructions are monadic or dyadic [GKW85], to medium-grain execution, where instructions can have more than two inputs, requires modifications to the basic matching mechanism. The Explicit Token Store [PC90], a matching and synchronization mechanism based on the use of frame memory, implemented in the Monsoon architecture, can be extended naturally to accommodate several inputs per cluster.

3.2 Cluster Generation Strategies

In this section, partitioning strategies that generate clusters are described along with their objectives. Two algorithms for partitioning dataflow graphs into clusters have been implemented: the *dependence sets* algorithm (DSA) as described in [Ian88] and [SCvE91] and the *basic block* algorithm (BBA) [NRB93]. The dataflow graphs used in this work are generated from programs written in Sisal [MSA$^+$85] and compiled using the Manchester Dataflow Machine [GKW85] compiler. The construction of clusters is guided by three main requirements:

1. The formation of clusters should not infringe on the function and loop-level parallelism in the graph. Since loop level parallelism is the most common and useful form in scientific codes, the construction of a cluster should not only preserve it but also facilitate its exploitation.
2. Ensuring deadlock-free execution of clusters.
3. Minimizing matching and synchronization overhead by maximizing the use of locality and making clusters as large as possible without violating the first two objectives.

The motivation for these requirements is to maximize locality within clusters and to take advantage of parallelism between clusters. The above objectives imply that a cluster cannot be larger than a loop body. However, clusters will frequently be smaller to ensure that they are non-blocking, allow strict execution and do not introduce potential deadlock. Even though Sisal is a strict language, deadlock can occur in bottom-up generated clusters due to certain partitioning that results in circular dependencies (see Figure 3.2 discussed later).

The strategies of the two algorithms differ. The DSA algorithm tries to gather in one cluster nodes that depend on the same set of inputs. The BBA algorithm tries to build as large a cluster as possible whether or not all nodes have common inputs or not; it is a greedy algorithm. The BBA algorithm generates larger clusters than the DSA algorithm, but the algorithm is more complex and can suffer from increased latency during run time due the larger size of clusters and larger set of inputs.

As a first step for both algorithms, the fine grain dataflow graph edges are classified into two types: *normal* and *terminal* edges. A terminal edge connects two nodes that must be in two distinct clusters. A normal edge connects two nodes that are in the same cluster[1]. The initial classification into normal and terminal edges is made by a graph analysis routine which uses the following set of rules:

- Output edges from a split-phase operation (e.g., structure store request) are terminal. When a request is sent to the structure store for a structure element access, the latency of the reply is unbounded. Therefore, in order to meet the non-blocking execution requirement, the output edges from these instructions must have a destination outside the cluster.
- Output edges from tag modifying instructions are made terminal. This ensures that functional and loop parallelism are preserved at the cluster level.
- Implicit merge edges are made terminal. An *implicit merge* occurs when more than one arc have the same input port of a node as their destination. The well-behavedness of the graph ensures that only one of these arcs will have a token of a given color for every instantiation of this node. Implicit merges occur in constructs such as: *function calls*, *if-then-else* and sequential *for-init* loops (backedges). An example of an if-then-else implicit merge is shown in Figure 3.1. In this example, either node 2 or node 3 will execute in any activation of this graph, but never both.
- Output edges of branches are terminal[2]. This is done as a conservative measure that simplifies the job of ensuring deadlock-free execution.

In later optimization stages, some of these edges might be changed from terminal to non-terminal if determined safe by the analysis.

3.2.1 The Basic Block Algorithm

In the basic block algorithm, the following steps are performed:

1. An initial set of clusters is built. This is derived immediately from the previous analysis step where the edges are classified. By removing the terminal edges, islands of connected graphs are formed, with each island becoming a cluster. However, these clusters may deadlock due to circular dependency. Circular dependency may arise from two possible cases:
    - *Static cycles* occur between two or more clusters with circular dependencies. Such a situation involving two clusters is depicted in Figure 3.2. Here, neither cluster can execute because each need the result of the other.
    - *Dynamic cycles* occur when compiling for non-strict structure store access mode[3] with the possibility of reading and writing the same memory location. In Figure 3.2b, RSS (read structure store) may read the same memory location that is written by WSS (write structure store). This effect is indicated by the broken line from WSS node to RSS node. Cluster C may deadlock because one of the inputs is the split-phase output of the RSS node and the value is not available until cluster C executes.

---

[1]The *strongly-connected arcs* of EM-4 [SYH+91] correspond to our normal edge whereas their *normal arc* correspond to our terminal edge.

[2]In [NRB93], the basic block algorithm that was described did not make branches terminal which required additional check for deadlock-free executions.

[3]In a non-strict structure store access, an element of a structure may be read before the entire structure is written, and requires I-structure [AT80] type of support.

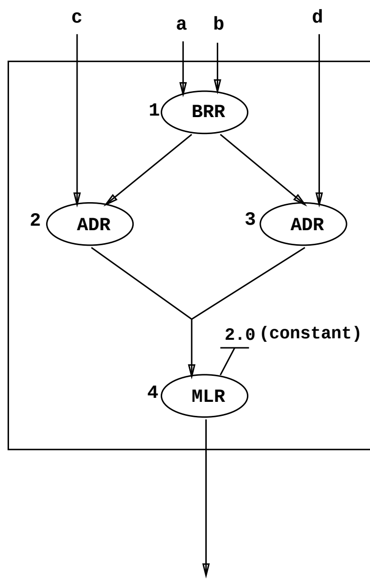


Figure 3.1: If-then-else merge

2. Deadlock cases are resolved during the fission stage by checking for dynamic and static cycles in each cluster. Static cycles are detected by starting from each result of each cluster and checking whether there is a path back to the cluster. If the path does not represent a safe cycle of a sequential loop, then there is a potential for deadlock. Reads and writes to the same array in one cluster are pessimistically assumed to create a dynamic cycle. When a cycle is detected, the cluster is split into two or more clusters, i.e. the target node of the cycle and its non-cyclic dependents are split off, and the process is repeated.

3.2.2 The Dependence Sets Algorithm

In the dependence sets algorithm, the goal is to try to put nodes that depend on the same set of input nodes (the *dependence sets*) into the same cluster. An input node is defined as target of either a terminal edge or an input datum. The steps used by the algorithm are summarized as follows:

1. Each input node is assigned a unique name α and its dependence set S is initially set to $\{\alpha\}$.

2. The dependence set of a node is the union of dependence sets of its normal-edged predecessors. The dependence sets do not cross terminal-edges. Then by a transitive closure algorithm, the dependence sets are built.

3. All nodes with the same dependence sets are then collected into one cluster.

3.2.3 Optimization (Fusion Phase)

Both the basic block and dependence sets algorithms are followed by an optimization phase whose purpose is to form larger clusters by combining two neighbors while preserving the semantics of strict execution and freedom from deadlock. This step corresponds to merge up/merge down in [CSS$^+$91, NRB93]. We look for two cases as candidates for this *fusion* (see Figure 3.3):

- *Merge up:* if cluster B gets all its inputs from another cluster A, then B are merged up with A. This is safe since cluster B can only execute when cluster A executes and produces results. Therefore, if A and B are combined, with inputs only to A, all the conditions are satisfied including deadlock avoidance. In this process, the new cluster has the same set of inputs as cluster A and therefore the latency of its inputs would be the same as for cluster A. Input latency, in this chapter, refers to the

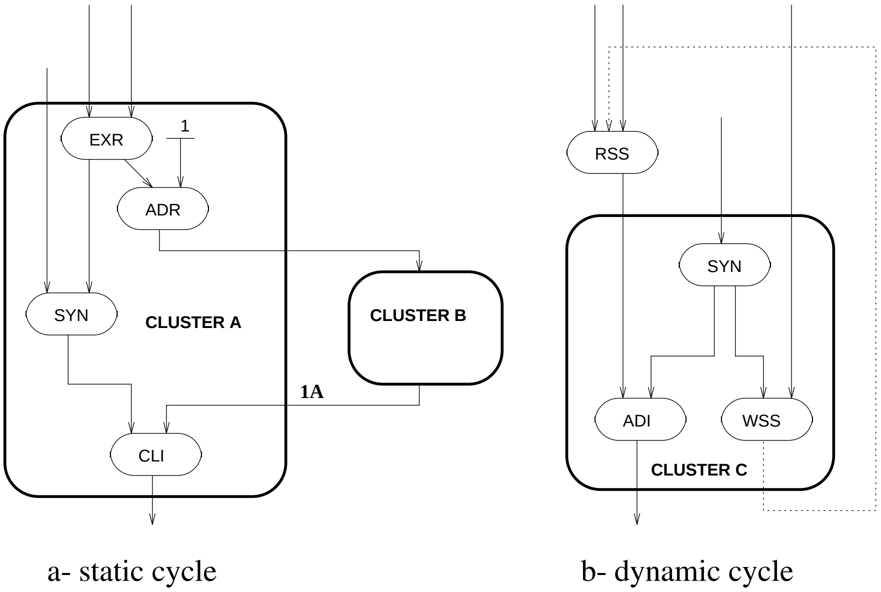


Figure 3.2: Deadlock potential due to circular dependency

time delay between the arrival of the first token to a cluster instance and that of the last token, at which time the thread can start executing. As the number of nodes in a cluster is increased, the input latency of the cluster *might* increase to the point where it offsets any gain achieved by the reduction in matching costs.

- *Merge down:* if cluster A sends all its outputs to the same cluster B, then A is merged down with B. This is safe since cluster B can only execute after cluster A executes and produces results. In this case however, the set of inputs to the new cluster is the union of the inputs to clusters A and B. Whereas cluster A could have started executing when all its inputs are available, it now must wait for the inputs to cluster B as well. Consequently, merge down can potentially reduce the pipelining among clusters by increasing input latency and therefore negatively affect the execution time. This issue is further discussed in Section 3.4.2.

In order to preserve function parallelism and retain bounded cluster execution time, fusion is not performed across remote memory access operations, and function call and parallel loop interfaces. Merge down, in addition, is not performed across tag modifying operators. An additional check is made to insure that no cycle with a potential for deadlock is introduced (this is because fusion can take place across branches).

Another optimization step checks to see if there are more than one arc carrying the same data value from one cluster to another, if so, they are redundant and all except one are removed.

3.2.4 Example

An example Sisal code is shown on Figure 3.4. The program is Purdue Benchmark 1, which is a trapezoidal integration of an exponential function. The corresponding dataflow graphs after applying the BBA and DSA algorithms are shown in Figure 3.5. The heavy lines group nodes into clusters. On the right side of each graph are the cluster-level representations of the graphs.

For the BBA graph, the cluster 2 acts as the initialization routine that takes the inputs n, a, and b and calculates the integration width and other constants and also drives the loop which is encompassed by clusters 3 and 4. Cluster 3 calculates $e^{a+real(i)*h}$ and Cluster 4 handles the summation reduction. Finally, Cluster 5 computes most of the last expression of the code and outputs the result.

In contrast, the DSA graph has a larger number of clusters (10 clusters versus 4) and consist of relatively small clusters. Nodes in Cluster 2 of the BBA graph now belong to a number of different clusters including several single node clusters in DSA. For example, Cluster 8 contains the loop control including the zero

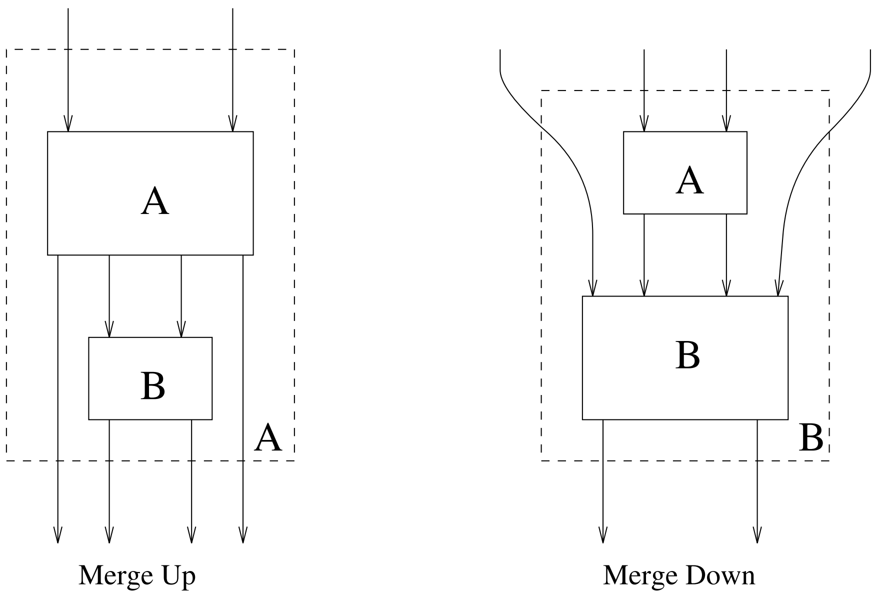
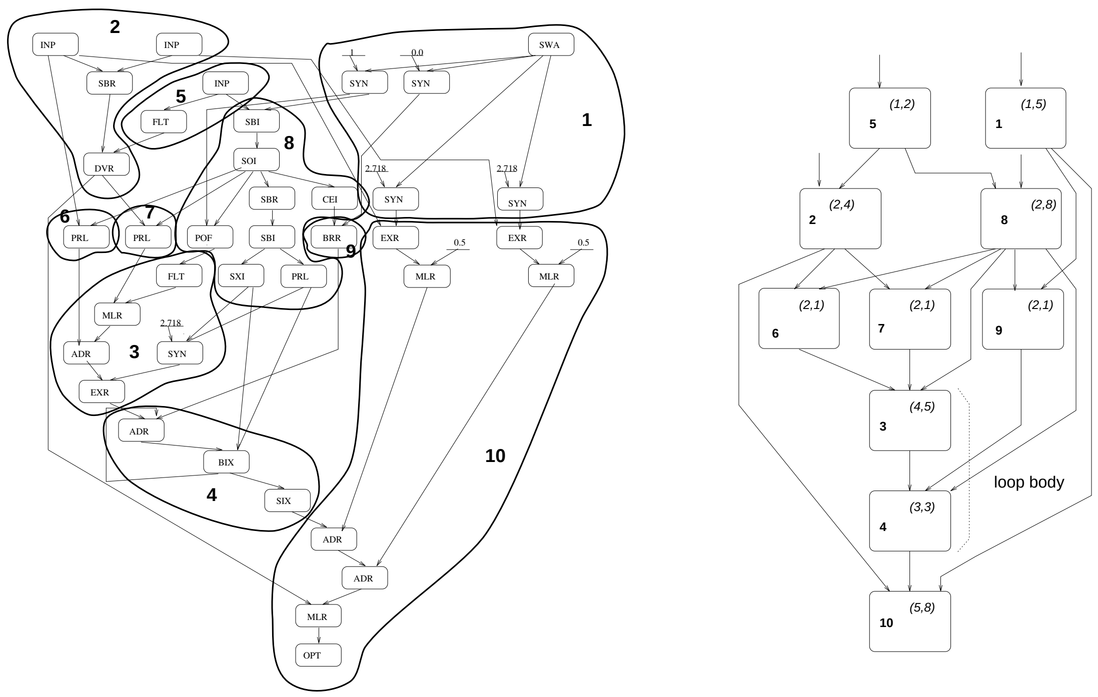


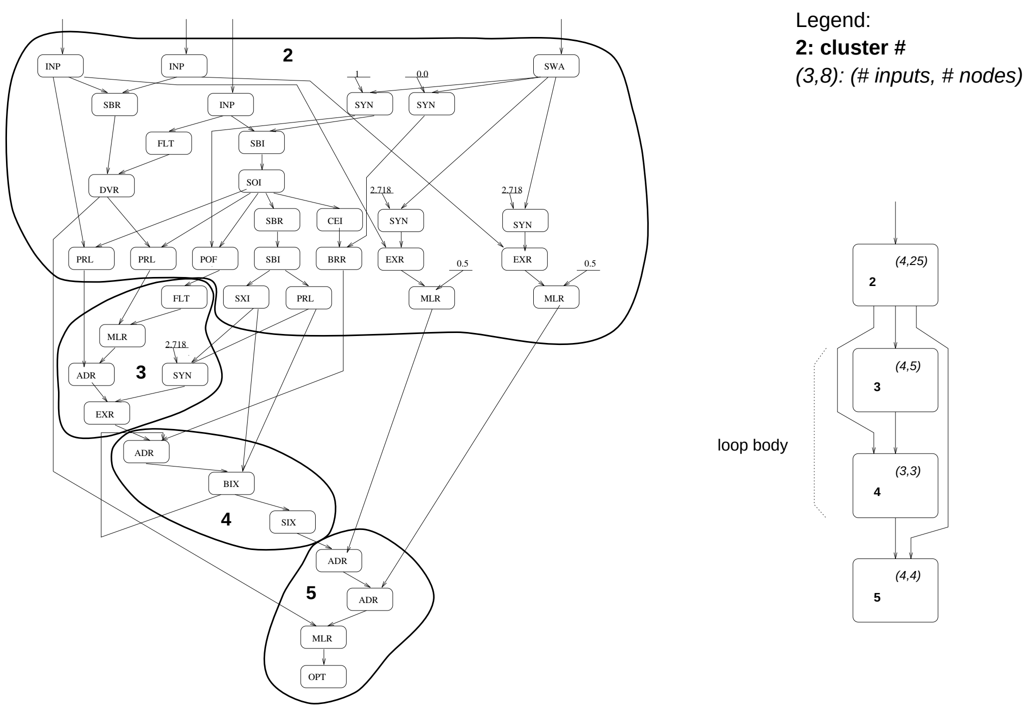
Figure 3.3: The Two Cases for Cluster Fusion

```
function Main(n: integer; a, b: real  returns real)
let
      h := (b-a)/ real(n);
      ans :=
           for i in 1, (n-1)
                returns value of sum exp(2.718, a + real(i) * h)
           end for
in
      h * (ans + 0.5 * exp(2.718,a) + 0.5 * exp(2.718,b))
end let
```

Figure 3.4: Example Sisal Program (Purdue 1)



DSA



BBA

Figure 3.5: Purdue Benchmark 1 Clustering Example

trip count handling. Another difference is that Cluster 10 of the DSA now embodies the entire expression $h*(ans+0.5*e^a+0.5*e^b)$. In both graphs, the loop body is identical (clusters 3 and 4 in both graphs). It has been observed that both algorithms generate similar loop bodies when the loop is simple (with no inner loops inside).

Even though the clusters generated seem quite different, the execution time of the two graphs is quite comparable (less than 5% difference), since most of the execution is in the loop body.

3.3 Cluster Characteristics

In partitioning the fine grain graph into clusters, the dependence sets algorithm (DSA) takes a more conservative approach: whereas the basic block algorithm (BBA) tries to find the largest possible clusters under the given constraint, the DSA has the additional constraint that all the instructions within a cluster must depend on the same set of inputs. In this section, various metrics of the resulting clusters on an idealized execution environment are measured (e.g., infinite number of processors, zero communication delays). Note that the intrinsic properties of benchmarks are measured without regard to any particular architecture.

The static measurements reveal the properties intrinsic to the graphs. In dynamic measurements, the clusters are given weight according to the number of times they execute. The dynamic measurements therefore give more weight to (inner) loop bodies. Unless otherwise noted, all measurements are dynamic.

3.3.1 Measurements

Benchmarks. For measurements purpose described in this and the next section, an extensive set of 50 benchmark programs are used which consist: Purdue Benchmarks [1-16] [Ric85, AHO90], Lawrence Livermore Loops [1-24] [McM86, Feo87], larger programs including SIMPLE (a 2-D hydrodynamics code) and SGA (a genetic algorithm program searching the minimum of a simple function [Gol89]), and non-numerical and miscellaneous small programs such as 8queens, Quicksort, Sieve of Eratosthenes, Bubblesort, and Matrix Multiply.

Parameters. The following set of parameters were collected from the cluster graphs:

- *Cluster size* (S_1) measures the number of instructions per cluster.
- *Cluster critical path length* (S_∞) measures the path length of each cluster.
- *Cluster parallelism* (π) measures the average parallelism of clusters (which is S_1/S_∞).
- *Number of inputs* is the average number of input arcs per cluster (I_c) in the medium grain model or per instruction (I_f) in the fine grain model, averaged over an entire program execution.
- *Matches per instructions* measures the ratio of the total number of inputs to the total number of instructions executed over the entire program. It is denoted by MPI_c for the cluster model. Note that the number of matches per instruction in the fine grain model is equal to I_f, the average number of inputs per instruction.

3.3.2 Results and Discussion

The cumulative distributions of the various cluster parameters (dynamic measures) across all benchmarks are discussed in this section. Note that a cluster of Manchester instructions would translate into a much larger number of RISC instructions. This is due the fact that many Manchester instructions are complex instructions and can generate one or two results. For example, DMD (divide integer with modulus) generates both integer quotient and modulo. Some iterative instructions, such as PRL (proliferate) that generates a range of tokens, really translate into a loop in RISC code.

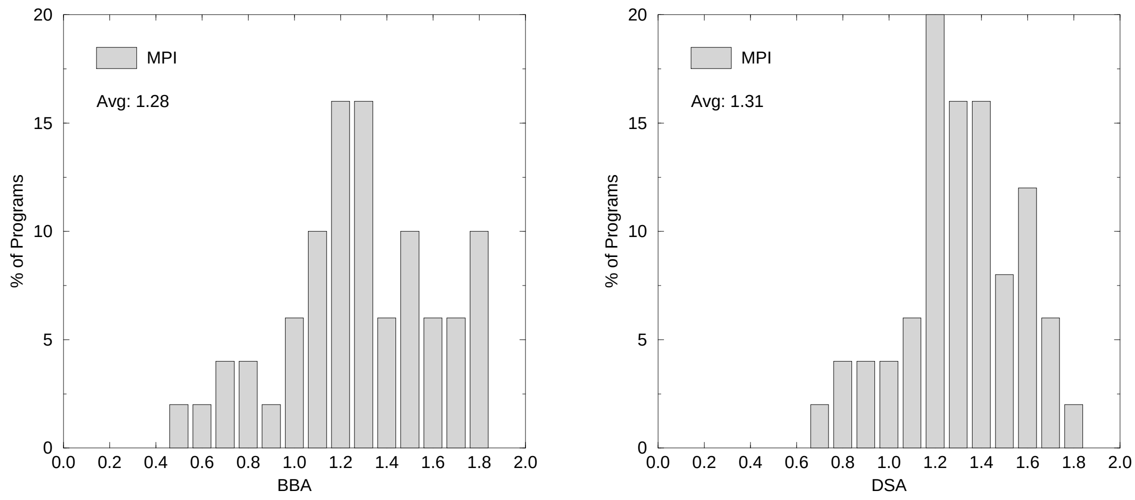


Figure 3.6: Matches per instruction (MPI_c)

Synchronizations (Matching). The average number of inputs per instruction (MPI_f) for the fine grain execution model is 1.78. Figure 3.6 shows the distribution of matches per instruction (MPI_c) in the cluster execution model for the BBA and DSA, respectively. The figures show that results are similar between BBA and DSA clusters and that there is a sizable reduction in the *MPI* compared to fine grain. This result indicates that a cluster based execution has a significantly lower amount of matching overhead.

Cluster Size. The cluster size across all benchmarks (Figure 3.7) ranged from as low as 1 to more than 14 nodes with the average being 4 nodes for BBA and 2.2 nodes for DSA. Approximately 35% of all clusters consist of a single node. This fact is, to a large extent, due to the way the Manchester Dataflow compiler generates fine-grain dataflow graphs. Furthermore, generating non-blocking code bottom up makes it difficult to eliminate many single node clusters. For example, a stream of inputs to the WSS (Write to Structure Store) results in the WSS instruction to be in a cluster by itself.

Inputs. In Figure 3.8, we can observe that more than 80% of all DSA clusters executed have inputs less than or equal to three and that over 95% have inputs less than 5. The corresponding figures are even higher for the BBA clusters. The average number of inputs per cluster (< 3) is less than a factor of 2 larger than the fine grain ($I_f = 1.78$).

Parallelism. The distribution of the average parallelism π within a cluster is shown in Figure 3.9. Note that over 80% of all clusters executed were sequential ($\pi \approx 1$) [NMB92] for both algorithms. The BBA clusters have a larger average internal parallelism (1.15) than the DSA clusters (1.05). Even though the averages are relatively low, due to variance, processors can still take advantage of the parallelism present.

Critical Path Lengths. The distribution of S_∞ shown in Figure 3.10 confirms the data for S_1 and π: most clusters have a relatively short critical path length averaging 2.5 for BBA and 2.0 for DSA. In 97% of cases, S_∞ is less than or equal to three.

Note that while the averages across all benchmarks of the cluster size and the number of inputs per cluster are 4 and 3 respectively (for BBA), the average number of matches per instruction is larger than 3/4. This is due to the fact that the former case represents the ratio of two averages over the entire benchmarks and the latter case represents the average ratio for each benchmark across all benchmarks.

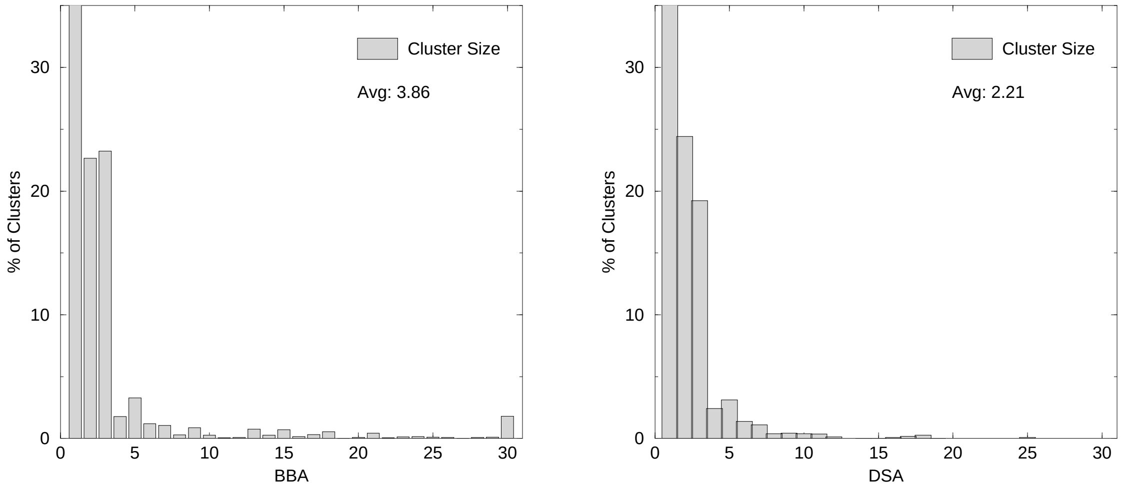


Figure 3.7: Cluster size (S_1)

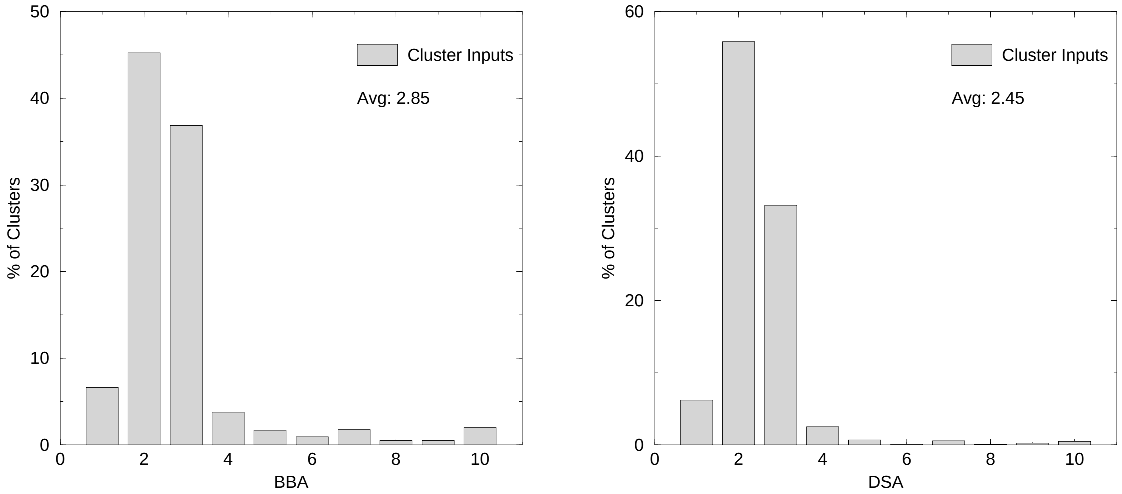


Figure 3.8: Number of inputs per cluster (I_c)

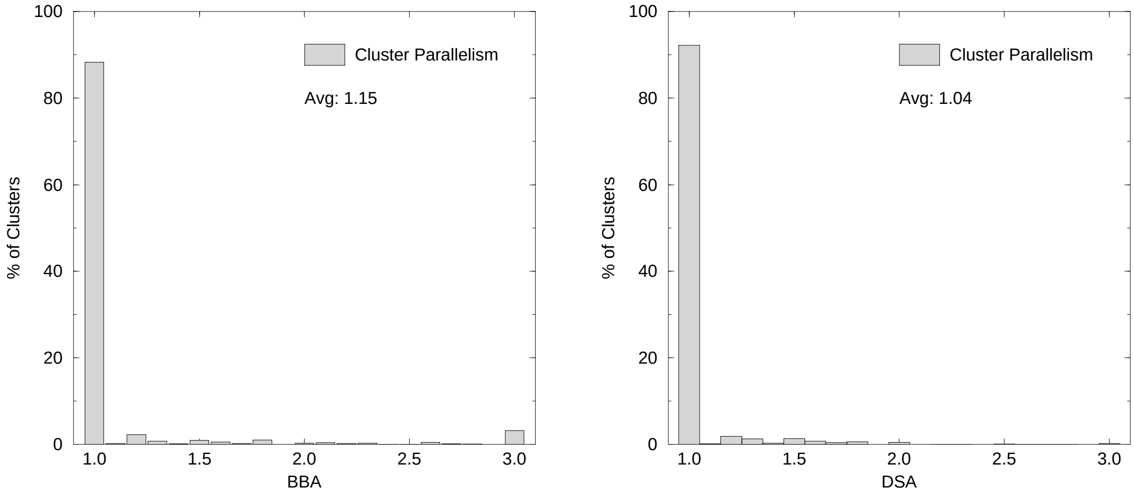


Figure 3.9: Parallelism per cluster (π)

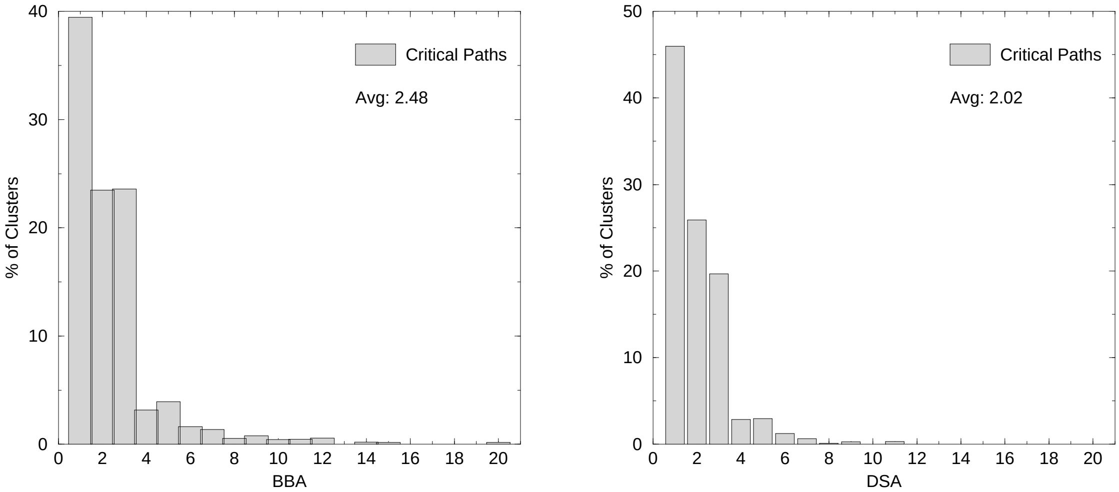


Figure 3.10: Critical Path Length (S_∞)

		S_1	I_c	I_f	S_∞	π
BBA	Static	3.89	2.35	1.48	2.23	1.13
	Dynamic	3.86	2.85	1.78	2.48	1.15
DSA	Static	2.20	2.06	1.48	1.83	1.07
	Dynamic	2.21	2.45	1.78	2.02	1.04

Table 3.1: Static and dynamic cluster parameters (average across all benchmarks)

		S_1	I_c	I_f	MPI_c	S_∞	π
BBA	Purdue	6.05	3.16	1.73	1.05	3.34	1.23
	LL Loops	2.90	2.80	1.85	1.46	2.04	1.12
	Non-numerical	3.08	2.30	1.67	1.17	2.12	1.16
DSA	Purdue	2.69	2.54	1.73	1.17	2.41	1.06
	LL Loops	1.89	2.45	1.85	1.43	1.78	1.02
	Non-numerical	2.19	2.13	1.67	1.25	1.78	1.10

Table 3.2: Dynamic cluster parameters (averaged per suite)

Static vs. Dynamic Results. Table 3.1 shows both the static and dynamic values of the cluster parameters averaged across all benchmarks for both algorithms. These values give some indication on the characteristics of clusters inside loops as opposed to clusters that are outside loops. They indicate that the differences are minor. The only notable difference is that inside-loop clusters tend to have larger number of inputs than outside-loop clusters.

Comparison Among Classes of Benchmarks. In Table 3.2 we show the averages of cluster parameters among different families of benchmarks for the dynamic case. We notice that there is a fair amount of difference between Purdue and Livermore loops benchmarks. For the BBA clusters, the average cluster size of Purdue benchmarks is more than twice as large as that of Livermore loops. For the DSA clusters, the difference is smaller but still sizable. The explanation for this fact is that there are more array accesses within the loop bodies of the Livermore loops than for the Purdue benchmarks. Since array accesses are terminal nodes in the cluster model this results in smaller clusters. This also explains why the Purdue benchmarks have a larger number of inputs per cluster, a larger degree of parallelism within the cluster and a lower MPI_c. The characteristics for the Non-Numerical benchmarks (i.e., Quicksort, 8Queens, and etc) fall somewhere in between the other two.

3.4 Architectural Performance Evaluation

In this section, we present the performance evaluation of cluster execution for a variety of machine configurations with varying processor characteristics. We compare the performance between different machine configurations and also against the fine grain execution model. The results are obtained using a cycle-level, discrete event machine simulator Aggies [Roh92] that allows variations on the basic dataflow machine model. The following set of parameters were varied to emulate different machine models.

- *Number of processing units (P):* where each processing unit consists of a matching store or synchronization processor, an execution processor, and a I-structure store[4] The values of P were varied from one to ∞ meaning the largest number of processors a benchmark could use for the given problem size.
- *Degree of instruction level parallelism (ILP) of a processor:* refers to the ability of a processor to take advantage of ILP within a cluster. This may range from 1 as in the conventional RISC processor to "infinite" (i.e., capable of taking advantage of any parallelism available within a cluster). The default value was chosen to be two because most advanced commercial microprocessors have this much internal parallelism. Given that the dependence among the instructions within a cluster is known from the original fine-grain graph, the scheduling of instructions at run-time to exploit any available hardware parallelism is trivial.
- *Output bandwidth (BW):* this refers to the number of tokens a processor can put on the network in each CPU clock cycle. It is not realistic to expect a value of BW much beyond two with the present state of technology. To investigate its effect on execution time, we have varied BW from one to ∞ with two being the default.
- *Network latency (L):* is the cost of remote memory reference and inter-processor communication (in clock cycles). The latency here represents an average measure, taking into account that some percentage of memory references are local and therefore need not travel through the network. L was varied from zero to 30 clock cycles with a default of five.
- *Synchronization cost (SC):* is the cost of the synchronizing/matching operation per token. It is a pair where the first value indicates the cost of a successful match (last token) and the second that of a failed match. Two sets of values were chosen, a low matching cost (2,1) (the matching cost of the EM-4 whether the input comes from the network or locally) and a high one (5,4) with the low one being the default.

These parameters are applicable to both fine grain and cluster execution, with ILP having a minor impact on the fine grain execution. It should be noted that in this performance modeling, the ILP and bandwidth could be independently varied. Although an instruction can only generate one result per cycle, multiple copies of this result could be sent in a single cycle if the bandwidth allows it (we are assuming there exists hardware for this in the output unit of both the processor and the structure store). In all the simulations, we have used the operation latencies of the Motorola MC88110 with some Manchester instructions mapping into several RISC instructions. Table 3.3 shows some of these latencies. While some of the parameters correspond to real values, e.g. MC88110 and EM-4, others are used to just evaluate the trend irrespective of actual values.

Instruction	**Latency**
ADI (add integer)	1
ADR (add real)	3
MLI (multiply integer)	3
DVR (divide real)	13
RSS (read structure store)	5 + network latency

Table 3.3: Instruction latencies used for the benchmarks.

The objectives of this analysis are twofold:

[4] We also experimented by varying the number of matching stores, processors, and I-structure stores independently. These results are not discussed any further in this thesis except to say that their relative proportion need to be balanced so that one class of modules does not become a bottleneck.

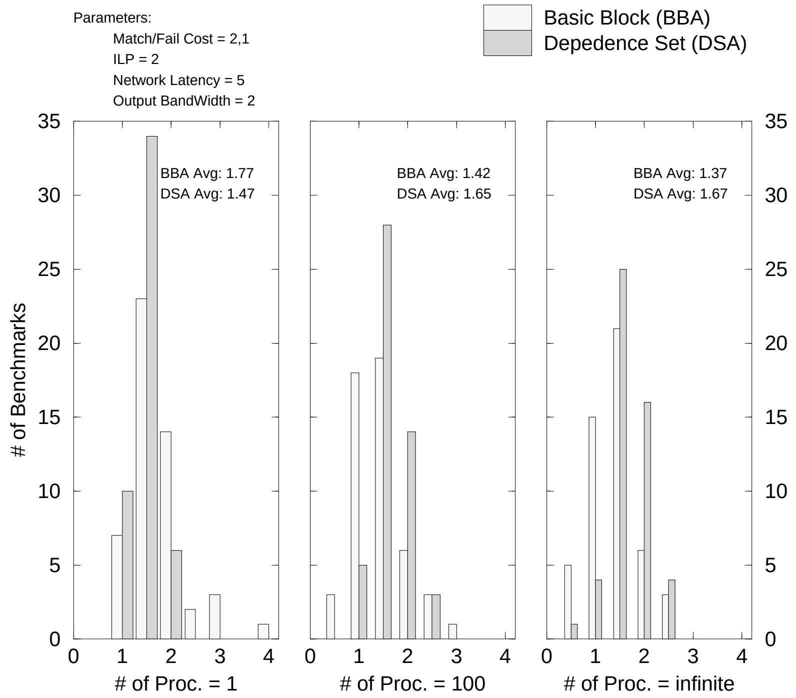


Figure 3.11: Distribution of Speedups

1. **Effects on the speedup.** Given the same machine configuration, we want to investigate how a given feature affects the speedup of the medium grain execution over the fine grain execution. The metric used here is the average speedup across all benchmarks denoted by S_f which is the ratio of the fine grain to medium grain execution times. In most cases, we will limit the data presented to the average speedup which is the average of the speedups across all 50 benchmarks. In some instances we will describe the distribution of the speedup across benchmarks without displaying the data.

2. **Effects on the medium grain execution time.** Here the objective is to measure the percentage of degradation in medium grain execution time, denoted by Δ, with respect to an ideal (but unrealistic) parameter value. For example, in evaluating the effect of BW, Δ would be the difference (in %) between the medium grain execution times for $BW = \infty$ and a finite BW. As an example, in Table 3.6, reducing BW from ∞ to one with $P = \infty$ increases the execution time by 52% in the DSA and 97% in the BBA.

It is to be noted that the above two concepts are distinct. The former objective represents the ratio of medium grain execution time over the fine grain execution time. As we vary various architectural parameters, both medium grain and fine grain execution times will change. In contrast, the latter objective represents the differences in medium grain execution times only.

3.4.1 Results

Speedup over Fine grain Execution. Figure 3.11 shows the three distributions of speedup S_f for $P = 1, 100$ and ∞. The average speedup over the entire benchmarks ranges from 1.34 to 1.77. The following observations can be made:

- For $P = 1$, the speedup distribution for BBA has a wider spread than that of DSA and has a higher average.

	DSA $P =$			*BBA* $P =$		
Class	1	100	∞	1	100	∞
Purdue	1.44	1.65	1.77	1.90	1.37	1.25
LL Loops	1.45	1.63	1.58	1.67	1.34	1.29
Non-numerical	1.75	1.87	1.86	2.08	1.86	1.83

$ILP = 2$; $SC = (2,1)$; $BW = 2$; $L = 5$

Table 3.4: Speedup Among Classes of Benchmarks

- This situation is reversed for $P = \infty$ where the average speedup for DSA is higher and the speedup spread of BBA is narrower.
- By looking at the actual data we observe that BBA achieves about 20% greater speedup than DSA for low number of processors (< 50) while DSA is up to 24% better for an infinite number of processors. Explanations for these phenomena are given in Section 3.4.3.
- With infinite processors, the average speedup is still greater than 1. Therefore, we can see that the medium grain model is scalable and can exploit available parallelism as well as the fine grain model.
- The data also illustrates that as the number of processors increases, the speedup increases for DSA while it decreases for BBA. This implies that DSA is more scalable than the fine grain model and BBA is less scalable as the number of processors increases. This is one of fundamental differences between the two algorithms and is mainly due to more pronounced role that increases in input latency plays. This issue is discussed further in Section 3.4.3.
- In this data we have observed that even for $P > 10$ there still are 2 benchmarks that have $S_f < 1$ (i.e., where the fine grain execution does better than the medium grain) indicating that not all programs do benefit from the proposed medium grain execution model.

These speedup results can be attributed to:

- The reduction in synchronization cost due to reduction in the number of tokens communicated.
- The ability of a medium grain processor to take advantage of ILP present in codes.
- The cases of $S_f < 1$ are most often attributable to a large increase in input latency that overshadows the above benefits.

In Table 3.4, speedup figures broken down into three classes of benchmarks are shown. The pattern observed when the average is taken over all the benchmarks is also observed in each class of the benchmarks, e.g., as the number of processors increase, speedup goes up for DSA but the situation is opposite for BBA.

Even though the average cluster size of Purdue is much larger than that of Livermore Loops, the speedup over the fine grain model is relatively similar. The non-numerical benchmarks, whose average cluster size is in between the other two, in general, achieves higher speedups than the other two. Therefore, the cluster size does not necessarily indicate the achievable speedup.

In Table 3.5, speedup figures are derived by weighing each benchmark by its execution time and by its code size rather than just simple average across all benchmarks. The pattern observed for the simple average is also observed in weighted averages, and is relatively uniform across different classes of benchmarks and spatial and temporal sizes of benchmarks. Also, for the execution time, weighted averages are very similar to simple averages. The partitioning behavior of loops which dominate the execution time is similar across different benchmarks. Also, the code size weighted averages indicate that the speedups are similar for larger programs for DSA, but less so for BBA.

	DSA $P =$			*BBA* $P =$		
Weighted by	1	100	∞	1	100	∞
Execution time	1.44	1.51	1.52	1.74	1.33	1.30
Code size	1.52	1.59	1.60	1.94	1.25	1.15

$ILP = 2$; $SC = (2,1)$; $BW = 2$; $L = 5$

Table 3.5: Weighted-average Speedups

		DSA $P =$			*BBA* $P =$		
	BW	1	100	∞	1	100	∞
S_f	1	1.48	1.53	1.57	1.77	1.26	1.21
	2	1.47	1.65	1.67	1.77	1.40	1.34
	3	1.47	1.67	1.72	1.77	1.47	1.40
	4	1.47	1.69	1.75	1.77	1.51	1.44
	5	1.47	1.70	1.76	1.77	1.53	1.46
	∞	1.47	1.75	1.85	1.77	1.67	1.62
$\Delta(\%)$	1	1	38	52	0	78	97
	2	0	12	21	0	33	44
	3	0	7	13	0	21	29
	4	0	5	9	0	16	21
	5	0	4	7	0	13	17

Δ wrt $BW = \infty$; $SC = (2,1)$; $ILP = 2$; $L = 5$

Table 3.6: Effects of Output Bandwidth

		DSA $P =$			*BBA* $P =$		
	L	1	100	∞	1	100	∞
S_f	0	1.45	1.33	1.25	1.75	1.03	0.88
	5	1.47	1.65	1.67	1.77	1.40	1.34
	10	1.50	1.81	1.86	1.80	1.63	1.61
	30	1.58	2.10	2.17	1.90	2.10	2.13
Δ (%)	5	3	50	61	4	36	39
	10	7	102	123	8	72	78
	30	28	317	373	28	220	237

Δ wrt $L = 0$; $SC = (2,1)$; $ILP = 2$; $BW = 2$

Table 3.7: Effects of Network Latency

Output Bandwidth. Table 3.6 shows the effects of varying the processor output bandwidth from 1 to ∞ on the average speedup S_f and the percent degradation Δ with respect to $BW = \infty$. The following can be observed:

- Increasing the output bandwidth has no effect on the speedup for a low number of processors. This is due to the fact that the available parallelism is large enough to keep the processors busy. In fact, the bandwidth limitation functions as parallelism throttle, resulting in smaller resource requirements. This suggests the need for a balance between bandwidth and number of processors.
- For larger P, the increase in BW has a significant positive effect on speedup. While the fine grain model cannot take advantage of output bandwidth beyond two, for the medium grain model the performance can be improved by up to 44% by going from $BW = 2$ to $BW = \infty$ when the number of processors is infinite.
- Comparing the two algorithms, BBA has larger clusters and can therefore benefit the most from a higher BW as can be seen from the Δ figures in the table.

Network Latency. In Table 3.7, we see the effect of varying the network latency from 0 to 30. Δ is computed with respect to $L = 0$. The observations are:

- The average speedup S_f increases as the latency L is increased indicating that the medium grain model is better able to tolerate network latency than fine grain. This is due to the fact that in the medium grain execution fewer tokens are transmitted and therefore the latency has less effect on the total execution time.
- For a small number of processors (< 20), the network latency does not play a major factor: there is enough work to keep most, or all, processors busy. However as the number of processors increases, the effect is more pronounced, with the fine grain model faring much worse than the medium grain (increase in S_f).
- With $P = \infty$, $L = 0$ and $BW = 2$, the BBA code does worse, on the average, than the fine grain code. The cause of this phenomenon is explained in the next section.
- In comparing the DSA and BBA, the speedup improvement when L goes from zero to 30 ranges from 9% to 74% for DSA, while the improvement is even better for BBA, ranging from 9% to 142%. The figures show that with relatively low latencies, speedup decreases with increasing number of processors for BBA while the opposite is true for DSA. However, when the latency is 30, speedup increases with increasing number of processors for both algorithms.
- From the values of Δ we see that the DSA is more susceptible to an increase in network latency than the BBA.

	DSA $P =$			*BBA* $P =$		
SC	1	100	∞	1	100	∞
low: (2,1)	1.47	1.65	1.67	1.77	1.40	1.34
hi: (5,4)	1.59	1.76	1.79	2.08	1.76	1.51
Δ (%)	131	47	23	119	22	16

Δ wrt $SC = (2, 1)$; $ILP = 2$; $BW = 2$; $L = 5$

Table 3.8: Effects of Different Synchronization Cost

		DSA $P =$			*BBA* $P =$		
	ILP	1	100	∞	1	100	∞
S_f	1	1.28	1.56	1.61	1.43	1.17	1.14
	2	1.47	1.65	1.67	1.77	1.40	1.34
	3	1.48	1.66	1.68	1.82	1.47	1.40
	4	1.48	1.65	1.68	1.83	1.50	1.43
	∞	1.49	1.67	1.69	1.85	1.56	1.49
Δ (%)	1	17	8	4	29	38	37
	2	1	2	2	3	13	13
	3	0	1	1	1	7	8
	4	0	1	1	1	5	5

Δ wrt $ILP = 0$; $SC =$ (2,1); $BW = 2$; $L = 5$

Table 3.9: Effect of Processor ILP

Synchronization Cost. In Table 3.8, we see the effects of varying the synchronization cost from $(2, 1)$ to $(5, 4)$. The first two rows show the average speedup S_f over the fine grain and the last row (Δ) is the degradation factor of the high matching cost with respect to the low one. As can be expected, as the cost of matching goes up, performance of fine grain execution degrades faster than the medium grain execution.

The effect of a larger synchronization cost SC on the execution time diminish with an increase in the number of processors. This is due to the fact that as the number of processors increases, more parallelism is exploited at run time and therefore fewer clusters contribute to the dynamic (run time) critical path length of the program.

Also as in the network latency case, BBA does better than DSA at tolerating higher synchronization cost. For a low number of processor (< 10), the high match-cost model does more than twice as bad as the low match-cost model. However, with a large number of processors, the high match-cost model does only about 16% worse than the low match-cost.

Instruction Level Parallelism of Processors. The effects of varying the ILP of processors on both the average speedup S_f and the percent degradation Δ, with respect to $ILP = \infty$, is shown in Table 3.9.

The speedup increases as the ILP increases since only the medium grain execution could benefit from the higher ILP. For both algorithms, an ILP of two is sufficient to provide 87 to 99% of maximal performance. This can be achieved with a commercial superscalar processor. The data shows also that the BBA exhibits a larger degree of intra-cluster parallelism than the DSA. Also, as the number of processors increase, the speedup increases for the DSA while it decreases for the BBA, in accordance with the trend we have observed previously.

3.4.2 Comments

DSA	$ILP = 2$ $BW = 2$	$ILP = \infty$ $BW = 2$	$ILP = 2$ $BW = \infty$	$ILP = \infty$ $BW = \infty$
No Optimization	756	756	698	698
Merge Up (MU)	2142	1393	825	689
Merge Up and Down (MUD)	2019	1331	673	530

$P = \infty$; $L = 0$

Table 3.10: Execution times of *Ricard*

It can be observed that for some benchmarks and certain machine models, the medium grain execution performs worse than its fine grain counterpart. There are two distinct, but related, reasons for this:

1. **The increased latency, which is a direct result of strict execution semantics of clusters**. In the fine grain model, an instruction can execute as soon as its operands are available, therefore latency can be masked by its finer granularity with at most two instructions waiting for its result at any one time. In the cluster model, even if an instruction within a cluster could execute, it is not enabled until all input tokens to the cluster are available. The larger clusters of BBA are especially prone to this effect and are why BBA is susceptible to poor speedup when the number of processors is large especially under extreme conditions (e.g., zero network latency). DSA by its construction method prevents this type of anomaly.

2. **The loss of pipelining**. The most striking example of this phenomenon is when the whole body of a sequential for-init loop becomes a single cluster. The strict semantics of clusters forces a fully sequential execution of the loop and prevents any potential pipelined overlap among successive loop iterations. This problem is more likely to occur in the BBA than in the DSA.

The optimization steps (merge up and merge down), used by both algorithms, can also unwittingly create these problems: in some cases, the merge down process can significantly increase the input latency of a cluster (i.e., the time between the arrivals of the first and last tokens) in effect removing any overlap or pipelining among successive clusters. The merge up optimization can also worsen performance by taxing the processor's output bandwidth and ILP beyond its limits. Several benchmarks fall in this category and would perform much better, on some machine models, when the optimizations are turned off.

Example. A benchmark that illustrates the optimizations exhibiting the above problems is Ricard (a numerical program). In table 3.10, we show the medium grain execution times with: no optimizations, merge up only (MU), and merge up and down (MUD). The clusters are built using the DSA. Two processor parameters, BW and ILP, are both varied from 2 to ∞. Note that the merge up process does not increase the input latency of clusters, it does, however, increase the demand on the processor in terms of ILP and BW. From this data we observe that:

- With $ILP = 2$ and $BW = 2$, the non-optimized case does about 3 times better than fully-merged case.

- The non-optimized case is minimally affected by increases in ILP and BW implying that it is not ILP or output bandwidth limited.

- In both MU and MUD cases, as we increase the ILP to ∞, speed improves significantly. The speed improvement is even more dramatic as we increase the output bandwidth from $BW = 2$ to $BW = \infty$. Thus, we observe that for this benchmark the optimized cases are significantly affected by both output bandwidth and the degree of ILP of a processor. In fact, we can observe that for infinite ILP and BW, as shown in the last column, the optimized cases does better than the non-optimized case.

This example illustrates that making bigger clusters is not always a good idea when the output bandwidth and processor ILP are limited.

These results leads to the following observations:

- The goal of minimizing the number of synchronizations must be pursued alongside the goal of minimizing input latencies, otherwise the effect of increase in latencies could dominate the performance, thereby causing poor speedup.
- The code generation strategy should take into account the limitations of both the output bandwidth and the ILP of processors. A performance bottleneck could result when a cluster puts too much demand on the ILP or bandwidth of a processor. This suggests that it may help to split the cluster into two or more clusters when it is necessary to reduce the ILP and bandwidth demands, in other words, shift the burden from within one processor to across multiple processors.

3.4.3 Discussion and Summary

Medium vs Fine Grain. Clusters generated by the two algorithms contain, on the average, 2 to 4 instructions and have low internal parallelism. It is still possible, however, to get 50% to 100% speedup over the fine grain execution. This results, mainly, from savings in synchronization cost due to static scheduling within clusters. This speedup was achieved without sacrificing the available function and loop parallelism. Furthermore, this speedup over the fine grain was achieved despite the fact that the current algorithms are not specifically concerned with minimizing input latency to a cluster and that both merge up and merge down optimizations are always performed. This indicates that in most cases, the decrease in the synchronization cost more than offsets the increase in input latency cost even under limited processor abilities, in general.

Over the whole range of architectures, we found that the medium grain model tolerates increases in the network latency and synchronization cost better than the fine grain model. It is also as good as, if not better than, fine grain in the ability to scale to larger number of processors. For small number of processors, network latency and output bandwidth have almost no effect on the execution time of the medium grain execution because of the large task queues. However, synchronization cost has a large effect on the execution time, for low number of processors. This is due to the compounding of the synchronization costs along the critical path.

It has been argued by Arvind and Iannucci [AI87] that efficient synchronization and latency masking are the two fundamental issues in multiprocessor design and that the dataflow model can provide both. Culler *et al.* [CSvE93] demonstrate that the ability of a dataflow/multithreaded execution to mask latency is determined to the largest extent by the architecture of the storage hierarchy. Therefore the speed and size of the storage hierarchy becomes another major issue in determining the performance and scalability of a multithreaded machine architecture. The results in this chapter show quantitatively that a medium grain grain execution is not only more efficient at masking latency than a fine grain one but can also afford a higher synchronization cost with minimal loss in performance. Yet another major issue to be concerned is the total volume of inter-processor communcations, hence the total number of synchronization points, incurred by the execution. By exploiting instruction level locality statically, the medium grain execution model can reduce the communication and synchronization overhead and therefore better tolerate latency.

DSA vs BBA. Of the two partitioning algorithms, we note that the dependence sets algorithm is more consistent in its performance than the basic block algorithm over a range of architectures. However, on some benchmarks, BBA performs significantly better than DSA. These results are due to more conservative partitioning strategy of DSA. The poor partitioning choices that BBA sometimes makes can dramatically increase input latencies of clusters. When the available number of processors is large and therefore the program execution time is shorter, the effects of these large latencies become noticeable and performance deteriorates. This accounts for the fact that speedup of BBA goes down as the number of processors increases, as the fine-grain model is not hampered by input latency. In contrast, the input latency does not increase in DSA clusters as they are built starting from one input node, and the overall speedup rises as result tokens are generated at least as early in the fine-grain model. Hence, the basic block algorithm could benefit the most from compile time analysis of input latency of clusters.

We are thus led to a major limitation of the pure bottom-up approach to generating medium grain from fine grain code: the information about input latency of clusters is not readily available without some extensive graph analysis and may sometimes be impossible to obtain. It amounts to reconstructing high level program constructs from low level code. A better approach would be to combine top-down thread generation with some bottom-up optimizations. This issue is tackled in the next chapter.

Architecture Models. Our evaluation indicates that the performance degradation is lower for the medium-grain execution than the fine-grain execution as the cost of synchronizations goes higher. In fact, for a large number of processors, the difference between the high and low costs models was only 16% in the medium-grain executions. This result indicates that a software based synchronization, such as the inlets in TAM [CSS+91], is likely to perform as well as a hardware based one when the number of processors is large. However, for a smaller number of processors, or equivalently, a large problem size, the synchronization cost is an important determinant of the overall performance. The subject of synchronizations, along with the techniques to reduce its cost, is studied in Chapter 5.

The medium grain execution can benefit from higher output bandwidth. One of the main reasons behind this is the extensive use of *iterative instructions* [BG90] in the Manchester dataflow code which are stream producing instructions. Even though they produce at most one result per clock cycle, when several of them are combined in a single cluster they put a large demand on the output bandwidth. This can happen, in particular, in the range generation code of *forall* loops. Since physical limitations make it impossible to satisfy a large demand on the output bandwidth, a possible future optimization would be to limit the number of iterative instructions within a single cluster.

An instruction issue rate of two per processor is sufficient to handle most of the ILP present within a cluster. This means that a simple superscalar processor would be sufficient to efficiently execute most clusters. This does not, however, preclude the use of a larger instruction word machine since in this study we did not perform any loop unrolling or software pipelining on the code.

Chapter 4

HYBRID THREAD GENERATION AND EVALUATION OF MULTITHREADED MODEL

In the previous chaper, the benefits of medium-grain executions over the fine-grain executions was quantitatively demonstrated. The bottom-up code generation schemes for generating clusters work quite well. However, the limitation of the bottom-up schemes remains, namely the limited size of clusters and the lack of knowledge of the program structures. In this chapter, a hybrid code generation scheme that makes use of both the top-down and bottom-up techniques is described that addresses the above limitations. In addition, a concrete multithreaded execution model, which is an enhancement of the previous, generic medium-grain hybrid execution model is described in detail, along with the target instruction set architecture.

The purpose of describing the multithreaded execution model is to create a concrete compilation target; however, many details of architectural issues are parameterized in order to study the effectiveness of each issue. The multithreaded processor affords a wide range of design parameters such as the use of blocking or non-blocking threads, hardware support for synchronization and matching, register files, code and data caches, multiple functional units, direct feedback loops within a processing element, and vector support. A challenge lies in generating code that can effectively utilize the resources of multithreaded machines.

In order to allow a fixed compilation target, some of the parameters, such as the use of blocking or non-blocking threads, are fixed. In a blocking thread model, a thread will block upon issuing a remote memory reference, a function call, or when performing a synchronization operation. This requires the processor architecture to handle suspension and resumption of threads along with saving and restoring their states. On the other hand, non-blocking threads, once started, runs to completions. This requires the support of the instruction set and the code generation scheme to generate threads that cannot block; for example, remote memory reads are turned into split-phase operations.

For non-blocking threads, which is the focus of this thesis, the dominant factor in the effectiveness of a multithreaded machine is the balance between the size of threads that a language and its compiler can provide versus the thread length required by the hardware to hide latencies in thread switching and synchronization. In general, the pure top down approach generates larger threads than the pure bottom up approach. However, by combining the two approaches, the benefits of each can be realized. This is accomplished, in this thesis, by initially generating threads top-down, and then optimizing these threads via a bottom-up method. In the following sections, research on the Pebbles multithreaded execution model and the compilation strategy for the model is presented. A particular emphasis is on the evaluation of the compilation techniques.

4.1 The Pebbles Multithreaded Model

In this section the Pebbles multithreaded execution model and relevant architectural issues, including the instruction set architecture, are described.

4.1.1 Execution Model

Pebbles[1] is a multithreaded execution model based on dynamic dataflow scheduling where each actor, or each node in the dataflow graph, represents a sequentially executing thread. A thread is a statically determined sequence of RISC-style instructions operating on registers. Similar to dataflow, threads are

[1]The name is derived from the execution of medium-grain, i.e. pebble-sized, threads.

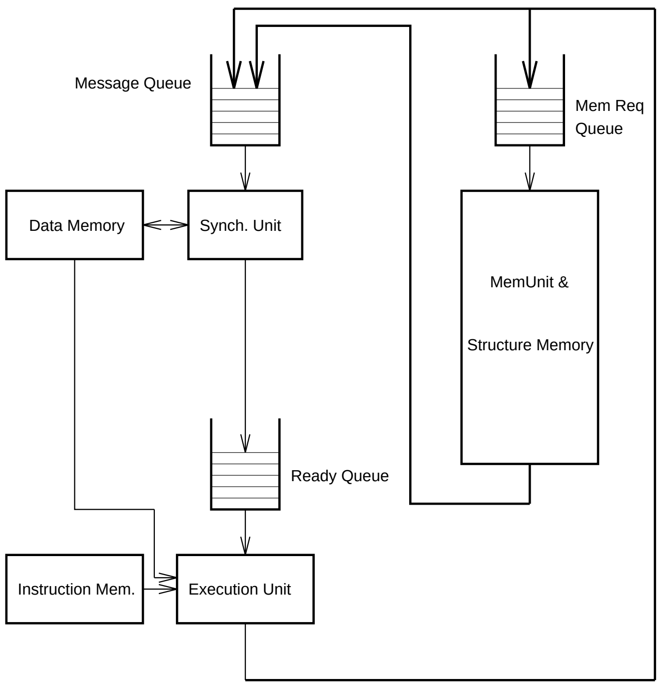


Figure 4.1: Abstract Model of a Processing Node.

dynamically scheduled to execute based upon the availability of data. Once a thread starts executing, it runs to completion without blocking and with a bounded execution time. The bounded execution time implies that each instruction in the threads must have a fixed execution time.

Inputs to a thread are comprised of all data values required to execute the thread to its completion. Hence, register values do not live across threads. A thread is enabled to execute only when *all* the inputs to the thread are available. Multiple instances of a thread can be enabled at the same time and are distinguished from each other by a unique "color." The thread enabling condition is detected by the matching/synchronization mechanism which matches inputs to a particular instance of a thread. Data values are carried by *tokens*. Each token consists of a tag[2], an input port number to the thread and one data value. Data structures, such as arrays and records, are stored in a logically shared structure store. The results of thread execution are either written to the structure store or directly sent to their destination thread(s).

4.1.2 Architectural Model

In this section we describe the baseline architecture model in more concrete terms, which will serve as a basis for our experiments. A Pebble architecture consists of one or more more processing nodes connected by a general, high speed interconnection network. It includes a global, logically shared memory for data structure storage which may be either distributed among the nodes, or among dedicated memory modules arranged in a dancehall configuration.

A Pebbles machine consists of one or more more processing nodes connected by a general, high speed interconnection network. The abstract logical structure of the model is represented in Figure 4.1. The local memory of each node consists of a *Instruction Memory* which is read by the *Execution Unit* and a *Data Memory* (or *Frame Store*) which is mainly accessed by the *Synchronization Unit*. The *Ready Queue* contains the *continuations* representing those threads that are ready to execute. A continuation consists of a pointer to the first instruction of the enabled thread and the context specifier that points to the data needed by the

[2]Tags consist of color, or context, of a thread instance and pointer to the start of thread.

thread. There may be different contexts of the same thread that may be enabled at any given time either on the same node or on different nodes. The global, logically shared memory (*Structure Memory*) for data structure storage may be either distributed among the nodes, or among dedicated memory modules arranged in a dancehall configuration. The *MemUnit* handles the structure memory requests.

For each instance of a thread, a fixed size storage area (called a *framelet*) is allocated in the Frame Store to hold the incoming inputs to that thread. When the first input of a thread activation (i.e. instance) arrives to a node, the Synchronization Unit will first allocate a framelet and set the count of the total number of inputs in the framelet. Each input token is stored in an appropriate slot within the framelet and the counter is decremented. When the count reaches zero, the thread is enabled to execute by making an entry in the Ready Queue. After the thread executes, the framelet is deallocated. In the next chapter, this architectural model is developed further.

4.1.3 MIDC Instruction Set

Pebble programs are represented in a form of dataflow graphs called MIDC (Machine Independent Dataflow Code). Each node of the graph represents a thread of machine independent instructions. Edges represent data paths through which tokens travel. In addition to the nodes and edges, there are pragmas and other specifiers to encode information (e.g. program-level constructs) that maybe helpful to the post-processors and program loaders. The use of low-level intermediate code, in MIDC, stems from the desire to have a degree of architectural freedom but at the same time expose various instruction latencies to the compiler. The objective is to have a reasonable platform on which to evaluate the code generation and Pebble architectures. The MIDC syntax definitions are presented in Table 4.1.

A MIDC program consists of a number of function definitions, one of which is called *main* and communicates with the outside world. A function consists of a header and a body. The function header provides the interface between callers and a callee; and, it consists of a *Function Input Interface* and a *Function Output Interface*, collectively referred as the *Function Interface*. The connection between the function header and its body is shown in Figure 4.2. Call parameters are passed to the function input interface which sends them to the function body. Return contexts are sent from the caller to the function output interface; unique contexts are provided for every function activation. The function output interface matches every context to the corresponding result value and creates *dynamic* arcs back to the caller.

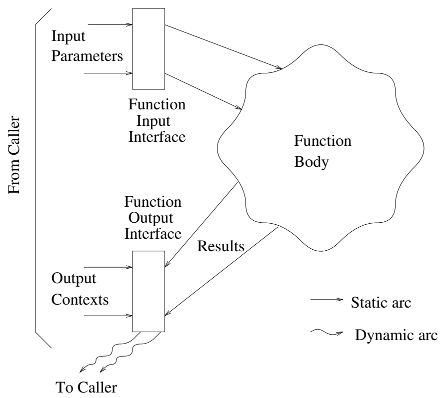


Figure 4.2: Relation between function header and function body

The body of each function consists of a number of nodes. For each node, the *node header* provides a node-label, the number of registers used, the number of input ports, and the destinations of all the outputs. The node header is followed by a stream of instructions. Instruction operands may be node input values, registers or literals. Most instructions are simple RISC-type. The control instructions in MIDC are the `IF-ELSE`, `Do_Loop` and `CALL` operations. Depending on the value of a register, instructions in either the IF-block or ELSE-block are executed. Instructions within the `Do_Loop` boundary are executed n times, and is used to generate inputs for parallel loop bodies. `Do_Loop` replaces the iterative instructions [BGT89] that produce an unbounded stream of values. Every instructions inside the `Do_Loop` have a fixed execution

<table>
<tr><td>Function_Interface:
OutList:</td><td>F FName Node# #ins OutList
< (Destination) ... ></td></tr>
<tr><td>Function_Body:</td><td>Node ...</td></tr>
<tr><td>Node:</td><td>N Node# #regs #ins OutList
Instruction ...</td></tr>
<tr><td>Instruction:</td><td>Target_Register = Operation
| Operation
| Conditional
| Do_Loop
| Func_Call</td></tr>
<tr><td>Conditional:</td><td>IF Operand_Register
Instruction ...
ELSE
Instruction ...
END_IF</td></tr>
<tr><td>Do_Loop:</td><td>Target_Register = DO Start Stop Stride
Instruction ...
END_DO</td></tr>
<tr><td>Func_Call:</td><td>CALL
Instruction ...
END_CALL</td></tr>
<tr><td>Target_Register:
Operation:
Operands:
Operand:
Operand_Register:
Literal:</td><td>$R_\# \mid V_\# \mid O_\#$
Operator Operands
Operand | Operand , Operands
Operand_Register | Literal
$V_\# \mid R_\# \mid I_\#$
"Type Value"</td></tr>
<tr><td>Destination:</td><td>node# port#</td></tr>
<tr><td>#: number or number of</td><td>...: a sequence of the previous construct</td></tr>
</table>

Table 4.1: Basic MIDC Syntax Definitions

time. The `CALL` operation provides an easy way of encapsulating all the instructions that are involved in the generation of a function call.

The basic data types used are integer, character, real, double and pointer. Pointers can point to any of the basic types and aggregate structures such as records, unions and arrays. Recursive data types are allowed via multidimensional arrays, records and unions. The differentiation between the data types pointed to by any pointer is done statically by the compiler. Arrays are represented by a pointer to an *array descriptor*, shown in Figure 4.3-a. An array descriptor consists of four fields, viz., a data pointer to the start of the array, the array lower bound, the shift value, and the array size. The shift value is used to distinguish among several concatenated logical arrays that are "built-in-place." Records are represented by a pointer to the first field of the record, shown in Figure 4.3-b. Unions are represented by a pointer to a tag followed by a value field (see Figure 4.3-c). Union elements may be either basic types, records, or arrays.

4.1.4 Comparison to Other Works

Pebble differs from TAM [CSS+91] in that TAM exposes all synchronizations to the compiler. Therefore, a TAM compiler generates explicit code to perform synchronizations and to enable threads. In addition, TAM is based on the *code-block* model where each code-block represents a semantically distinguishable code segment such as a non-nested loop or function body [Ian90]. A storage segment, called *frame*, is allocated for each code-block instance or set of instances (as in k-bounded loops [Cul85]). All the data values pertinent to a given code-block are stored in the corresponding frame, these include: synchronization slots, temporary

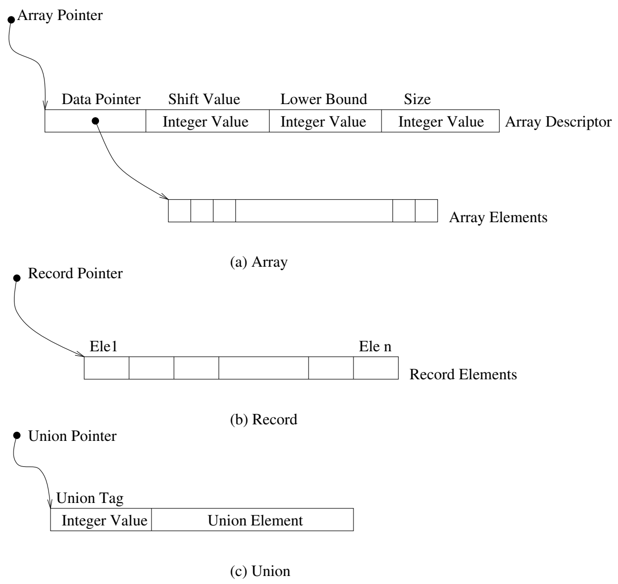


Figure 4.3: Representation of the data structures in memory.

storage area, loop constants, etc. The frame is resident in the frame store until all the threads in the code-block have completed: its lifetime is therefore expected to be longer than that of a framelet.

Unlike the Iannucci's Hybrid model [Ian88], each iteration of a parallel loop instance in Pebbles can be concurrently active on different processors. Another difference from most other dataflow-derived models are that all computations in Pebbles are strict, and therefore Pebbles do not require I-structure type of memory (or software emulation of I-structures) as in TAM, Hybrid, Tera, or Monsoon. Although this reduces the potential exploitable parallelism, the hardware requirement is simpler. Since each thread is relatively small (10 to 30 MIDC instructions), global (dynamic) scheduling and near perfect load balancing is achieved by a simple hashing of the tag (color and the thread pointer) in each token to specific nodes.

4.2 Multi-Threaded Code Generation

In this section, the philosophy and details behind the top down code generation scheme is discussed. This portion of the work is done in collaboration with Bhanu Shankar who developed the top-down code generator.

4.2.1 Code Generation Approach

The goal for most code generation schemes for non-blocking threads, including ours, is to generate as large a thread as possible [Ian88], on the premise that the thread is not going to be too large, due to several constraints imposed by the execution model. The construction of our threads is guided by the following objectives:

1. Minimize synchronization overhead.
2. Maximize intra-thread locality.
3. Assure non-blocking (and deadlock-free) threads
4. Preserve functional and loop parallelism in programs.

The first two objectives call for very large threads that maximize the locality within a thread and decrease the synchronization overhead. The thread size, however, is limited by the last two objectives. Due to the third objective, a thread will typically be much smaller than a loop or function body, as memory

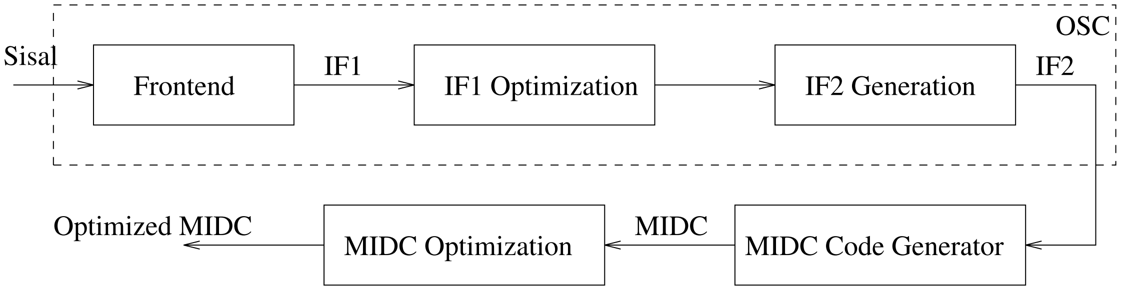


Figure 4.4: Compilation phases of SISAL to MIDC

reads must be turned into split-phase transactions. In addition, even when the non-blocking and parallelism objectives are satisfied, blind efforts to increase the thread size can result in a decrease in overall performance [RNB93]. Larger threads tend to have larger number of inputs which can result in a larger input latency[3]. The resulting MIDC code should exploit many forms of parallelism including parallel loops and functions.

4.2.2 Top Down Code Generation

A compiler that generates MIDC code from Sisal programs is described. Figure 4.4 shows various phases of the compilation process. Sisal [MSA+85] is a pure, first order, functional programming language with loops and arrays. Sisal programs are initially compiled into a functional, block-structured, acyclic, data dependence graph form IF1 [SG85b] which closely follows the source code. The functional semantics of IF1 prohibits the expression of copy-avoiding optimizations. The frontend generates very simplistic IF1 graphs. Function inlining and other optimizations such as loop fusion and some amount of common subexpression elimination are performed on the generated IF1 as part of the second phase. The optimized IF1 is in turn compiled into IF2.

IF2 [WSYR86], an extension of IF1, allows operations that explicitly allocate and manipulate memory in a machine independent way through the use of buffers. A buffer comprises of a buffer pointer into a contiguous block of memory and an element descriptor that defines the constituent type. All scalar values are operated by value and therefore copied to wherever they are needed. On the other hand, all of the fanout edges of a structured type are assumed to reference the same buffer; that is, each edge is not assumed to represent a distinct copy of the data. IF2 edges are decorated with pragmas to indicate when an operation such as "update-in-place" can be done safely, which dramatically improves the run time performance of the system.

The top down cluster generation process then transforms IF2 into MIDC. The first step in the IF2 to MIDC translation process is the *node rewriting* phase to remove any hidden latencies represented in a single node. For instance, the `AElement` node in IF2 represents reading of a value from an array, with an array descriptor and an index value being the inputs to this node. The data representation in MIDC represents the array as two different structures, the *array descriptor* and the *array data.* In order to read an array element, the array descriptor is first read to find the start address of the data elements and then the actual data value is read. This represents two different latencies in the operation. In order to make the latencies explicit, the `AElement` node is rewritten as two distinct nodes, a non-standard IF2 node `AReadDesc` and the `AElement` node. All nodes that contain more than one latency are rewritten to make the latencies explicit.

At the end of the IF2 compilation phase, most but not all of the memory copy operations are eliminated. Fortunately, IF2 edges have pragmas that indicate whether or not the data is built in place. The necessary edges have to be checked to see if the data is built in place or not. If the array is not built in place, code has to be introduced to perform the necessary copy operations. It is easiest to add the copy operations at

[3]Input latency, in this paper, refers to the time delay between the arrival of the first token to a thread instance and that of the last token, at which time the thread can start executing [NMB92]

this time rather than during code generation. It is recognized that the copy operations are independent of each other when introducing the copy code.

In this process of rewriting different nodes, redundant nodes will be generated. The use of classical compiler optimizations such as local common subexpression elimination removes the redundant operations. For instance, if there exist two `AElement` nodes that read elements from the same array, they will independently be rewritten as two `AReadDesc` and `AElement` nodes during the rewriting phase. Since the array being read is the same, the two `AReadDesc` nodes can be merged into one. Thus, redundant memory operations are eliminated. In a similar vein, if the newly introduced nodes are within loop bodies and are identified as loop invariant, they can be easily moved out of the loop body. Thus, this code motion should improve the run-time performance.

When parallel loops are considered, array gather operations can be performed within the loop body, thereby reducing the amount of code necessary to perform the synchronization and thus reducing the amount of sequential code. Thus, in this case, moving the array gather operation from the *returns* code to the body is performed.

The second step in the IF2 to MIDC translation process is the *graph analysis and partitioning* phase. This phase breaks up the complex IF2 graphs so that threads can be generated. Initial values for reduction operators are generated in the appropriate threads. Threads terminate at control graph interfaces for loops and conditionals[4], and at nodes for which the execution time is not statically determinable, such as function calls and memory (arrays and structures) accesses. These nodes are called *terminal nodes.* More accurately, it terminates at the *use* of the values returned from the function calls and memory reads. Therefore, multiple calls and memory reads may be initiated in a thread. Structure store reads are turned into split-phase reads, with the initiator and consumer residing in different threads. In the case of function calls and memory reads, the above step ensures that threads can execute deterministically, in keeping with our objectives. Terminal nodes are identified and the IF2 graphs are partitioned along this seam.

The code motion optimization described above helps reduce the number of threads generated (and executed). Consider a case where code motion was not performed due to the presence of an additional latency in loop bodies, then this would reduce the thread size and generate an additional thread. This results in larger memory traffic, due to the presence of the memory latency operation in the loop body. When code motion is performed, the latency is moved out of the loop body and can be coalesced with a thread in the outer graph. The number of threads needed to represent the loop body thus reduces by one, with fewer number of tokens generated and matched.

The next several paragraphs describe the translation scheme for complex nodes in IF2.

Call Node: Function interfaces are generated for those functions that have survived function inlining. When a function call is encountered, the code to connect the call site to the function interface is generated, where the input values and return contexts are given a new activation name. The return contexts combine the caller's activation name and the return destination and are tagged (as are the input values) with the new activation name. The function interface takes the responsibility of creating the dynamic arcs required to return the result of the function. Threads terminate at function calls.

Select Node: The `Select` node encodes the `if-then-else` construct. A `select` node consists of three types of sub-graphs, the `selector`, the `else` and the `then` subgraphs. The `selector` graph is assigned the responsibility of directing the tokens to the correct subgraph. It has a single input which evaluates `true` or `false` by which tokens are directed.

Iterative Loops: Iterative loops with loop carried dependences and termination tests, contain four subgraphs: `initializer, test, body` and `returns`. The returns graph produces the results of the loop. The subgraphs are wired up sequentially, such that activation names and other resources for these loops can be

[4] The control graph interfaces are utilized in two ways, to control the flow of tokens from one thread to another as well as to perform implicit merge operations. At run time it is not possible to predict the thread tokens will flow to, thus it is not possible to merge the producer and consumer threads. Implicit merges are utlized to eliminate redundant tokens and instructions.

kept to a minimum. Of the four subgraphs, i.e. the `initializer, test` and the `returns` will consist of exactly one thread each[5]. The `body` subgraph can consist of many threads and can consist of compound structures. The `returns` consists of reduce and synchronization operations. Intermediate results and array pointers have to be recirculated back to itself. Recirculation of values and data pointers is made necessary due to the absence of call frames in the MIDC model. The sequential wiring is done to trade-off parallelism with resource utilization. When the cost of this trade-off becomes clear, we may revisit this implementation to increase parallelism.

Parallel Loops: Forall loops with data independent body graphs and a known loop count consist of a `generator`, a `body`, and a `returns` graph. The `generator` is responsible for starting up all instances of the `body` and providing the `returns` with the initial values allowing it to produce its result. The `generator` graph is implemented with the `Do_Loop`. The wiring of the loop itself is done such that all possible parallelism[6] can be exploited. To that end, all ordering in the reduction operators has been removed. This is valid as Sisal reduction operators, such as *sum, product, least,* and *greatest* are commutative as well as associative. There is a cost associated with out-of-order reductions, as extra instructions are needed to control the execution. Results from each loop iteration are sent to the `Returns` thread. The `Returns` thread keeps count of the number of times it executes and exits after the count reaches zero. Intermediate results for all reduction operations have to be recirculated within the thread.

4.3 Bottom-Up Multithreaded Code Optimization

MIDC represents low level intermediate code from which machine code for a target machine can be relatively easily generated. It also contains several pieces of data, including structural level information, represented via pragmas that are useful to target machine code generator or postprocessors. Even though an impressive set of optimizations are performed at the IF1 and IF2 levels [Can89] including function in-lining, loop transformations, CSE, and update-in-place, and that MIDC compiler also does some optimizations, thread generation creates opportunities for more optimizations that were not visible before.

We have applied these optimizations at both the intra-thread and inter-thread levels.

Local Optimizations. These are the traditional compiler optimizations whose main purpose is to reduce the number of instructions *within* a thread.

- Local dead code elimination eliminates instructions whose results are not used.
- Constant folding/copy propagation removes unnecessary data movement.
- Redundant instruction elimination removes uncessary duplication of work.

---

[5]According to the semantics of SISAL, the `initializer` subgraph should be treated as the first iteration of the iterative loop, and thus cannot be merged with any subsequent thread of the loop. Since iterative loops are of two forms, `while-do` loops and `repeat-while` loops, the `test` and the `returns` threads are executed in differing orders, i.e., for the `while-do` loop, the `test` is executed before the `returns` and vice versa for the `repeat-while` loop. In the case of the `while-do` loop the `test` thread must be accessible to the `initializer` and the `body` threads and in the case of the `repeat-while` loop the `returns` threads must be accessible from the `initializer` and the `body`. This is due to the manner in which termination tests are performed for the two varieties of the loops. Thus, it was decided to let the `initializer`, the `test` and the `returns` occupy distinct threads.

Note that this is the code generated by the compiler. In some instances it is conceivable that the optimizations would merge some or all of these threads.

[6]There is the possibility that the loop body is too small, making it expensive to exploit all possible parallelism. The application of two different techniques, *slicing* and *chunking*, helps reduce this problem. Slicing is a method by which parallelism is constrained by spawning out a variable size of work among a fixed number of workers. Chunking, on the other hand, spawns out a fixed amount of work over a varible number of workers. In both the techniques, the slice or chunk of work is performed sequentially. The application of these techniques is beyond the scope of this paper.

The need for local optimizations arise when threads are first generated (there are some dead code and copy propagation opportunities that arise after the initial thread generation), and after global optimizations are performed (e.g. after merging).

Global Optimizations. The objectives of these optimizations are threefold:

1. Reduce the amount of data communication among threads.
2. Increase the thread size without limiting inter-thread parallelism.
3. Reduce the total number of threads and possibly the critical path lengths of programs.

These inter-procedural optimizations consist of:

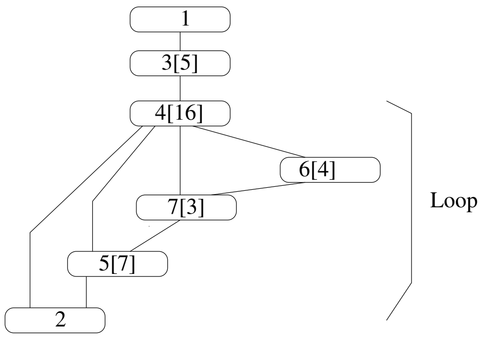
- *Global copy propagation/constant folding:* reduces unnecessary token traffic by bypassing intermediate threads. For example, if Thread A sends a value to Thread B which in turn passes it on to Thread C without using it, then the code is rewired so that Thread A sends the value directly to Thread C. Also, if Thread A sends a constant value to Thread B, then change is made so that Thread B uses the constant value directly without having Thread A send the value. This optimization should also reduce the critical path length of programs.
- *Merging:* attempts to form larger threads by combining two neighboring ones while preserving the semantics of strict execution. The merge up/merge down phases have been described in more detail elsewhere [SCvE91, RNB93]. In order to ensure that functional parallelism is preserved and bounded thread execution time is retained, merging is not performed across remote memory access operations, function call interface and parallel loop bodies. However, merging is allowed to take place across branch boundaries. Figure 3.3 shows the two merge operations. The results of these merging operations are the elimination of the tokens passed between the merged threads and the corresponding output instructions, and therefore the synchronization cost is reduced.
- *Redundant arc elimination:* opportunities typically arise after merging or copy propagation when arcs carrying the same data can be eliminated. For example, if Thread A sends the same value to Thread B and Thread C, and Thread B and C are later merged, there is a duplication of data sent.
- *Global dead code and dead edge elimination:* typically applied after other global optimizations which could reduce the arcs or even the entire thread. Since these optimizations are inter-procedural, they can trace through all calls of a given function and eliminate unused arguments and return values and the related computations. For example, if no caller to a given function uses one of its results, then that unused result along with any predecessor which is itself not used in deriving the other results are eliminated. This process can extend past the callers' arguments and beyond.

These optimizations are performed in a bottom-up fashion; that is, the optimizer has the global picture of the entire program graph. The optimizer reads in the MIDC code and first builds a dataflow graph of threads. Each thread is then decoded into a dataflow graph of instructions. Hence, a two-level dataflow graph is built. At the same time, by decoding the pragmas, it builds a second, dual graph whose block structure corresponds to the program source structure with its constituent threads and instructions.

At this point, we have the complete, equivalent dataflow graph that a pure bottom-up compiler would generate but retain the additional top-down structural information. The structural information graph is used to help determine when certain types of optimizations, such as merging, are safe or even advisable to do. When the optimized code is generated, each thread is generated from these mini-dataflow graphs. The global view thereby enables various optimization steps to take place and enable redrawing of "boundaries" to easily remold threads. In addition, when generating threads, instruction scheduling is performed to exploit the instruction level parallelism. This is accomplished by ranking instructions according to dependencies such that dependent instructions are as far from each other as possible. The data and control flow representations allow this to be accomplished relatively easily.

The optimization step also make the following changes to the syntax:

```
define main
type OneDim = array[integer]
function main (list:OneDim returns double_real)
   for I in list
      returns value of sum 1.0d0/double_real(I) when I ~= 0
   end for
end function
```



n[m] = { thread # = n, number of instructions in thread = m }

```
F main 1 1 0 <(3 1) >

F main 2 3 0 < >
IN 1 3

N 3 1 1 < >
R1 = EAN
RSS.I I1 "3" "4:1" R1
RSS.I I1 "2" "4:2" R1
RSS.I I1 "1" "4:4" R1
RSS.A I1 R0 "4:3" R1
```

```
N 4 8 4 <(2 1 0)(5 3 0)(5 1 0)
         (6 4 0)(6 1 0)(6 2 0)
         (6 3 0)(7 2 0) >
R1 = EAN
R2 = ADD.I I1 I2
R3 = SUB.I R2 "1"
R4 = LE.I I2 R3
IF R4
 02 = OUT "0.0D0" R1
 R6 = SUB.I R3 I2
 R7 = ADD.I R6 "1"
 03 = OUT R7 R1
 R8 = DO I2 R3 "1"
  R5 = GAN
  04 = OUT I3 R5
  05 = OUT I4 R5
  06 = OUT I2 R5
  07 = OUT R8 R5
  08 = OUT R1 R5
 END_DO
 ELSE
  01 = OUT "0.0D0" R1
END_IF
```

```
N 5 4 3 <(2 1 0)(5 3 0)(5 1 0) >
R1 = EAN
R2 = SUB.I I1 "1"
R3 = EQ.I R2 R0
R4 = ADDC.D I2 I3 I2
IF R3
 01 = OUT R4 R1
ELSE
 02 = OUT R4 R1
 03 = OUT R2 R1
END_IF

N 6 3 4 < >
R1 = EAN
R2 = SUB.I I1 I2
R3 = ADD.I I3 R2
RSS.I I4 R3 "7:1" R1

N 7 2 2 <(5 2 3) >
R1 = ITD I1
R2 = DIV.D "1.0d0" R1
01 = OUT R2 I2
```

Figure 4.5: Sisal and MIDC thread code for Purdue benchmark 4

- The top down generator does not allow branches within a thread. Conditional outputs (OUTC) are used as gateways to branches. Since merge operations could introduce branches in the middle of a thread, we introduce if-then-else operator. In the process, conditional outputs are eliminated.
- The code generator uses several operators that output results (e.g. PARAM, OUT, SAN, DST) that add descriptive power. These operators can either explicitly specify a color value or implicitly assume the color of the incoming color. Their descriptive power have served their usefulness at this point, so they are all replaced by a single OUT instruction and the color value is given explicitly.

A simple Sisal source and its compiled, optimized MIDC code is shown in Figures 4.3 and 4.5. The code is taken from Purdue Benchmark 4 which inverts each nonzero element of an array and sums them. Figure 4.3 shows the thread level graph descriptions. Nodes 1 and 2 are the main function input and output interfaces, respectively. Node 3 reads the structure pointer and size information. Node 4 is the loop initializer/generator. Nodes 5, 6, and 7 comprise the loop body with Node 5 handling the reduction. reduction.

4.4 Performance Evaluation

In this section we evaluate the dynamic properties of our top-down code before and after applying various bottom-up optimizations. We have obtained these results by running codes on the Bedrock[7] machine simulator. The following set of dynamic parameters are measured to evaluate the intra-thread characteristics:

- S_1*:* measures the average number of MIDC instructions executed in a thread.
- Π*:* measures the average internal parallelism of threads ($\Pi = \frac{S_1}{S_\infty}$ where S_∞ denotes the critical path length of a thread in terms of MIDC instructions).
- *Inputs:* measures the average number of inputs to threads.
- *MPI:* measures the average number of matches (each input arrival constitutes a synchronization) per instruction. It is computed by dividing the total number of inputs received by the total number of instructions executed.

We have also measured the following inter-thread (program-wide) parameters:

- *Matches:* measures the total number of matches performed.
- *Instructions:* measures the total number of instructions executed.
- *Threads:* measures the total number of threads executed.
- *CPL:* measures the critical path length of programs in terms of threads.

The values of these inter-thread parameters are presented in normalized form with respect to the top-down generated code.

For evaluation purpose, we use the following optimization levels:

- "No optimization" (*NO*) is the code generated by the MIDC code generator.
- "Local optimization" (*LO*) performs the intra-thread only optimizations.
- "Full optimization" (*FO*) performs both intra-thread and global optimizations.

4.4.1 Benchmarks

For all our experiments, we use a suite of benchmarks including seven nontrivial programs all written in Sisal. *PURDUE benchmarks* [Ric85] consist of 16 programs used to benchmark parallel computer systems. *Lawrence Livermore LOOPS* [McM86] consist of 24 loop kernels used in scientific programs. *AMR* is an unsplit integrator taken from an adaptive mesh refinement code at Livermore. *BMK11A* is a particle transport code. *RICARD* is a production code simulating elution patterns of proteins and ligands in a column of gel. *HILBERT* computes the condition number for Hilbert matrix coefficients using Linpack routines. *LIFE* is the game of life. *SGA* uses a genetic algorithm to find a local minimum of a bowl-shaped function. *SIMPLE* is a Lagrangian 2-D hydrodynamics code that simulates the behavior of fluid in a sphere. The size of the nontrivial benchmarks ranges from 300 to over 2000 lines of Sisal code. LIFE and SGA form the nonscientific codes. The figures for the Purdue and Livermore Loops are each given as a weighted average values of their constituent programs.

The amount of parallelism available in each benchmark can be gleaned from Table 4.2. It shows the number of total loops and the number of parallel forall loops in the generated code[8]. They show that a substantial fraction of loops are forall loops. This pattern is what we normally expect from a well-written

[7] Bedrock is a cycle-level, discrete event Pebbles machine simulator with configurable parameters.

[8] The number of parallel loops in the generated code will differ from the SISAL source due to the fact that additional parallel loops are inserted in the code to perform the task of copying when needed and the fact that some loops would be fused in the process of optimizing the intermediate code

Program	Total Loops	Forall Loops	Threads Executed
PURDUE	86	74	826,343
LOOPS	48	27	1,323,233
AMR	79	2	284,322
BMK11A	18	9	620,945
RICARD	12	10	539,374
HILBERT	73	65	922,798
LIFE	11	6	581,736
SGA	13	7	1,546,145
SIMPLE	88	78	1,419,463

Table 4.2: Program Characteristics of the Benchmarks.

	S_1			Π			*Input*		*MPI*		
Optims.	*NO*	*LO*	*FO*	*NO*	*LO*	*FO*	*NO/LO*	*FO*	*NO*	*LO*	*FO*
PURDUE	12.5	11.9	11.1	4.44	4.24	3.67	5.78	4.45	0.46	0.49	0.40
LOOP	22.1	21.0	19.7	7.08	6.68	5.48	12.2	9.63	0.51	0.56	0.47
AMR	23.0	22.3	23.9	4.29	4.44	4.34	12.6	11.6	0.55	0.57	0.49
BMK11A	28.7	23.9	29.7	4.00	4.28	4.38	15.6	15.3	0.54	0.58	0.52
RICARD	24.6	24.1	20.6	3.56	3.60	3.08	9.88	6.81	0.40	0.41	0.33
HILBERT	9.22	8.96	7.38	2.81	2.87	2.42	5.38	3.75	0.58	0.60	0.51
LIFE	10.5	10.2	10.7	2.81	2.78	2.51	4.86	4.64	0.46	0.48	0.43
SGA	9.97	9.44	10.5	3.10	3.14	3.16	5.77	5.54	0.58	0.61	0.53
SIMPLE	13.7	13.0	11.9	3.54	3.65	3.25	8.02	6.37	0.59	0.62	0.54
Average	17.1	16.1	16.2	3.96	3.96	3.59	8.90	7.57	0.52	0.55	0.47

Table 4.3: Intra-Thread Characteristics

Sisal programs. In other words, it is expected that the programmer will use the parallel loop construct whenever possible and revert to the sequential loop construct if and only if the parallel loop construct cannot be utilized. The last column in Table 4.2 shows the number of threads being executed per run in fully optimized cases.

4.4.2 Intra-thread Results

Table 4.3 shows the intra-thread characteristics of each benchmarks and cumulative results.

Thread Size. The results show a wide range of average thread sizes, ranging from 7.38 instructions in Hilbert to 29.7 instructions in BMK11A with the average of 16 instructions all in fully optimized cases. Among the benchmarks, nonscientific codes have smaller average thread sizes.

There is a steady reduction in S_1 as we go from no optimization to local optimizations. In going from local to full optimizations, there are noticeable differences in each benchmark, but no clear pattern. On the average, the average thread sizes are similar. There are two opposing forces at work: the merge operations tend to increase the thread size while other global optimizations tend to reduce the total number of instructions executed.

The weighted distribution of S_1 across all benchmarks is shown in Figure 4.6 for $S_1 \leq 20$. It shows nearly 30% of threads executing have 5 instructions. The number of threads with $S_1 \geq 20$ account for more than 10% of all threads.

Thread Parallelism. The figures for Π show no clear trend when going from no optimization to local optimization. S_1 decreases in going from no optimization to local optimizations, so this implies that the critical path length (S_∞) of each thread in general are reduced.

However, when going from local to full optimization, the parallelism decreases. In this case, the merging operations play the bigger role in determining the average parallelism they tend to increase S_∞.

In fully optimized cases, average parallelism ranges from about 2.4 to 5.5 with an average of 3.6. The distribution of parallelism is shown in Figure 4.7. It shows that 75% of threads have parallelism less than 3 with about 6% having more than 6. The intra-thread parallelism is large enough to justify a superscalar processor implementation. A four-way issue processor, such as the PowerPC, should be sufficient for almost 90% of all threads executed.

Thread Inputs. For local-only optimizations, the number of inputs per thread does not change. However, there is a significant reduction after global optimizations are performed, as it goes from 8.9 to 7.6 inputs, a drop of 17%. The average number of inputs varies widely between different benchmarks: Hilbert has about 3.8 inputs versus 15.3 in BMK11A. The average number of inputs is 7.6.

The distribution of the number of inputs per thread is shown in Figure 4.8. More than 40% of threads executed have between 2 to 4 inputs. Threads with seven or less inputs account for more than 70% of all executed threads across all benchmarks. However, there are a significant 16% of threads having ten or more inputs.

Matches Per Instruction. For a given optimization level, the matches per instructions are confined within a relatively narrow range for all benchmarks except RICARD. The MPI goes up when local optimizations are applied, since the number of instructions executed goes down while the number of inputs remains the same; MPI goes down by 9 to 21% from no optimization to fully optimized with an average decrease of 11%. This implies that the reduction in the number of matches is greater than the reduction in the number of instructions, resulting in a smaller MPI for the fully optimized code. Among benchmarks, RICARD's MPI is only 0.33 versus SIMPLE's 0.54.

4.4.3 Program-wide Characteristics

In Table 4.4, the inter-thread characteristics of the benchmark programs are shown. The values shown are normalized with respect to the un-optimized code.

Matches. As would be expected, the number of matches remains unchanged when only local optimizations are applied. When global optimizations are also applied, there is a significant reduction in the number of matches performed, typically about 31%. The reductions are remarkably similar for all the benchmarks except LIFE which achieves only a 17% reduction.

Instructions. There is on the average a 4% reduction in the total number of instructions executed after local optimizations are performed. After applying full optimizations, the average reduction is about 23%. The reductions are most noticeable in SGA and smallest in LIFE.

Threads. The number of threads executed remains unchanged for locally optimized code. When full optimizations are applied, there is a 17% reduction in the number of threads on the average. However, there are significant differences among the benchmarks with RICARD and HILBERT having almost no reduction whereas SGA has more than 35% reduction.

Critical Path Length. The characteristics of critical path lengths of programs are very similar to the number of threads executed. The critical path lengths are measured in terms of the number of threads. The path lengths are reduced by about 18% on the average after the global optimizations.

While the CPL is reduced by 18%, the total number of threads is reduced by 17%: this implies that, on average, the program parallelism at the inter-thread level remains relatively the same.

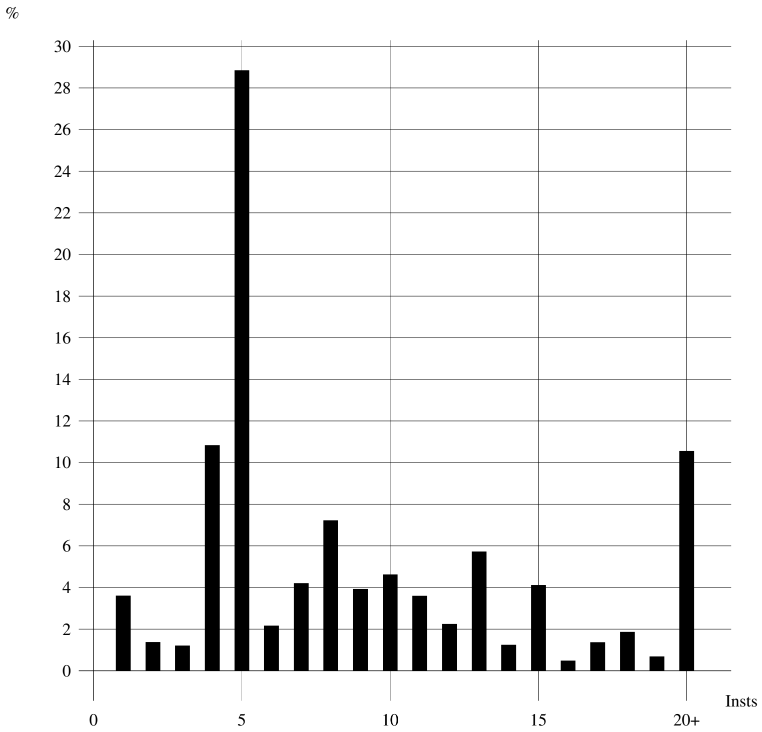


Figure 4.6: Distribution of S_1 across benchmarks (up to $S_1 = 20$)

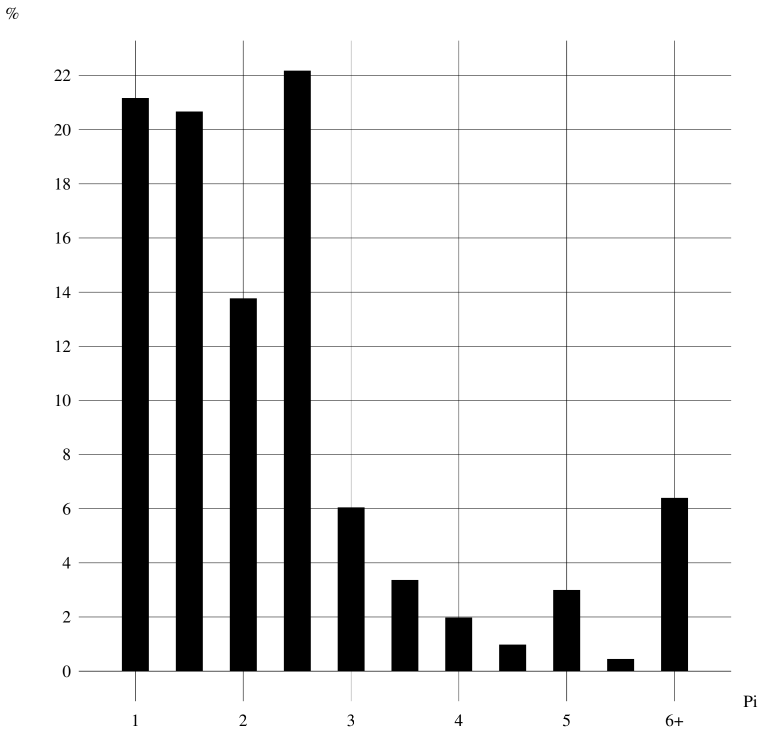


Figure 4.7: Distribution of Π across benchmarks (up to $\Pi = 6$)

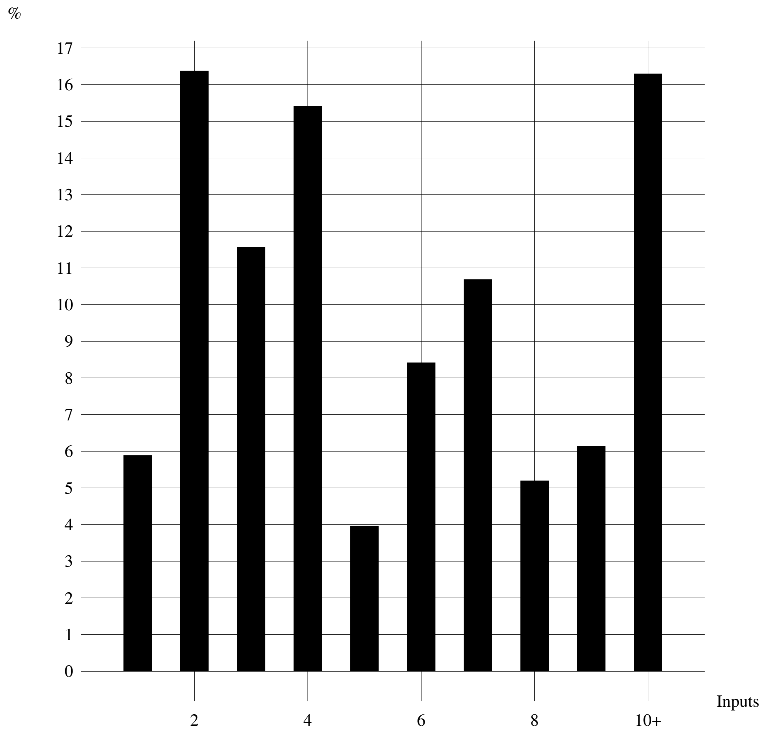


Figure 4.8: Distribution of inputs per thread (up to $Inp = 10$)

	Match	Instruction		Thread	CPL
Optimizations	FO	LO	FO	FO	FO
PURDUE	0.68	0.95	0.78	0.88	0.81
LOOP	0.67	0.94	0.73	0.85	0.77
AMR	0.67	0.97	0.76	0.73	0.90
BMK11A	0.68	0.94	0.72	0.69	0.59
RICARD	0.69	0.98	0.84	0.99	0.99
HILBERT	0.68	0.97	0.78	0.98	0.93
LIFE	0.83	0.97	0.88	0.86	0.79
SGA	0.63	0.95	0.69	0.65	0.76
SIMPLE	0.70	0.95	0.76	0.88	0.87
Average	0.69	0.96	0.77	0.83	0.82

Table 4.4: Program Characteristics (Normalized to Un-optimized Code)

	P=1		*P=10*		*P=∞*	
Optimizations	*LO*	*FO*	*LO*	*FO*	*LO*	*FO*
PURDUE	1.0	17.6	0.9	22.3	0.8	23.6
LOOPS	1.2	32.1	1.2	31.7	0.9	31.8
AMR	1.1	18.4	2.8	22.4	0.7	28.2
BMK11A	0.8	60.0	0.8	74.2	0.9	75.6
RICARD	0.8	7.0	1.0	19.6	0.2	11.5
HILBERT	0.0	11.3	0.2	11.6	0.2	13.0
LIFE	0.4	7.0	0.0	5.0	0.2	25.6
SGA	0.2	36.3	0.2	35.2	0.2	36.4
SIMPLE	0.6	9.4	0.1	17.3	0.5	19.0

Table 4.5: Percent Speedup over Un-optimized Code

4.4.4 Run-time Performance

The results of simulated real time performance applied to different optimization levels are used to measure the effects of these optimizations on the actual overall performance. We assume that all structure store reads and writes are remote, and hence we do not consider the effect of data partitioning.

The following processor architectural configuration is used to get real time measurements: a 4-way issue super-scalar CPU with the instruction latencies of the Motorola 88110 and an output network bandwidth of two tokens per cycle. The synchronization latencies are those of the EM-4: a pipelined synchronization unit with a throughput of one synchronization per cycle and a latency of three cycles on the first input. All inter-node communications take 50 CPU cycles in network transit time. Every structure memory read takes the minimum of two network transits (one to send the request and another to send the reply). Also, the size of matching store is unlimited, and therefore can handle any amount of parallelism. We obtained the results for 1, 10, and infinite number of processors. In each case, the run time performances for both locally and fully optimized codes are compared with respect to the un-optimized code.

As a side note, the speedup of n-processor over 1-processor executions of threaded code in our model is mostly determined by its parallelism. This is due to each thread activation being independent from the others and that a given thread activation can execute on any processor. For example, with 20 parallel threads and 5 processors, each processor executes 4 threads sequentially; with 10 processors, each processor executes 2 threads sequentially, halving the execution time. Hence, we do not further consider this kind of speedups.

Table 4.5 shows the percentage speedup over the un-optimized code. It can be seen that the improvement in speed is significant with almost all improvement occurring when global optimizations are applied. With local optimizations only, the speedup is almost insignificant. In general, with full optimizations, speedup ranges from a low of 5% in LIFE up to 75% in BMK11A. In the case of LIFE, the reduction in the number of matches are much smaller than the others (as shown in Table 4.4) and is a limiting factor with a smaller number of processors. With a larger number of processors, the limitation is removed and the speedup also increases to 25%. In the case of BMK11A, the reduction in the critical path length (CPL) is much greater than the others, resulting in significant improvement in performance.

In general, there seems to be a slight increase in speedup as the number of processors increase. This can be partly explained by the reduction in CPL. With relatively small number of processors, there is plenty of work for each processor to do, and therefore the speedup comes mainly from the reductions in the amount of work to be done. With larger number of processors, CPL plays the limiting role; therefore, the reduction in CPL give rise to even greater speedup. This is especially apparent in RICARD which have almost no reduction in the CPL when full optimizations are applied. Here, the speedup drops with infinite number of processors compared to 10 processors.

Next, the effectiveness of various optimizations are measured. Since local optimizations alone do not give much speedup, we concentrate on the global optimizations. The following symbols are used for the different types of optimizations: FD is fuse down, FU is fuse up, DC is global dead code elimination, CP is global copy propagation/constant folding, and RA is redundant arcs elimination.

Optim:	FD	FU	FD, FU	DC	DC, CP	DC, RA	DC, CP, RA	FO
AMR	7.6	0.9	15.6	2.4	12.1	7.2	14.0	22.4
BMK11A	5.3	6.4	7.4	0.9	68.4	1.3	69.7	74.2
RICARD	8.3	4.8	7.2	0.9	5.9	2.0	6.8	19.6
HILBERT	3.3	3.8	8.9	0.5	1.2	0.5	1.0	11.6
LIFE	0.1	2.0	1.7	0.4	3.8	0.5	3.5	5.0
SGA	4.2	21.4	29.1	3.1	7.4	4.3	8.0	35.2
SIMPLE	6.3	5.9	14.8	1.4	5.6	1.4	5.7	17.3

Table 4.6: Effectiveness of Different Global Optimizations

In Table 4.6, speedups due to various optimizations are given using 10-processor configuration. Some optimizations rely on other optimizations to be performed afterwards; for example, copy propagation must be followed by global dead code elimination that performs some cleanup operations. The result clearly shows that merging optimizations contribute the most toward the speedups. For BMK11A, global copy propagation/constant folding is extremely effective. Its effectiveness for this benchmark explains the sharp drop in the program critical path. In some cases, such as RICARD and HILBERT, performing one optimization creates more opportunities for other optimizations, resulting in bigger total speedups than the sum of individual speedups. In other cases, performing one optimization can remove an opportunity for others; for example, a copy propagation removing the only link between Thread A and Thread B prevents A and B from merging, since they are unconnected now.

4.4.5 Discussion and Summary

Thread Characteristics: Overall, the average size of threads, 16.2, is relatively large compared to some of the values reported (3-5 instructions) for bottom-up thread generation techniques [SCvE91, NRB93]. In particular, the threads in BMK11A are much larger than the other benchmarks.

The internal thread parallelism, 3.59, achieved is large compared with our previous work on the evaluation of the bottom-up Manchester cluster generations which only achieved parallelism of about 1.15-1.20, even considering that Manchester instructions are more CISC-like. This is mainly due to the larger thread sizes of MIDC.

We observe that the number of inputs required are relatively large, on the order of 4 to 10. This implies a need for handling these variable number of inputs efficiently.

Effect of Optimizations: In general, we observe that even though there is some improvement in the various measures when local-only optimizations are applied, the biggest improvement comes from global optimizations. Among global optimizations, the merging operations are particularly effective. The results of the bottom-up optimizations indicate that they have achieved our objectives in code generation:

- Synchronization overhead (e.g. number of matches) has been reduced significantly. The total number of matches has been reduced by 17-37% and MPI has been reduced by about 10%, thus reducing the required synchronization bandwidth.
- Number of inputs to a thread has been reduced by about 15%, thus reducing the input latency.
- Internal parallelism has been reduced by about 10%, meaning that processor requirements are not as large.

While the statistics on intra-thread and program level parameters are important to understand the behavior of the thread generation and the effects of the various optimizations, the bottom-line of any performance measure is the overall program execution time. In this respect, the local optimizations had an insignificant effect while the global optimizations resulted in 20-30% reduction in execution time on the average[9]. In local-only optimizations, the critical path lengths of the program (at the thread level) have not been changed; and, due to the ability to hide latencies, slightly smaller threads do not speed up the execution as much. Therefore, global bottom-up optimizations are essential in achieving a much better performance.

[9] In [RNSB94], we reported higher performance improvements due to optimizations and smaller average thread size (12.5). Our improved code generation scheme at the top down level has reduced the effectiveness of local only optimizations. Also, there, we used a set of relatively small benchmarks that resulted in relatively small thread size

Chapter 5

ANALYSIS OF MULTITHREADING AND COST REDUCTION TECHNIQUES

In the previous chapter, the research on the hybrid code generation strategy for the Pebbles model was described. The results indicated the effectiveness of a bottom-up optimization scheme and also demonstrated that the hybrid approach generates much larger threads than the pure bottom-up approach described in Chapter 3. This chapter tackles important issues that arise in multithreading and analyzes the tradeoffs between code generation and architectural features. Even more importantly, this analysis is used to derive techniques that can be used to either improve the overall performance or lower the cost of multithreading.

The research in this chapter focuses on three areas: synchronization, communication, and parallelism. A common theme running through all three areas involves identifying and exploiting various forms of locality. For the study of synchronizations, the extent to which multithreading can mask synchronization operations is analyzed in detail through simulations. It has been argued that the benefit of multithreading comes at the expense of reduced locality arising from frequent context switching. The reduced locality could undermine the effectiveness of a cache system and the effectiveness of synchronization operations; therefore the locality issue is studied in depth. Based upon these analyses we can design an effective memory hierarchy that reduces the overall synchronization cost.

Conceptually, each thread runs on its own virtual processor, and many virtual processors are multiplexed onto a single physical processor. At any given time, some of these virtual processors are either sending or waiting to receive messages. The higher the degree of multithreading (i.e. the more virtual processors per physical processor), the higher the potential load on the interconnection network. Therefore, multithreading not only increases the processor utilization but also the interconnection network utilization. In fact, the message handling overhead and the network bandwidth limitations can limit the achievable degree of multithreading, and thus limit the ability to tolerate latencies [Cul94]. The second part of this chapter focuses on the nature of inter-processor communications between multithreaded processors, and this is used to develop techniques that reduce the amount of inter-processor communications and the associated overhead.

In dataflow-derived multithreading models such as Pebbles, abundant parallelism can overwhelm the machine resources and reduce locality. Therefore, the management of parallelism is crucial in improving the overall performance and avoiding deadlocks. The third part of this chapter addresses and analyzes the parallelism issues, and offers techniques that attempt to reduce parallelism and also improve the locality. In the following sections, the research on the analysis of multithreading and the techniques that reduce the cost of multithreading is presented.

5.1 Locality and Synchronization Issues

A variety of multithreaded execution models were examined in Chapter 2. Some relied on large threads that can block during their execution while others opted for shorter threads that always execute to completion. Each thread with its own working set would seem to compete against other threads for the cache space, and this may result in increased cache misses, thus negating the benefits of multithreading. Regardless of the model, during each "run" between context switches, the actual working set of that thread can be considered to be small. If the working set is sufficiently small a regular cache design could capture the locality of references within a thread as well as accommodate a large number of threads. At this point, although fine instruction-level locality is captured, the aggregate behavior of threads becomes important since not all threads can be resident in the cache. This raises the issue of *inter-thread level locality*.

In the preceding discussion, it was assumed that effective cache performance is an important measure of overall system performance. However, this assumption needs to be validated and its effect quantified. This is presented in Section 5.1.1 and indicates a need to design an effective storage hierarchy. In Section 5.1.2, a Pebbles storage hierarchy architecture is developed. In Section 5.1.4, the analysis of inter-thread locality and synchronizations are evaluated: the miss rate is measured in a systematic way using different cache sizes and various cache policies, and evaluate the storage hierarchy. In Section 5.1.5, a scheme to reduce the compulsory miss cost is evaluated. The results indicate that: 1) there is a high degree of thread-level locality; 2) relatively simple cache/memory subsystem can capture this locality; 3) and with a small increase in the complexity of the caching mechanism the performance can be improved.

5.1.1 Analysis of the Synchronization Cost

In order to discovery why an effective storage hierarchy is necessary, the effect of synchronization cost on the overall system performance is quantitatively evaluated. The Bedrock simulator is used to perform the evaluation.

Benchmarks. The experiments are conducted using a set of seven Sisal programs: 1) *SGA* is a genetic algorithm program finding a local minima of a bowl-shaped function developed at Colorado State University; 2) *FFT* is a one dimensional complex Fast Fourier Transform code; 3) *PSA* is a parallel scheduler code using a variation of simulated annealing to solve the problem; 4) *SDD* solves an elliptic partial differential equation using the Symmetric Domain Decomposition method; 5) *SIMPLE* is a Lagrangian 2-D hydrodynamics code that simulates the behavior of fluid in a sphere developed at Lawrence Livermore National Laboratory; 6) *AMR* is an unsplit integrator taken from an adaptive mesh refinement code at Lawrence Livermore National Laboratory; 7) *WEATHER* is a one level barotropic weather prediction code and was originally developed at the Royal Melbourne Institute of Technology.

Simulation. The following architectural parameters are specified to the Pebble simulator: a 4-way issue super-scalar CPU execution unit per node with the instruction latencies of the Motorola 88110 microprocessor. An output network bandwidth of two tokens per node is used. For simplicity, all inter-node communications take 50 CPU cycles in network transit time. Every structure memory read takes the minimum of two network transits (one to send the request and another to send the reply). Unless otherwise noted, ten processing nodes are used. The number of processing nodes, ten, was chosen because it provides a realistic processor utilization for the problem sizes that can be run in a reasonable time on our simulator. These architectural parameter values are chosen to give a reasonable approximation of a real machine rather than exact measurements.

Starting with this baseline configuration, the synchronization cost parameter is varied. The cost of synchronizing each token is varied between 0 and 20 cycles. Synchronizing tokens on a mismatch (the first reference to a framelet) incurs 2 additional cycles.

Result. Figure 5.1 shows the plot of the execution times as the synchronization time is varied. The execution times shown are normalized with respect to the base execution time when the synchronization time is ideal (0 cycle). The result shows that for the synchronization time beyond the first few cycles, the execution time grows linearly with respect to the synchronization time. In this linear regime, the synchronization operations are exposed to the multithreaded executions. For the match time less than a few cycles, especially less than 3 cycles, the performance degradation is relatively small compared to the ideal case. In this regime, the synchronization operations are well masked.

Figure 5.2 and Figure 5.3 shows the plots for 5 and 50 processing nodes, respectively. The trend for both is similar to the previous figure: a linear regime exists for synchronization time beyond a few cycles. The difference lies in the slope of the linear regime. It is clear that as the number of processors increases, the slope decreases. This can be explained by the fact that as the number of processors increases, the utilization of various functional units drop, and are thus less able to mask synchronizations. Hence, the effect of increasing the synchronization cost at this point is not as significant as for the case of a smaller number of processors.

For a further insight, Table 5.1 shows the execution time, the processor and synchronization unit utilizations of each benchmark for 10 processing nodes. The processor and synchronization unit utilizations

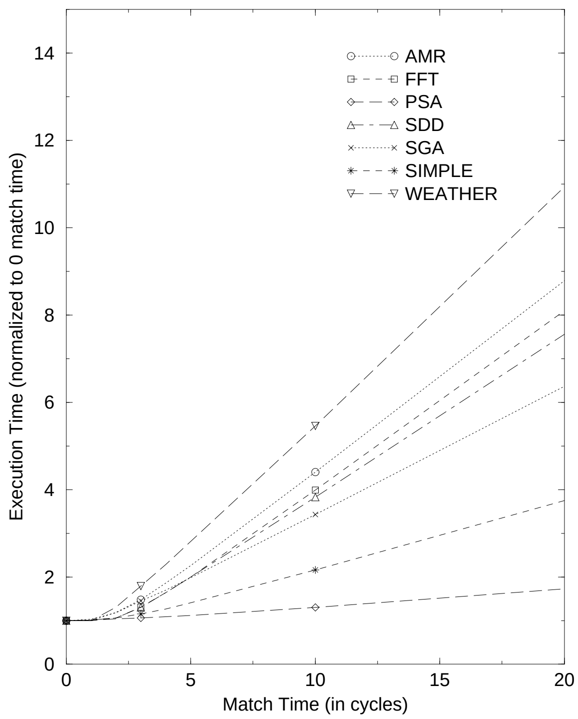


Figure 5.1: Execution Time versus Synchronization Time (10 Nodes).

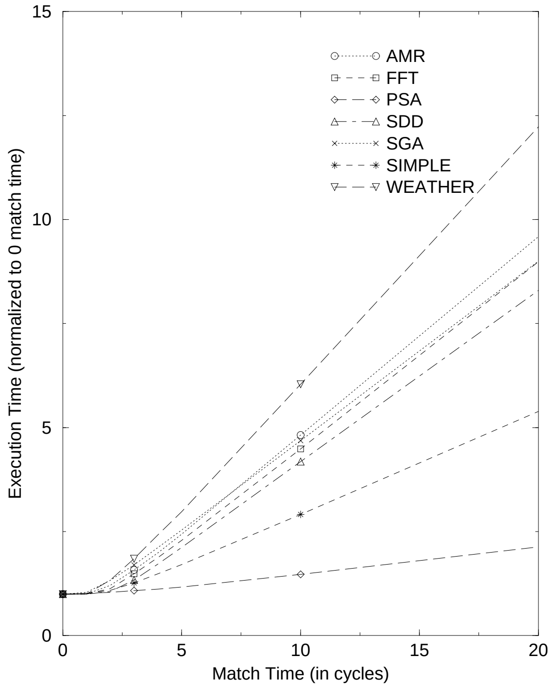


Figure 5.2: Execution Time versus Synchronization Time (5 Nodes).

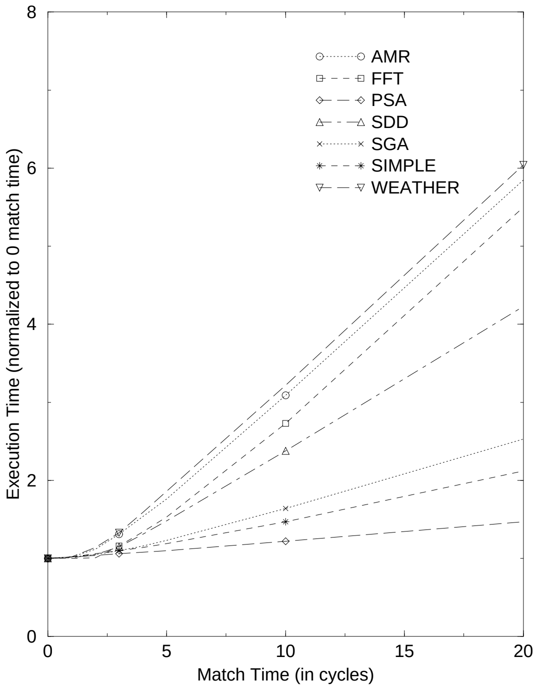


Figure 5.3: Execution Time versus Synchronization Time (50 Nodes).

represents the percentage of time the corresponding units are busy. In the first few columns where the synchronization cost is relatively low, the execution unit and the synchronization unit are well-balanced. However, as the synchronization cost is increased, the execution unit utilization drops and the synchronization unit utilization increases correspondingly. Here, the synchronization operations are no longer masked but rather represents a bottleneck. Therefore, the objective should be to have the ratio U_S/U_P as close to one as possible which, for most benchmarks, occur with the synchronization time of 1 or 2 cycles.

The results indicate that it is crucial that the cost of synchronizations be relatively small, otherwise the performance suffers. Therefore, an effective storage hierarchy is important.

5.1.2 Pebbles Storage Hierarchy

In this section, a storage hierarchy for the Pebbles model described in Chapter 4 is developed. First, the Pebbles architectural model is revised slightly as shown in Figure 5.4. The main difference from Figure 4.1 is the addition of two buffers: the *Framelet Buffer* and the *Instruction Buffer*. Also, there is a slight difference in the semantic of synchronization operations to be described shortly.

The allocation of the framelet is formalized as follows: for each instance of a thread, a *fixed size* framelet is allocated in the Frame Store. A framelet is large enough to hold inputs for most threads.[1] When a particular thread's requirement exceeds the size of a single framelet, one or more additional overflow framelets are allocated as needed. Another method, which the MIDC compiler does not yet implement, would be to break up the thread into two or more threads so that inputs can fit within a given size.

Figure 5.5 shows the design of the Frame Store. It is important to distinguish between the concept of *logical framelets* from that of *physical framelets*. Each *logical framelet* is associated with a particular instance

[1] The size of the framelet is discussed in Section 5.1.3: a framelet of 128 bytes can accommodate most of framelets in our benchmarks.

Sync. Time		0	1	2	3	4	5	10	20
AMR	T	772	769	917	1143	1433	1742	3400	6791
	U_P	80.1	80.4	67.5	54.1	43.2	35.5	18.2	9.1
	U_S	0.0	38.8	65.0	78.3	83.2	85.7	87.8	88.0
FFT	T	1381	1382	1468	1805	2258	2767	5510	11182
	U_P	58.2	58.2	54.8	44.6	35.6	29.1	14.6	7.2
	U_S	0.0	27.4	51.5	62.9	67.1	68.3	68.9	68.7
PSA	T	23859	24204	24746	25392	26048	26791	31077	41395
	U_P	5.9	5.8	5.7	5.6	5.4	5.3	4.5	3.4
	U_S	0.0	2.7	5.3	7.7	10.0	12.2	21.0	31.6
SDD	T	2908	2924	3087	3768	4767	5804	11147	21988
	U_P	57.1	56.8	53.8	44.1	34.9	28.6	14.9	7.6
	U_S	0.0	33.9	64.3	79.0	83.3	85.5	89.0	90.3
SGA	T	3121	3200	3699	4511	5295	6192	10698	19887
	U_P	30.8	30.0	26.0	21.3	18.1	15.5	9.0	4.8
	U_S	0.0	24.8	43.1	53.0	60.3	64.4	74.4	80.1
SIMPLE	T	7048	7190	7470	8135	8959	9951	15204	26421
	U_P	17.1	16.7	16.1	14.8	13.4	12.1	7.9	4.5
	U_S	0.0	12.0	23.2	31.9	38.6	43.4	57.0	65.7
WEATHER	T	1477	1483	1928	2637	3382	4142	8069	16152
	U_P	65.6	65.3	50.2	36.7	28.6	23.4	12.0	6.0
	U_S	0.0	48.4	74.4	81.6	84.8	86.6	88.8	88.8

T: Execution Time (in thousands of cycles)
U_P: Processor Utilization
U_S: Synchronization Unit Utilization

Table 5.1: Execution Time and Utilizations for Different Synchronization Costs.

of a thread and is allocated a space (a *physical framelet*) in the Frame Store. Until all the inputs to the thread arrive, the framelet is resident in the Frame Store. When the input set to a thread is complete, the thread becomes ready: its continuation moved to the *Ready Queue*, and its logical framelet moved to the Framelet Buffer. At this point, the *physical framelet* is freed in the Frame Store. Note that the logical framelet will never be referenced again. In addition, the instructions associated with the enabled thread can be prefetched into the *Instruction Buffer*.

In comparing the framelet-based storage model to the frame-based model [Pap88] discussed in Chapter 2, the frame-based model presents two distinct advantages: (1) fewer allocations and deallocations of storage in the frame memory and, (2) no need for copying of data values shared among threads within a code block: a thread writes a value once to a frame slot which may be read by multiple threads within that code-block. An advantage of the framelet-based model over the frame-based model is that the hardware based memory allocator designed to work with a small fixed size storage unit can be very fast. These two storage models are analogous to the segmentation (frames) and paging (framelets) strategies in virtual memory management. A hybrid model, where a frame consists of a set of framelets, is conceivable. It has the potential of combining the advantages of both models. This issue is not further pursued in this thesis. There is an orthogonal issue concerning the scheduling of threads such that all ready threads belonging to a code-block should be scheduled to execute one after the other. The framelet-based storage model does not preclude this type of *scheduling* discipline.

The objective of the Frame Store design is to reduce the average access time to the Frame Store and thereby reduce the synchronization costs. Among the framelet resident in the Frame Store, we can distinguish between two types: those that have been referenced in a recent past (e.g. within a window of references) and are therefore "active," and those that have not been accessed within the window and are thus "inactive." For a given window of references, the set of *active framelets* can be viewed as the *working set* of the program.

At any given time in the execution of a program, there may be a large number of framelets that are resident in the Frame Store: the active framelets are resident in the *Framelet Cache* (Cache) while the

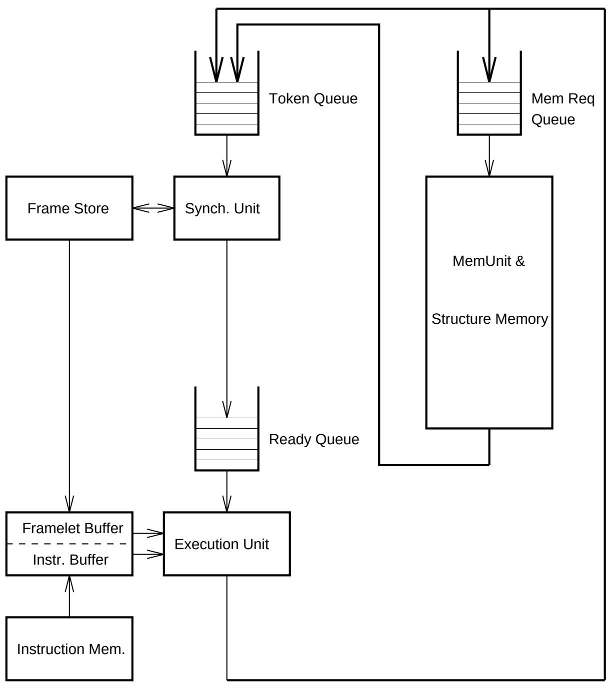


Figure 5.4: Revised Node Model.

inactive ones are stored in the *Backup Framelet Store* (B-Store). The Cache and B-Store operate exclusively: any framelet in the cache will not be resident in the B-Store and vice versa. With sufficient locality in the stream of incoming references, all active framelets can be stored in the Cache. For example, in the extreme case, if tokens arrive in such a fashion that the inputs for a given framelet arrive all at once followed by all the inputs for the next framelet, and so on; then, a single framelet sized cache would be sufficient to fully exploit locality.

Note that the locality refers to the accesses of the Frame Store by the Synchronization Unit. It is not the locality of execution since the Execution Unit does not access the Frame Store: all the input data for each ready thread are, already, in the register-like Framelet Buffer.

The Cache is modeled as a fully associative cache where each block corresponds to a framelet and is logically addressed. The logical address of a framelet consists of the thread identifier and the color. The access to the cache is fully pipelined and is stalled on a miss. The B-Store is physically addressed. Translation from the logical to the phyical address is performed by the *Backup Framelet TLB* (TLB) which has one entry per framelet. Figure 5.6 shows operations involved in storing an input to the Frame Store. When a logical framelet address is presented to the Frame Store, the address is simultaneously sent to both the Cache and the TLB. In the case of a cache miss and a TLB hit, the framelet is brought from the B-Store, possibly replacing a cached framelet. In the case of a miss in both the Cache and TLB, implying the reference is to a new framelet (i.e a compulsory miss), a block is allocated in the Cache again possibly replacing a cached framelet. Replacing a cached framelet involves allocating a framelet space in the B-Store, updating the TLB, and transferring the framelet from the cache. When a framelet is deallocated, the Synchronization Unit notifies the Cache so that the corresponding block can be marked invalid.

5.1.3 Cache Model and Methodology

A trace-driven simulation is used to model different cache designs and to determine the locality of accesses to the Frame Store. The traces of the incoming tokens of all the processors are generated by the Bedrock machine simulator.

Four cache replacement policies are studied for the experiments: 1) *FIFO* (First In First Out); 2) *LRU* (Least Recently Used); 3) *LFU* (Least Frequently Used); and 4) *MIR* (Most Inputs Remaining), where framelets with the most number of inputs remaining are replaced. The first case is chosen for its simplicity

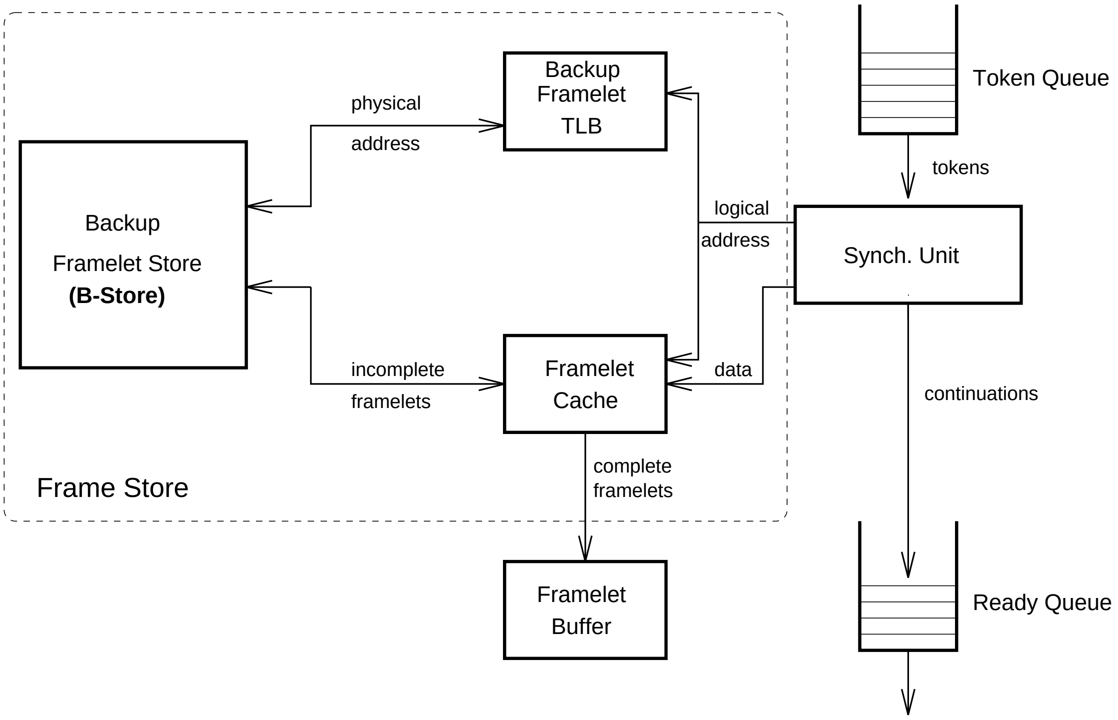


Figure 5.5: Cache Model.

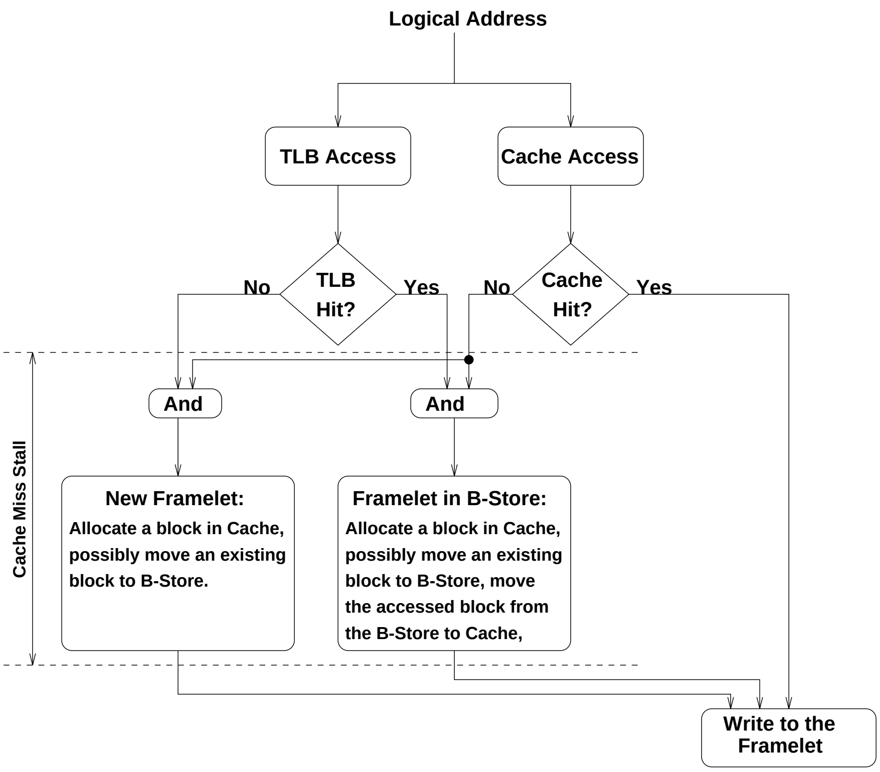


Figure 5.6: Processing of an input through the Frame Store.

Benchmarks	No. of Framelets	No. of Inputs	Avg. Input Set Size	Input Set Size < 128 bytes (%)
SGA	1,372,099	8,712,006	25.3	99.6
FFT	225,706	4,142,029	69.1	96.2
PSA	1,664,336	7,098,854	18.1	100.0
SDD	1,716,892	10,520,838	29.6	100.0
SIMPLE	1,328,271	9,269,106	34.4	98.1
AMR	205,774	3,207,898	85.1	84.2
WEATHER	828,983	8,045,883	35.9	99.8

Table 5.2: Benchmark Characteristics

in the actual implementation. The fourth case is chosen in the belief that framelets that still need a lot of inputs might not be active for a while and thus can be replaced.

In addition to the replacement policies, two framelet *allocation* policies are modeled:

- In the *Basic* policy, the Framelet Cache always uses the replacement policy to allocate a physical block to a referenced framelet not resident in the Cache. If the chosen block is not invalid, then the block is replaced to the B-Store.
- In the *Freelist* policy, the Framelet Cache maintains a list of free (invalid) blocks. When a cache block needs to be allocated (e.g. to store a new framelet or bring back an existing framelet from the Backup Store), the Cache allocates one from the free block list, if possible. If no free block is available, an existing framelet is replaced using the replacement policy. The advantage of this policy over the Basic is that it prevents an active framelet from being unnecessarily replaced.

Since the cache is fully associative, the misses are either compulsory or capacity misses. A compulsory miss occurs when the first input of a thread activation arrives. The capacity miss occurs when the number of resident framelets exceed the capacity of the cache. The rate of compulsory misses is inversely proportional to the average number of inputs to threads, and is independent of the cache size. There is no concept of invalidation miss since the only shared memory is the structure store and the strictness of Sisal program guarantees that the reading of a structure store location is not initiated until the write to the location is completed.

Benchmark Characteristics

The same set of benchmark programs described in Section 5.1.1 is used to conduct the cache experiments. Also, the same architectural parameters are used except that the synchronization latencies are fixed to that of the EM-4: a pipelined synchronization unit with a throughput of one synchronization per cycle and a latency of three cycles on a mismatch (first reference to a framelet). The choice of values in the architectural parameters have had only secondary effects on our measurement; they are provided here for the sake of completeness.

The default parameters used in our trace simulation are: a FIFO replacement policy and the Basic cache block allocation. Starting with this baseline configuration, we vary each parameter and measure its effect.

The dynamic characteristics of the benchmarks are given in Table 5.2. The second column in each row is the total number of framelets that were allocated per run, they range from around 200,000 to 1,700,000 framelets per benchmark. The third column is the total number of inputs to the framelet (which is the number of token), the range is from 3 to 10 million inputs. Handling each input requires several accesses to the Framelet Store by the Synchronization Unit, e.g. reading/writing of the counter and writing the value. The fourth column is the average size, in bytes, of the input set per framelet. The average size ranges from a low of 18.1 bytes in PSA to 85.1 bytes in AMR. The last column is the percentage of framelets whose inputs require less than 128 bytes. Except for AMR, a high percentage of the framelets can fit within a 128-byte physical framelet.

Benchmarks	Avg. Lifetime	% Lifetime < 2000	% Lifetime < 5000
SGA	3692	61.87	93.65
FFT	13960	84.81	87.07
PSA	698	93.78	96.86
SDD	5579	66.38	74.55
Simple	1702	81.64	89.02
AMR	6746	57.43	68.29
Weather	38099	58.13	67.18

Table 5.3: Lifetime (in References) of Framelets

5.1.4 Analysis and Evaluation

Performance of Baseline Configuration: Figure 5.7 shows the plot of the capacity miss rates of various benchmarks as the number blocks in the cache is varied. The compulsory misses are shown within parentheses next to each legend. The figure shows that there is a high degree of locality of references to the framelets. With a 256K cache (2000 blocks of 128 bytes each), the capacity miss rates are below 1% for all the benchmarks except for SDD and WEATHER. With a 5000 block cache, the miss rates drop below 2% for WEATHER and well below 1% for the others. The WEATHER benchmark works with a large data set and hence its inherent locality is rather poor. FFT has a very low miss rate of around 1% with only 100 block cache, but the miss rate does not drop much after a 1000 block. We can attribute the low miss rates to the following: 1) only a fraction of parallel loop iterations are likely to be active concurrently for an extended period of time and therefore present a higher locality; and, 2) there is a high locality among neighboring threads (e.g. within a loop body). Note that the compulsory miss rates are relatively high, 6.4% to 23.4%. A technique for alleviating the penalty of compulsory misses is discussed later in this chapter.

Block Replacement Policies: Figure 5.8 shows the performance of the four replacement policies for the AMR benchmark. The surprising result is that FIFO achieves as good a performance as LRU. Both are much better than either LFU and MIR. Similar results, not shown here, hold for all the other benchmarks: With a 1000 blocks, the miss rate of LRU is three times that of FIFO and LRU.

From this result we can conclude that there exist a certain pattern of references to the framelets that makes the *least recently* referenced framelet also the *earliest* referenced (first in). In order to understand the reference pattern, we have measured the *lifetime* of a framelet. The lifetime of a framelet is defined as the interval between the arrival of its first input and that of its last (when it is deallocated). This interval is measured in number of references to the framelet store (or token arrivals). Figure 5.9 shows the distribution of the lifetime for AMR and WEATHER which are representative of all the benchmarks. For the majority of threads, the lifetime less than 50 cycles. The cumulative distributions rise quickly, reach their plateau at around 1000-2000 cycles, after which they rise rather slowly. In Table 5.3, the second column shows the average lifetime of all the benchmarks. WEATHER has the largest average lifetime with about 38000 references, and PSA the smallest with about 700. The next column shows the percentage of framelets whose lifetime is less than 2000 references: 57% to 94% of framelets are in this category. Considering that the number of references per benchmarks is in the millions, a lifetime of under 2000 references is quite short. The next column is the percentage of framelets whose lifetime is under 5000 references: the range is from 67% to 97%. These results explain why aging-based replacement policies, such as LRU and FIFO, work well: With about 2000 cache blocks, by the time a cache block is replaced, it is highly probable that the associated framelet is already deallocated (i.e the thread became ready), and therefore represents a better candidate than a block that might still be active. Also, in FIFO, the time of arrival of the first input of each framelet is used for the replacement decision; whereas, in LRU, the time of arrival of the latest input of each framelet is used. If the difference in references between the first and latest of each framelet is similar to the others on the average, then FIFO functions just like LRU, since the oldest block is replaced (i.e. a framelet

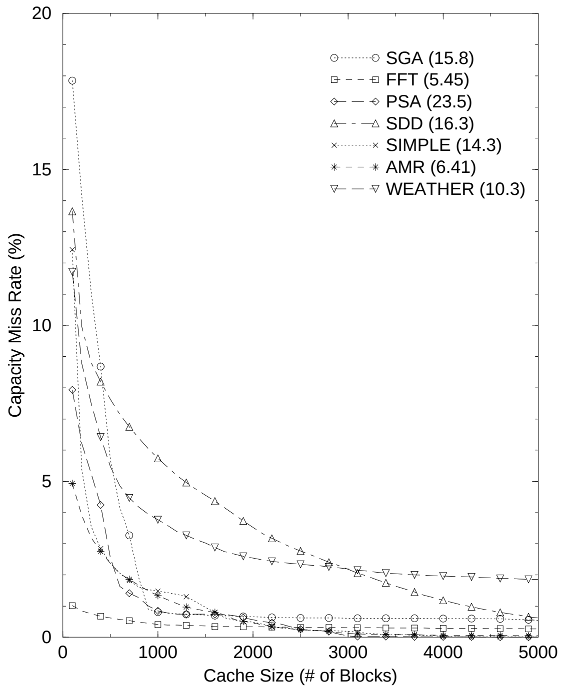


Figure 5.7: Capacity miss rates of baseline model per benchmarks.

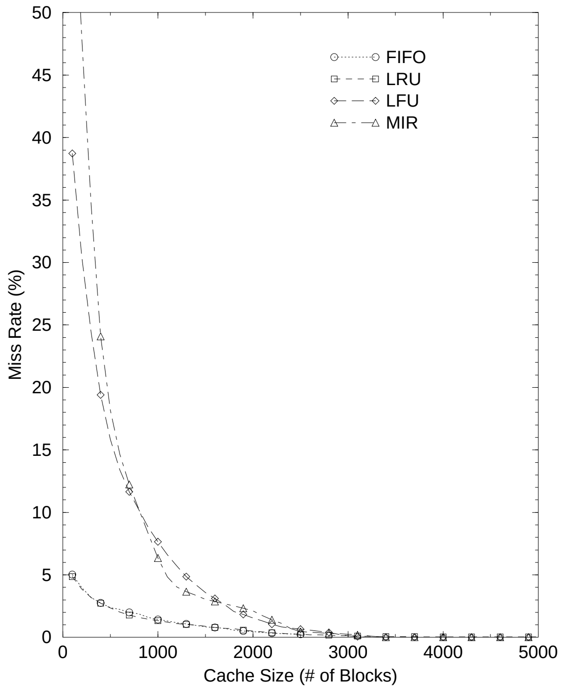


Figure 5.8: Capacity miss rates per replacement algorithm for AMR.

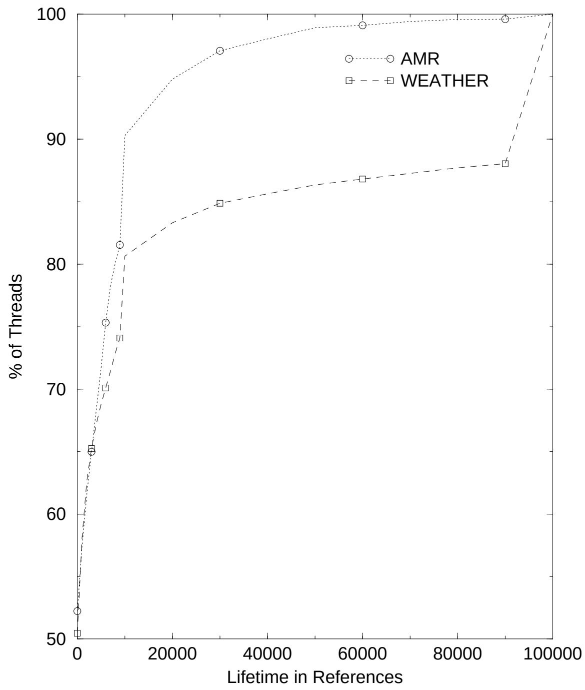


Figure 5.9: Distribution of Lifetime of Framelets in AMR and WEATHER.

whose latest input is old is likely to have old first input also). Therefore, the performance of the two should also be similar.

Block Allocation Policies: The comparison between the Freelist and Basic allocation policies is shown in Figure 5.10 for a set of representative benchmarks. It shows that the Freelist policy is very effective in reducing the capacity misses. Table 5.4 compares the two showing the miss rates of the Basic policy with 2000 and 5000 blocks cache and the number of blocks needed in the Freelist policy to achieve the same miss rates (to the nearest 50 blocks). It shows that fewer cache blocks are needed by the Freelist policy in order to achieve the same miss rates as the Basic policy. In some cases, such as in SGA and PSA, increasing the cache size by 50 blocks with the Freelist policy achieves the same reduction in miss rates that the Basic policy can achieve with 3000 more blocks. Note that the improvement comes from not replacing resident framelet whenever possible, thus saving the cost of bringing that framelet back at a later stage.

5.1.5 Reducing the Cost of Compulsory Misses

Our results indicate that the capacity misses can be made very small with a reasonable sized cache at which point the compulsory miss become the dominant type of misses. For example, with a 2000 blocks Basic cache (256K bytes), the capacity misses range from 0.34 to 3.52%, whereas the compulsory misses range from 5.45 to 23.4%. Unless the cost of the compulsory miss can be controlled, the effect of further reducing the capacity miss becomes negligible in the overall performance. The cost of a compulsory miss consist of the allocation of a cache block, and possibly replacing cached framelet. In this section we investigate an extension to the Freelist allocation policy that aims at reducing the cost of a compulsory miss, whenever possible, to that of a cache hit. This scheme is referred to as the *Reserve Block* scheme.

The Reserve Block scheme, in essence, attempts to reduce the cost of the compulsory misses by making use of the free-blocks and marking one of them as *reserved* specifically to handle the *next* compulsory miss. The reserved block *cannot* be used to satisfy the capacity miss, i.e it will not be used to allocate a framelet

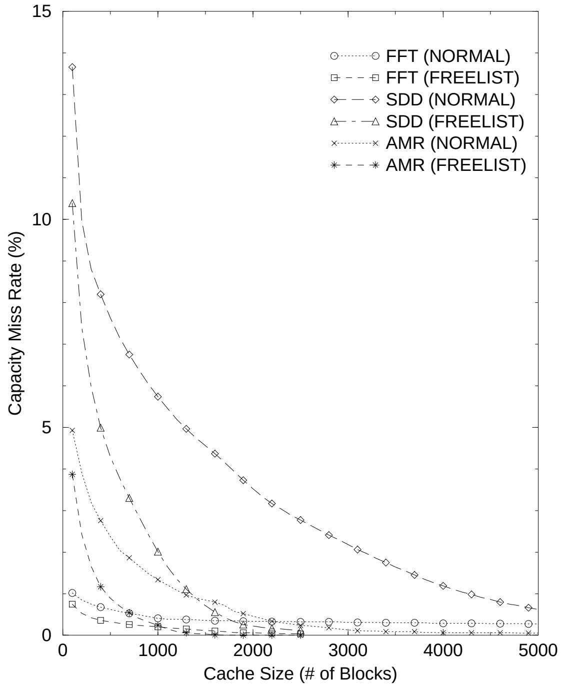


Figure 5.10: Effect of Freelist and Basic allocation policies on miss rates

Benchmarks	BASIC (A) Miss rate	FREELIST (B) Blocks	BASIC (C) Miss rate	FREELIST (D) Blocks
SGA	0.66	550	0.54	600
FFT	0.34	450	0.27	650
PSA	0.56	350	0.01	400
SDD	3.52	650	0.62	1550
SIMPLE	0.49	450	0.01	650
AMR	0.46	750	0.05	1350
WEATHER	2.54	700	1.86	1200

A: Miss rates with 2000-block cache
B: Blocks needed to achieve the same miss rates as A
C: Miss rates with 5000-block cache
D: Blocks needed to achieve the same miss rates as C

Table 5.4: Comparison of Miss Rates between Freelist and Basic cache

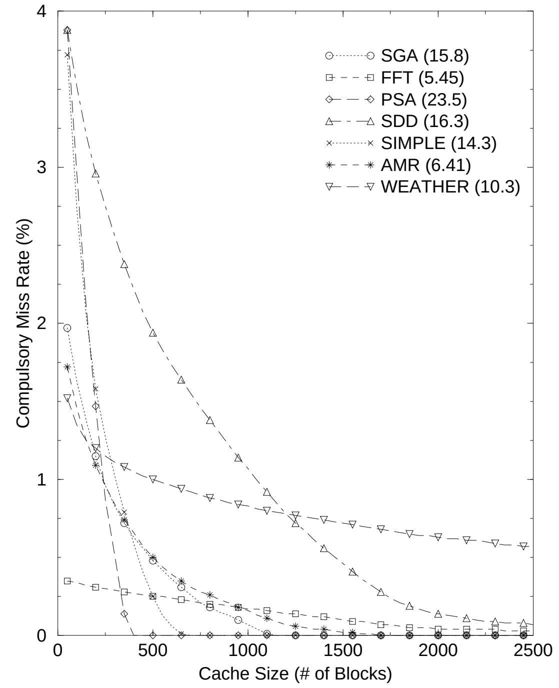


Figure 5.11: Compulsory miss rates using the Reserve Block scheme (Original miss rate in parentheses next to legends).

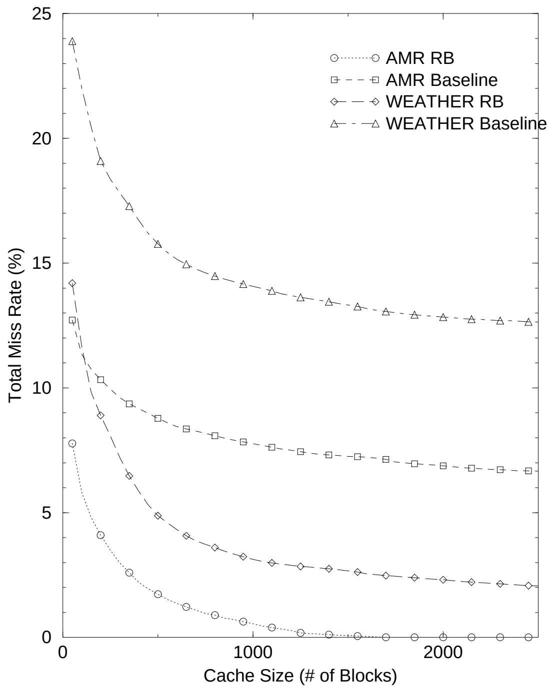


Figure 5.12: Total miss rates using the Reserve Block (RB) and the Baseline schemes.

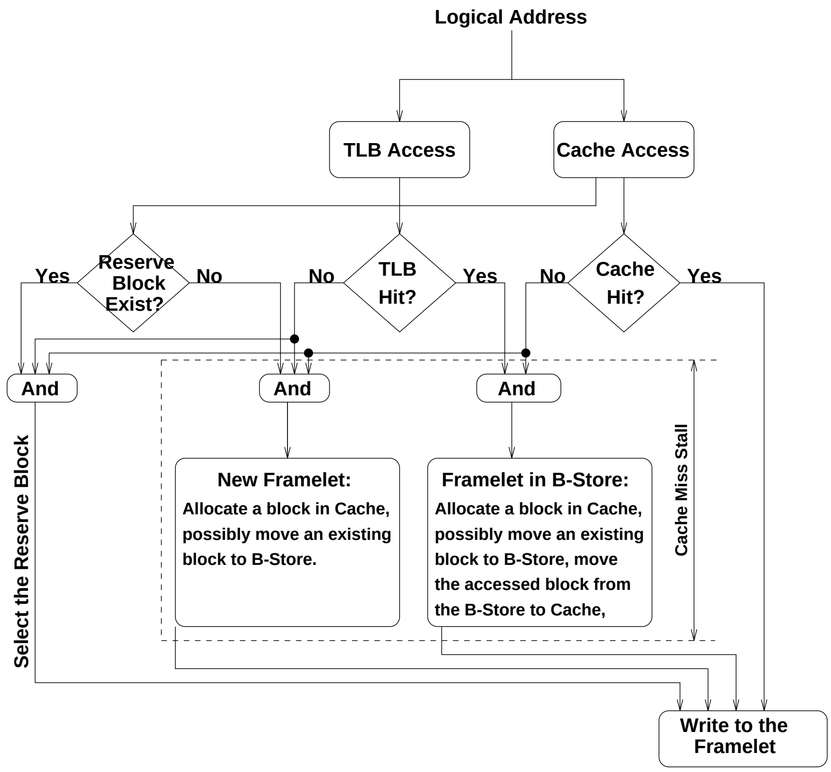


Figure 5.13: Processing of an input through the Frame Store.

	Total Miss Rates for 2000 block cache:						
System	SGA	FFT	PSA	SDD	SIMPLE	AMR	WEATHER
Baseline	16.4	5.79	24.0	19.8	14.8	6.88	12.8
Reserve	0.01	0.14	0.01	0.45	0.01	0.01	2.31

Table 5.5: Comparison of the Total Miss Rates between Baseline and Reserve Block Schemes

from the B-Store. The index of the reserved block in the cache is kept in a special register. Recall that a compulsory miss is detected by a miss in both the Cache and TLB. When such a miss is detected, the reserved block immediately becomes the substitute block associated with the logical framelet with a very small delay. The management of the free list in the cache and the designation of the reserved block is done in the "background" and would not affect the cache access time. A stall occurs when there is no reserve block (i.e. the free-block list is empty). In this case a cache block must be replaced. Note that a stall occurs whenever a compulsory miss is *immediately* (i.e in the next cycle) followed by one or more compulsory misses. Figure 5.13 shows the operations involved in the Reserve Block scheme.

The compulsory miss rate is now dependent on the probability of encountering an empty free block list. Hence, the cost of accessing a new framelet is near that of the cache hit if the allocation occurs out of the reserve block. Figure 5.11 shows the effect of the Reserve Block scheme: the effective compulsory misses are significantly reduced even with a very small cache size. With a reasonably sized cache (i.e. 2000 blocks), the compulsory misses are smaller than the capacity misses for all the benchmarks. A beneficial side effect of this scheme is that the capacity misses are also reduced since an existing block will not be replaced to make a room for the allocated framelet as in the Freelist policy. Figure 5.12 shows the total miss rates (both compulsory and capacity) of the Reserve Block and the Baseline schemes for representative AMR and WEATHER. The total miss rates with a 2000 blocks cache is shown in Table 5.5 for the two schemes. The reduction in the total miss rate ranges from a factor of six to 2400! The miss rates for the Reserve

2000 block cache:						
SGA	FFT	PSA	SDD	SIMPLE	AMR	WEATHER
1.000	1.006	1.000	1.018	1.000	1.000	1.092

Table 5.6: The Average Synchronization Time of the Reserve Scheme

scheme range from 0.01% to 2.31%. This can be considered a tremendous improvement for a relatively simple scheme. It demonstrates the effectiveness of exploiting a constant stream of framelets being allocated and deallocated.

Table 5.6 shows the average synchronization time given the miss rates implied by the Reserve Scheme. The hit time of 1 cycle and miss penalty of 5 cycles are used to derive the effective synchronization cost. It shows that the average cost is nearly equally to one cycle. Therefore, the result plainly indicate that the storage model described in this section causes minimal degradation in overall performance.

5.2 Communications Issues

In this section, three specific issues related to communications are explored: 1) categorization of messages into different types which maybe helpful in designing code generators, 2) input locality of threads that may be useful in designing caches and effective scheduling disciplines, and 3) output locality of threads that reveal how the outputs are dispersed. Section 5.2.1 explores these issues along with simulation results based on a set of benchmarks are given.

Results indicate that about a third of messages are related to remote memory accesses, and another third are related to handling parallelism. In addition, there is a high amount of both input and output localities. These results indicate that a couple of compiler techniques, called *grouping* and *bundling*, can effectively reduce the requirements on the network via reductions in the number of inter-processor messages and in overheads. Although these techniques are not novel, their appeal lies in their simplicity of implementations and applications of these techniques demonstrate the effectiveness of quantitative analysis of communication patterns in multithreaded executions. Section 5.2.4 discusses these overhead reduction techniques along with their effectiveness.

5.2.1 Message Characteristics

For the purpose of characterizing tokens, we classify tokens into two major categories: *memory* tokens represent structure store reads and writes, and *regular* tokens represent non-memory related control and data tokens. The *regular* tokens are broken down into three types: *loop*, *call*, and *other*. *Loop* tokens represent tokens (data and control) used in distributing parallel work and in gathering their results. *Loop* tokens could represent a significant fraction of regular tokens, especially for programs with a large number of parallel `forall` loops and small loop bodies. *Call* tokens represent all tokens (data and control) involved in calling a function and returning its results. *Other* tokens represent all other tokens.

Experimental results discussed in this section are based on the following set of eight Sisal programs: 1) *SGA*, 2) *FFT*, 3) *HILBERT* computes the condition number for Hilbert matrix coefficients. It uses Linpack routines. 4) *PSA*, 5) *SDD*, 6) *SIMPLE*, 7) *AMR*, and 8) *WEATHER*.

The characteristics of each benchmark are given in Table 5.7. The first and second columns give the number of threads executed and tokens generated, respectively, in each run. The third column gives the average number of matches performed per MIDC instruction executed (MPI). The average figures are fairly consistent of around 0.5 matches per instruction.

Again, the experiments are conducted with the Bedrock simulator. Where timed measurements are reported, the same architectural parameters as described in Section 5.1.1 is used, unless otherwise noted. It is assumed that all messages take the same latency unless they are specifically tagged as local. Again, *all* structure store reads and writes go through the interconnection network. Obviously, some of these messages can be made local by a judicious allocation of data structures. This, however, requires extensive static analysis in the compiler which is beyond the scope of this thesis.

	Threads	Tokens	MPI
AMR	207,178	3,207,898	0.50
FFT	225,706	4,148,429	0.41
HILBERT	744,047	3,286,375	0.56
PSA	1,670,678	7,098,854	0.44
SDD	1,716,892	10,520,838	0.53
SGA	1,409,905	8,713,308	0.54
SIMPLE	1,333,209	9,269,106	0.56
WEATHER	882,508	8,045,883	0.56

Table 5.7: Benchmark Characteristics

	Regular				Memory		
Types	Loops	Calls	Others	Total	Reads	Writes	Total
AMR	65.0	1.1	5.50	71.6	21.7	6.7	28.4
FFT	68.9	0.0	2.0	70.9	14.9	14.3	29.2
HILBERT	59.6	0.1	7.6	67.3	26.3	6.4	32.7
PSA	17.9	14.4	23.0	55.3	31.8	12.9	44.7
SDD	44.5	0.1	13.2	57.8	36.1	6.2	42.3
SGA	24.4	4.3	49.1	77.8	13.2	9.0	21.2
SIMPLE	28.0	5.0	32.7	65.7	29.0	5.2	34.2
WEATHER	38.6	0.8	35.5	74.9	19.8	5.3	25.1
Average	43.4	3.2	21.1	67.7	24.1	8.2	32.3

Table 5.8: Breakdown of All Messages in Percentage

Table 5.8 shows the breakdown of tokens according to different types. It shows that memory tokens represent a significant fraction of the total traffic, amounting to roughly a third on the average. This implies that techniques that reduce memory tokens are essential in achieving a significant reduction in the amount of network communications.

Among *regular* tokens it can be observed that *loop* tokens comprise 30-97% of the total. For AMR, FFT, and HILBERT the loop tokens represent a vast majority of regular tokens; hence, a significant amount of traffic is devoted to handling parallelism. These results are not surprising considering that threads, in our model, are constrained to be no larger than a loop body or function body and are terminated at structure store accesses[2]. The three programs above are highly parallel with each parallel loop containing only few (1-3) threads.

Because our code generation and optimization in-lines all small functions, *call* tokens represent a relatively small proportion of all tokens. Therefore, except in the case of PSA, most regular tokens are either *loop* and *other* tokens.

5.2.2 Input Locality of Messages

We define input locality of a thread as the inverse of input latency. Input latency refers to the time delay between the arrival of the *first* token to a thread instance and that of the *last* token, at which time the thread can start executing. Figure 5.14 shows the cumulative distributions of input locality in terms of processor cycles. The trend shows that for most programs, about 30% of threads have inputs that arrive

[2]Several optimizations at the global level attempt to make threads as large as possible [RNSB94].

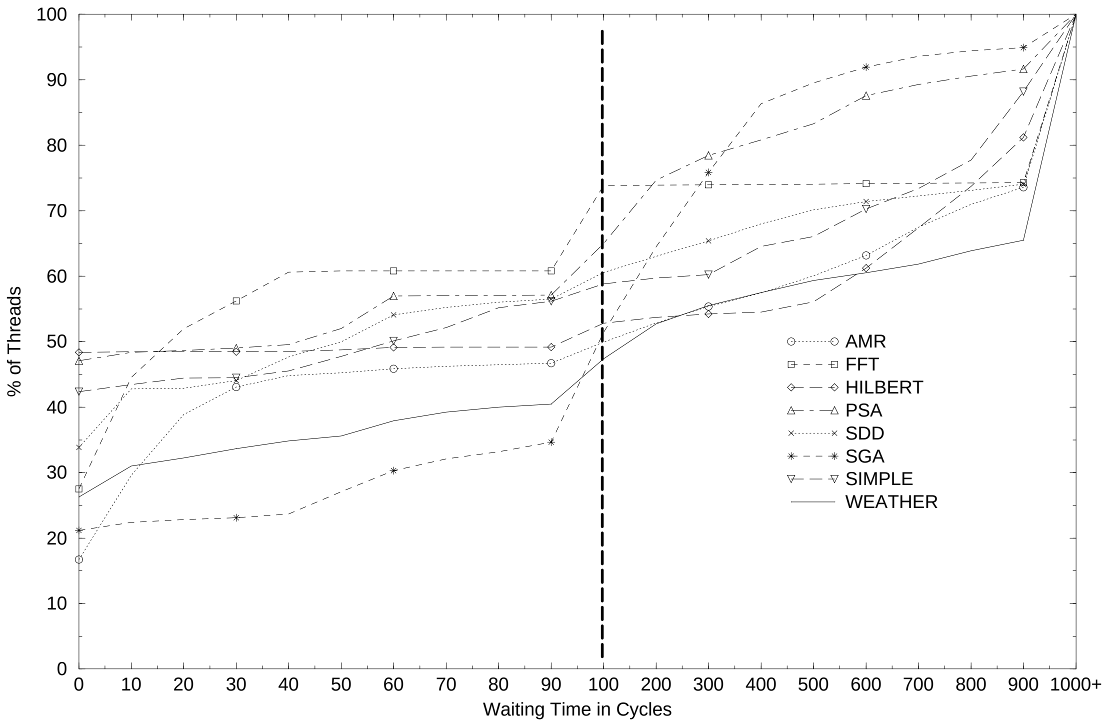


Figure 5.14: Input Locality in Cycles (Cumulative Distributions)

within 10 cycles of each other; and in 50-60% of threads, inputs arrive within 100 cycles of each other. This is significant considering that each network transit takes 50 cycles. On the other hand, between 5% to 35% of threads have input latency greater than 900 cycles. These results are most useful in the design of a token storage hierarchy where short input latency threads are allocated in the cache and long input latency ones are migrated to the slower, longer term main memory. A cache architecture that exploits these properties is described and evaluated in [RN95].

5.2.3 Output Locality of Messages

One of the metrics related to message generations is *output locality* which attempts to quantify the amount of locality in the outputs messages (tokens) generated by threads. It is measured by the ratio of the total number of regular tokens generated by a thread (M) and the number of distinct threads to which these tokens are sent (T). This measure attempts to capture the locality in the output destination of messages. Note that in any type of implementation, a thread destination will eventually be translated into a frame or framelet address. The output locality therefore captures the locality of writes to the frame or framelet store. The values of M, T and their ratio are shown in Table 5.9 for our benchmarks.

The table indicates that there exist a significant amount of output locality in most programs. It shows that, on the average, about 3-4 tokens are sent to each target thread. Except for AMR and FFT, the average number of targets are less than two. For both AMR and FFT, each thread generates a much larger number of regular tokens sent to a larger number of threads, compared with other benchmarks; however, the ratio M/T is only slightly larger than some of the other benchmarks. AMR and FFT both have large average thread sizes and each loop body has a smaller number of threads, as revealed in the next section. These results indicate that this output locality can be exploited by *bundling* tokens destined for a same thread into one message to reduce the overhead is message set-up and reception.

5.2.4 Overhead Reduction

	M	T	M/T
AMR	11.9	2.7	4.40
FFT	15.2	3.4	4.47
HILBERT	3.2	1.14	2.80
PSA	2.7	1.13	2.39
SDD	3.8	1.28	2.96
SGA	5.3	1.82	2.92
SIMPLE	4.8	1.50	3.21
WEATHER	7.2	1.94	3.71

Table 5.9: Output Locality

In this section, given the results in the previous section, we present two techniques designed to reduce the amount of regular token traffic. In this study, we do not consider ways to reduce the remote memory accesses. For example, static data partitioning can reduce the amount of remote accesses. This, however, requires extensive compile-time static analysis which is beyond the scope of our present work. Instead, we assume that all structure store reads and writes still go through the network.

Grouping Technique and Measurements. As a first step toward reducing the regular token traffic, we first observe that *loop* and *call* tokens are the primary means by which parallelism is exploited. *Other* tokens represent intra-loop and intra-function thread communications that could be localized to a processor. In essence, by making sure that threads within a loop or a function body are allocated to the same processor, the communications between these threads remain local to that processor. This *allocation* strategy should also result in a higher locality and better exploit the memory hierarchy.

Although the grouping idea is rather obvious, our objective is to quantify its possible benefit against the cost it might entail. The allocation of a group, either a loop or function body, to a processor is done by hashing the *color* portion of a token tag, rather than the entire tag. In addition, the compiler also generates code so that threads directly write to the data memory, rather than generate locally bound tokens, for intra-group communications.

Since different colors are generated for each `forall` loop iteration and for each call, each activation of a group would never cross the boundaries of loops or functions. We emphasize that grouping applies only to the processor allocation and does not entail any scheduling policy. The order of execution of threads within a group is still determined by the availability of data. However, a policy which schedules all enabled threads from the same group together should better exploit memory hierarchy.

In summary, the execution of thread groups is characterized by

- A thread group is a statically determined set of threads that are allocated on one node (processor).
- Different activations of the same group could be allocated on different nodes, their allocation is determined dynamically at run time (by the hashing on the color field).
- The order of thread execution within a groups is purely determined by the availability of data.

The characteristics of thread groups are shown in Table 5.10. S_1^G represents the average number of threads in a group, and Π^G represents the average parallelism of a group in terms of threads. The table shows that each group consists of a relatively small number of threads. The table also shows the average number of instructions in a thread and the average parallelism within a thread, represented by S_1^T and Π^T respectively. FFT in particular has very large threads and large internal parallelism. Note that the average instruction level parallelism within a thread, Π^T, is very close to four instructions per cycle.

The second column indicates that intra-group parallelism is nearly one for all the benchmarks. In other words, even when there are infinite number of processors that can exploit all inter-thread parallelism, threads within a group execute nearly sequentially. Therefore, no parallelism is lost by grouping threads and allocating them on a single processor, which is an important objective of our code generation.

	S_1^G	Π^G	S_1^T	Π^T
AMR	2.47	1.00	30.79	5.29
FFT	1.65	1.00	44.35	4.77
HILBERT	3.91	1.00	7.95	2.41
PSA	4.72	1.04	9.65	2.69
SDD	3.77	1.01	11.61	3.05
SGA	5.43	1.01	11.45	3.39
SIMPLE	5.05	1.02	12.42	3.40
WEATHER	3.65	1.00	16.35	4.22
Average	3.83	1.01	18.1	3.65

Table 5.10: Thread and Group Characteristics

	Tokens (Millions) Before Grouping	Tokens (Millions) After Grouping	% Reduction
AMR	2.30	2.12	7.5
FFT	2.94	2.86	2.8
HILBERT	2.21	1.96	11.3
PSA	3.93	2.43	38.0
SDD	6.08	4.67	23.2
SGA	6.78	2.71	60.0
SIMPLE	6.09	3.54	41.9
WEATHER	6.03	3.37	44.1
Total	36.36	23.66	35.0

Table 5.11: The Effect of Grouping on Regular Tokens

The result of thread grouping on reducing the amount of inter-processor communication is shown in Table 5.11. The first two columns show the number of the *regular* tokens before and after the grouping. The last column show the percentage reduction of the *regular* tokens. The table shows that there are significant reductions in the amount of regular tokens generated (nearly 30%). Grouping only affects the *other* tokens, in fact eliminating most of these tokens. A main exception arises from sequential `forinit` loops passing data across iterations. As expected, some benchmarks, such as AMR and FFT, do not gain much from grouping due to the small number of *other* tokens.

Message Bundling Technique and Measurements. Output locality results suggest that it may be profitable to combine separate tokens to the same thread into one larger token. This *message vectorization* should reduce the number of network packets generated at the expense of larger message size. It is worth noting that the *bundling* does not introduce any additional latency since the target thread cannot be enabled until all the inputs arrive anyway. Although *grouping* is only effective for the *other* tokens, *bundling* should be effective for both the *loop* and *call* tokens. Therefore, they are complementary techniques.

Table 5.12 shows the result of combined bundling and grouping. It shows the number of messages generated before and after applying the techniques. In the bundling technique, we limited the maximum token size to consist of no more than five data values. The reduction is most significant for the *loop* tokens. This is due to the fact that activating each loop iteration requires sending a lot of information to the same node. Almost all the *other* tokens are eliminated.

The overall percentage reductions in the number of regular tokens is given in the last column of the table. The results are fairly uniform in that all programs result in significant reductions ranging from 68.2-86.2% with the average of nearly 80%. These result demonstrate that bundling and grouping are complementary

	Loop Tokens (x10^6)		Call Tokens (x10^6)		Other Tokens (x10^6)		%
	Before	After	Before	After	Before	After	Reduction
AMR	2.09	0.47	0.04	0.01	0.18	0.00	79
FFT	2.86	0.64	0.00	0.00	0.01	0.00	78
HILBERT	1.96	0.70	0.00	0.00	0.25	0.00	68
PSA	1.27	0.41	1.02	0.53	1.63	0.07	74
SDD	4.68	1.58	0.01	0.00	1.39	0.00	74
SGA	2.13	0.65	0.37	0.21	4.28	0.07	86
SIMPLE	2.60	0.71	0.46	0.12	3.03	0.22	82
WEATHER	3.11	0.84	0.06	0.02	2.86	0.05	85
Total	20.70	6.00	1.96	0.89	13.63	0.41	78

Table 5.12: The Reduction in Tokens After Grouping and Bundling

	% Reduction	MPI
AMR	56.6	0.19
FFT	55.4	0.14
HILBERT	45.9	0.29
PSA	41.0	0.23
SDD	42.7	0.29
SGA	68.1	0.14
SIMPLE	53.9	0.24
WEATHER	63.6	0.18
Average	53.4	0.21

Table 5.13: Overall Percentage Reduction of All Tokens and MPI

techniques; when one is ineffective, the other makes up for it. For example, AMR and FFT have a small number of intra-group tokens to be eliminated, but bundling reduces the number of their *loop* and *call* tokens by more than a factor of four, resulting in overall reductions comparable to other benchmarks.

In the first column of Table 5.13, the overall percentage reduction in the number of *all* tokens generated is given. We see that more than half of all tokens are eliminated. The second column of the table gives the resulting MPI figures and shows a large drop in matches per instructions. This ranges from about 0.14 to 0.29 with the average of 0.21.

The larger average message size should result in slightly higher overhead per message processing. However, the network transit time should not be significantly affected if we assume a cut-through type of routing. This message vectorization should therefore be effective in reducing the overall communication overhead.

5.3 Parallelism Issues

In dataflow derived multithreading models such as Pebbles, abundant parallelism could overwhelm the machine resources and reduce locality. In this chapter, two techniques, *slicing* and *chunking*, that attempt to address the above shortcomings are presented. *Slicing* mainly attacks the problem of managing the resources, whereas *chunking*, like vectorization, mainly attempts to improve locality. The work presented in this section is done in collaboration with Bhanu Shankar.

An implementation of the two techniques are described in Section 5.3.1. Section 5.3.2 describes the resulting machine independent code characteristics including the average number of instructions executed per thread. The simulation results that compare the effectiveness of these techniques against code with unrestrained parallelism are presented in Section 5.3.3. The simulation results indicate that the chunking method helps reduce execution time and also show an appreciable decrease in the utilization of the synchronization

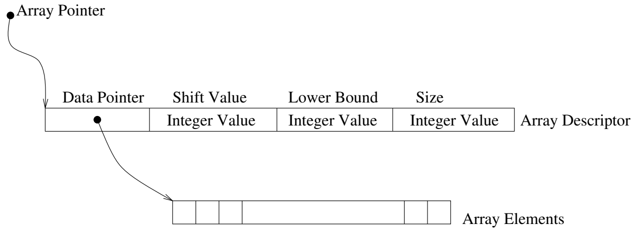


Figure 5.15: Layout of Arrays in MIDC.

unit. The slicing method shows lower average and maximum matching store occupancy at the expense of increased execution time. By combining both techniques, it is possible to balance speedup with resource utilization.

Slicing was chosen over K-bounding as the method of parallelism control for the following reasons.

- *Increased Token traffic:* As the model of computation uses non blocking threads, token traffic is increased due to the recirculation of the various input values from the generator back to itself even when there are no more iterations to execute, this will happen at least k times.
- *Increased Hot Spot activity:* In the case of k-bounded loops, there would exist two serial hot spots, the first in the generator thread and the second in the reduction and synchronization step. It is not possible to merge these two threads as there is a phase difference of k between the execution of the generator and the reduction steps. In the case of loop slicing, the generator would create a number of sub-master processes which would control the different iteration spaces in parallel and alleviate the hot spot in the generation of tasks. Partial reductions can be done in parallel using a similar mechanism. Thus, the generation and reduction bottleneck is widened to some extent. It is our claim that the bottleneck is removed completely in the slice generator and widened enough at the final reduction stage removing the hotspot.
- *Reuse of resources:* The MIDC model of execution does not allow for the easy reuse of resources as the threads are non-blocking. The best way of implementing k-bounded loops is via the use of circular buffers. Its implementation in the MIDC model is difficult at best.

5.3.1 Analysis

There are two types of loops in SISAL. *Iterative loops* which have loop carried dependencies and termination tests, and *Parallel loops* which have data independent loop bodies and known loop counts. Only the parallel loops are considered for parallelization and vectorization.

As most parallel loops deal with streams or arrays, it is important to know the layout of these data structures in memory. The layout of an array data structure containing an array descriptor and the data elements was discussed in Chapter 4.1.3, reproduced in Figure 5.15. SISAL arrays can start at any lower bound and can be of a variable size, and this information is encoded in the array descriptor. In order to reduce copy operations (e.g. when concatenating arrays), additional memory may be allocated on either side of the array data elements and several arrays can therefore be "*built-in-place.*" This requires a "shift" value to specify where the logical array starts. Thus, the start of the array is given by adding the values of the `data pointer` to the `shift value`. All elements of that array are offset off this resultant start address. With this layout, two memory latencies is required in order to fetch a single array element.

Loop Chunking

Chunking is a method by which vector execution can be simulated on a non-vector processor. A fixed amount of work is provided to a variable number of worker processes. In order to generate chunked code, the innermost parallel loops have to be identified. For the loop to be vectorizable, or chunkable, consecutive loop bodies must refer to consecutive array elements. Other types of array references and constructs causing unknown latencies, such as function calls, cause the loop to be identified as non-chunkable. The presence of a

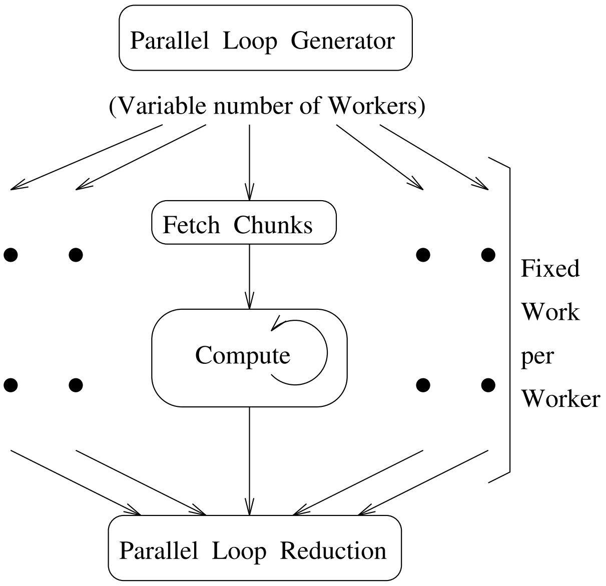


Figure 5.16: Chunk Control in loops.

dual level memory access to fetch an array element would cause a problem in chunking loops, as the memory access pattern would be unknown at compile time. Fortunately, when reading an array element, reading the array descriptor in the case of a vectorizable/chunkable loop is a common subexpression. Common subexpression elimination moves it out of the loop allowing the compiler to analyze the access pattern. Chunking is used when we require the work done by a worker to be fixed and we do not mind having a variable number of workers.

The chunk size is fixed at the compile and is limited by the underlying memory architecture (e.g. the size of a cache line); hence, the loop is strip-mined to allow for variable loop sizes. The chunk generation and control schema are shown in Figure 5.16.

At run-time the number of workers required have to be computed. For a loop of iteration space `n` and a machine chunk size of `c`, the number of workers, `w`, are computed as: $\lfloor \frac{n}{c} \rfloor + 1$ if $n \; mod \; c \neq 0$ or $\frac{n}{c}$ otherwise. The code dealing with the irregularly sized chunk would be used if such a chunk exists.

A split phase **`FetchCHunk`** operator is used to fetch a chunk of data from structure memory. The semantics of this operation is defined as follows: memory is reserved in the target processor's data memory to hold the chunk. The MemUnit sends the chunk to the target thread if and only if all the chunk elements are present, i.e., the presence bits for the entire chunk is set. Since SISAL is a strict language, all the data elements of an array will be written sooner or later with no hole left in the array. Once the chunk is received and stored into the target processor's local memory, the handle to the memory is passed onto the thread that consumes this data chunk.

While the thread executes, the **`LOAD`** instruction is used to move the chunk element from local memory into a register for the computation to proceed. A simple sequential loop structure is used for the computation of successive elements.

The code necessary to compute the number of workers, the size of the odd chunk and the code to spawn off the odd chunk gives rise to overheads in spawning chunked loops. In addition, each chunk element has to be loaded into a register before computation can proceed, although the load latency can be masked via software pipelining.

Loop Slicing

Slicing is a method by which a parallel loop is split up among a fixed number of worker processors. This enables the control of parallelism. Slicing allows for multiple serial latencies in the loop body. The iteration space is split between a fixed number of workers. All workers will perform at least $\lfloor \frac{n}{k} \rfloor$ work (n is the size

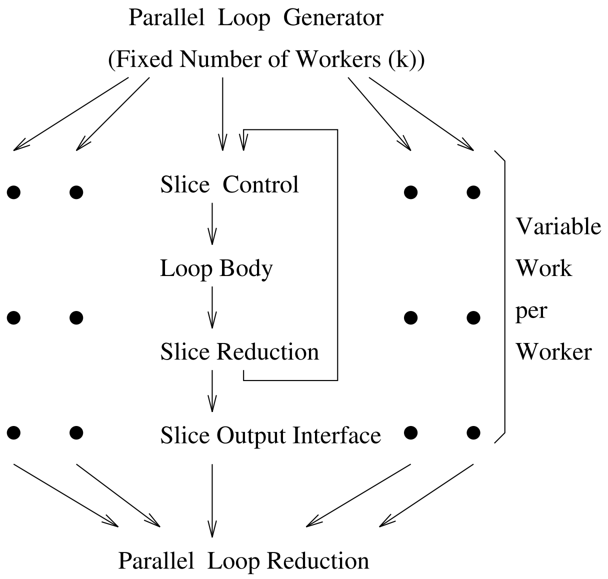


Figure 5.17: Slice Control in loops.

of the iteration space, k is the number of workers). $n \ mod \ k$ workers will perform an additional iteration of work. Slice control is shown in Figure 5.17.

Each portion of the iteration space assigned to workers are executed in a sequential fashion. Inputs to all iterations of the iteration space are equal except for the index value. The index value port is identified and is updated at each execution of the loop body. The reduction required in the parent loop is also divided over the iteration spaces, reducing the amount of serial reduction required, i.e., the bottleneck caused by the strict semantics of SISAL is widened to some extent.

Three additional threads are introduced to control the execution of the run time slices as shown in Figure 5.17. These are the `Slice Control`, the `Slice Returns` and the `Slice Output Interface` threads. The `Slice Control` is provided with the loop count and the iteration space. A simple sequential counter is used to determine whether further work is required to be performed. The `Slice Returns` takes care of all synchronization and partial reduction of results. The `Slice Output Interface` is used to color the outputs of each slice to that of the parent, and furnishes partial results to complete the final reduction of results.

Slicing reduces the resource load on the system in terms of the number of colors or activations required, with each slice taking one color rather than with each iteration. Essentially, by slicing, the demand for system resources are "smoothed out" over time; for example, a sudden high demand caused by a parallel loop generation is spread out over time. Therefore, the peak load on the synchronization unit is also reduced. The overhead of loop slicing is mainly due to the `Slice Control` thread that control the iterations. Both `Slice Returns` and `Slice Output Interface` threads do the work (i.e. reduction and recoloring) that need to done anyway in the non-sliced execution and represent only slightly higher overhead.

5.3.2 Code Characteristics

In this section, we evaluate the dynamic properties of our code before and after applying various combinations of slicing and chunking. We have obtained these results by running codes on a multithreaded machine simulator.

Benchmarks and Machine Model

The experiments are conducted using the following set of benchmark programs: *FFT*, *HILBERT*, *BMK11A* is particle transport code developed to evaluate Cray Computer systems at Los Alamos National Laboratory, *SDD*, *SIMPLE*, *AMR*, and *WEATHER* using 840km grid size and 5 time steps. The characteristics of the benchmark programs in terms of the number of parallel loops and vectorizable or chunkable

Program	Problem Size [Time Steps]	Parallel Loops	Vectorizable Loops
FFT	2^{15}	14	11
HILBERT	100 x 100	65	30
SDD	2^6	80	23
SIMPLE	100 x 100 [5]	78	42
AMR	4 x 80 x 40 [4]	87	34
WEATHER	840km [5]	81	29

Table 5.14: Program Characteristics of the Benchmarks.

loops are given in Table 5.14. It is noted that the SDD has the lowest percentage of vectorizable parallel loops of around 29% and FFT has the highest percentage with 79%.

The same machine configuration as described in Section 5.1.1 is used. It is assumed that the size of matching store is unlimited, and therefore can handle any amount of parallelism.

Measurements

The four combinations of slicing and chunking are compared: R: unconstrained parallel threaded code, used as the base case for all comparisions, C: vectorizable loops chunked, S: loops sliced, and CS: vectorizable loops chunked and other loops sliced. The following measures are used to evaluate performance:

- *Time:* the number of cycles taken by the program to execute.
- Ω_{avg}*:* the average occupancy of the matching store in terms of the number of threads that are waiting for inputs.
- Ω_{max}*:* the maximum occupancy of the matching store in terms of the number of threads that are waiting for inputs at any given time.
- U_P*:* the processor utilization, i.e., the percentage of time the processors are busy.
- U_S*:* the utilization of the synchronization unit, i.e., the percentage of time the synchronization unit is busy.

In evaluating the comparative performance between the different sets of parallelism control methods, the following measures will be used.

- *Speedup:* the percentage improvement of execution time (e) over the execution time of the unconstrained case (u), i.e. $u/e - 1$.
- $R(\Omega_{avg})$ *and* $R(\Omega_{max})$*:* the ratios of space utilization over the space utilization of the unconstrained case.

5.3.3 Performance Evaluation

In this section, the performance of the various parallelism control models against the unconstrained model are measured. The baseline performance given by the unconstrained model is given in Table 5.15. The table shows that the processor utilizations range from a low of 15.3% for HILBERT and up to 93.5% for AMR. The low utilization for HILBERT is due to the fact that a typical parallel loop body is relatively small with only one or two threads, and a significant fraction of the time is spent in the serial reductions of those parallel loops. Also, in general, the synchonization unit utilization is in the same order, albeit smaller, as the processor utilization.

Bench	Time	U_P	U_S	Ω_{avg}	Ω_{max}
AMR	2294940	93.5	45.7	3774	22148
FFT	1154821	70.7	36.1	778	7199
HILBERT	2474720	15.3	13.1	259	3730
SDD	2907122	57.2	36.2	1265	11716
SIMPLE	7040310	54.5	38.1	28485	101758
WEATHER	1427385	67.8	54.1	4102	20775

Table 5.15: Performance with unconstrained parallelism (R).

Bench	Chunk Size	U_P%	U_S%	Speedup%
AMR	32	95.2	29.4	17.5
FFT	32	76.8	15.8	34.7
HILBERT	8	15.0	12.8	0.3
SDD	32	55.1	31.9	3.3
SIMPLE	16	51.9	32.1	7.3
WEATHER	16	63.8	46.3	3.5

Table 5.16: Performance with inner loops chunked (C).

Chunked Code.

Chunking exploits data locality and hence should decrease the execution time in addition to restraining parallelism. The speedup of the various benchmarks is shown in the Table 5.16. The best performing chunk size for each benchmark is also presented. The experiment was conducted with chunk sizes of 8, 16 and 32. HILBERT showed the lowest speedup of 0.3%. *FFT* showed the best speedup of 34.7%. The vectorizable loops in HILBERT have much smaller loop bodies than the nonvectorizable loops and therefore the impact of vectorization is minimal. The data access pattern of FFT is very regular and hence highly vectorizable. Since parallelism is controlled only in those loops that can be vectorized, the occupancy of the matching store memory does not show any significant decrease. However, the table shows a noticeable reduction in the synchronization unit utilization. Since the matching is done a chunk at a time rather than a single value at a time, the utilization of the synchronization unit utilization should be reduced.

Sliced Codes.

Slicing is used to control matching store memory occupancy. All parallel loops, including vectorizable loops, in the benchmarks have been sliced in this experiment. The experiment has been conducted with the loops sliced with 10 or 20 workers each. In this experiment, space refers to matching store memory.

In most of these experiments, we are trading time for space. The more the parallelism is throttled, the less space it uses and the more time it takes to complete the execution in general. This is evidenced in Table 5.17 where the slice sizes are chosen that favor execution time over the space, and in Table 5.18 where the space is favored over the time. In the tables, slice sizes are specified in terms of the number of worker processes that are used to execute the parallel loops. For instance, slice size 10,20 indicates that the innermost parallel loop would be split up between 10 worker processes and the outer, second level parallel loop is split up between 20 worker processes.

Table 5.17 shows that the processor utilizations is approximately equal to that of the unconstrained case. Good processor utilization implies that the parallelism has not been throttled to the extent where the processor is sitting idle when it should not. The speedup ranges from -5.4% in SDD to -18.1% in HILBERT. The average space used drops dramatically for the experiment. The best saving, in the average case, is shown by SIMPLE which utilizes just 5.8% of that required by the unconstrained (R) case. The worst is WEATHER which requires 78.0% of the space occupied in the unconstrained case. When maximal occupancy is considered, the best saving is still shown by SIMPLE, utilizing just 8.7% of the maximum occupied in the unconstrained case. The worst saving is shown by WEATHER, saving just 13.3% of the space.

Bench	Slice Size	U_P%	U_S%	Speedup%	$R(\Omega_{avg})$%	$R(\Omega_{max})$%
AMR	10,20	93.8	58.5	-8.2	28.8	15.1
FFT	20,20	70.4	52.0	-12.5	21.5	10.9
HILBERT	20,20	16.8	18.2	-18.1	60.2	77.4
SDD	20,20	64.2	49.2	-5.4	73.2	33.5
SIMPLE	20,20	51.5	41.1	-12.5	5.8	8.7
WEATHER	20,20	74.9	64.4	-8.8	78.0	86.7

Table 5.17: Performance with loops sliced, best execution times.

Bench	Slice Size	U_P%	U_S%	Speedup%	$R(\Omega_{avg})$%	$R(\Omega_{max})$%
AMR	10,10	92.3	57.6	-9.5	14.9	8.6
FFT	20,10	69.5	51.3	-12.5	15.5	6.4
HILBERT	20,10	15.4	16.6	-24.9	48.2	39.9
SDD	20,10	60.0	46.0	-11.6	51.6	26.2
SIMPLE	20,10	48.8	39.0	-16.9	2.9	4.2
WEATHER	10,10	60.4	51.7	-19.2	29.3	44.6

Table 5.18: Performance with loops sliced, best occupancy.

Bench	Chunk Size	Slice Size	Speedup%	$R(\Omega_{max})$ %
AMR	32	10,20	+8.9	15.2
FFT	32	20,20	+30.5	5.4
HILBERT	16	20,20	-16.8	77.4
SDD	32	20,20	-0.6	43.4
SIMPLE	16	20,20	-1.6	5.9
WEATHER	16	20,20	-0.6	86.6

Table 5.19: Performance with inner loops chunked and loops sliced, best processor utilization.

Table 5.18 shows the results for those cases where processor utilization remains fairly high but the space utilized is the lowest. In this case, the speedups drop even more with the range of -9.5% in AMR to -24.9% in HILBERT. In the average space case, the best saving is shown by SIMPLE, requiring just 2.9% of the space. The worst is shown by SDD, requiring 51.6% of the regular space. In the maximum space case senario, the best savings is again shown by SIMPLE with 4.9% ratio and the worst savings is shown by WEATHER with 44.6% ratio.

Sliced and Chunked Codes.

Since the best speedups and the best space savings have been extracted, the next logical step is to combine the two and try for good execution times with low space utilization.

Table 5.19 shows the combination of the chunking size from Table 5.16 and the slice parameters from Table 5.17 that favors the execution time over the space saving. For the problem sizes used in this section, the space saving seems to give a good trade off with respect to the time taken to solve the problem. In the case of WEATHER the space saving is of the order of 13% over the unbounded case for a time slow down of 0.6%.

Table 5.20 shows the combination of the chunking size from Table 5.16 and the slice parameters from Table 5.18, that favors the space saving over the execution time. This table shows greater savings in the matching store space utilization. However, as expected, the time taken to solve the problem is greater. One extreme case is WEATHER where space usage is about half of that shown in Table 5.19, but the execution took almost 20% longer. In the case of FFT, the speedup is 27.8% while only using 5.3% of the space used by the uncostrained case.

Bench	Chunk Size	Slice Size	Speedup%	$R(\Omega_{max})$%
AMR	32	10,10	+7.7	8.1
FFT	32	20,10	+27.8	5.3
HILBERT	16	20,10	-23.8	39.9
SDD	32	20,10	-7.8	32.1
SIMPLE	16	20,10	-6.2	3.5
WEATHER	16	10,10	-18.7	44.7

Table 5.20: Performance with inner loops chunked and loops sliced, best occupancy.

Exp.	Chunk	Slice	Speedup%	U_P%	U_S%	Ω_{avg}	Ω_{max}
R	-	-	1	70.7	36.1	778	7199
C	8	-	32.5	78.2	18.7	411	7199
	16	-	34.3	77.5	16.8	386	7203
	32	-	34.7	76.8	15.8	371	7203
S	-	10	-43.0	45.8	33.5	228	4380
	-	20	-11.9	70.9	52.2	428	5600
	-	20,10	-13.7	69.5	51.3	121	461
	-	20,20	-12.5	70.4	52.0	167	783
CS	32	20,20	30.5	77.3	23.1	79	391
	32	20,10	27.8	75.7	22.6	71	387

Table 5.21: FFT: Arriving at the Best Throttle Value.

Overall, when chunking is used along with slicing there is an improvement in the space utilization, and in some cases, an improvement in execution time as well. If chunking is effective, then the combination of chunking and slicing offers the best of the both world. This is evidenced for AMR and FFT. The choice of throttling parameters depends on the system size. However, an important point to be made is that the upper bound on the space utilized is determined by the base slicing scheme used and should no longer be dependent on the problem size.

Determining Throttle Values

In this section, we describe the method used to arrive at the best throttle parameters. For this we have chosen the benchmark `FFT`. Table 5.21 shows the results of various experiments executed for the `FFT` benchmark.

The first experiment is for the unconstrained parallelism case. Results indicate that processor utilization is about 71% and the maximum occupancy to be 7199 thread slots in the matching memory.

The next set of experiments is run for the three different chunk sizes of 8, 16 and 32. As expected, the matching store occupancy does not change much in all the cases. In this particular case, the table indicates that all three chunk sizes yield similar performance with the chunk size 32 being the best at 34.7% speedup. Processor utilization remains fairly uniformly high in all the cases. The synchronization unit utilization drops by half, representing a significant reduction.

In the third experiment, the best slicing levels are determined. The first runs only sliced the innermost parallel loops. As stated earlier, the slice size is expressed in the number of workers that are chosen for parallel loops at the nesting level specified. The sizes chosen were 10 and 20. When the number of worker processors is 10, the throttle is too strong and processor utilization drops to 45.8%. The processor utilization with 20 workers remains fairly high at 70.9% and hence is better. The next step is to slice the parallel loops at the second level also, using the inner loop slice of 20. The experiments were run for 20,10 and 20,20. Processor utilization for both runs remain fairly high. Slicing with 20,20 is faster but slicing with 20,10 has a greater space saving.

The last set of experiments combines the chunking and slicing techniques, which will give us an acceptable execution speed at a low resource utilization. Two experiments are run for chunk size 32 and loop slicing 20,20 and 20,10 respectively. In this case, a chunking factor of 32 and a loop slicing factor of 20,20 would probably provide the best balance between the execution time and the space utilization. The choice may not always be this clear cut.

Chapter 6

CONCLUSION

The popularity of shared memory programming models for distributed memory machines is growing. Explicitly parallel languages such as HPF require the help of programmers (e.g., in specifying data distribution) in order to detect parallelism. Implicitly parallel languages such as Sisal make it easier to detect parallelism.

Multithreading is a promising approach that works in conjunction with shared memory programming models that can tolerate latency. It provides an efficient mechanism to overlap communication with computation resulting in improved processor utilization. Furthermore, the approach exploits irregular and dynamic parallelism and can tolerate "bad" programming. For example, bad data mappings by programmers, resulting in more frequent remote memory reads, can be better tolerated under multithreading.

This thesis explores multithreading at the junction where architectures and compilers meet, rather than at the programming language or the machine engineering levels. Several aspects of multithreaded executions are examined and, for each aspect, different solutions are explored.

Clusters and Bottom-up Code Generation. As a first step toward multithreading, a von Neumann/dataflow hybrid model is evaluated. Issues considered include bottom-up code generation, instruction level characteristics, and performance for a variety of machine configurations. The concept of *clusters* was used as a model of medium-grain execution. A cluster can be thought of as a mini-dataflow graph which must be executed strictly and deterministically.

Two bottom-up algorithms for generating cluster code from the Manchester dataflow graph, namely *Basic Block* (BBA) and *Dependence Sets* (DSA), were developed and evaluated. The algorithms attempt to avoid potential deadlocks, to preserve any loop and function level parallelism, and to maximize the exploitation of instruction level locality within a cluster.

The results from the simulated execution of these clusters show a significant reduction in the number of matches per instruction executed: from 1.78 in the fine grain dataflow model to 1.28 for BBA and 1.31 for DSA in the cluster model. Measurements on an extensive set of 50 benchmark programs indicate that an average cluster has 4.0 Manchester instructions and 3.0 inputs for the BBA and 2.2 instructions and 2.5 inputs for the DSA. Overall, the small size of clusters has led to the development of a hybrid code generation approach.

The simulation results also indicate that cluster execution achieves a significant speedup over fine-grain execution and it requires less resources such as token buffer and match memory. Due to the small size of clusters, achievable speedup is limited. For example if each thread contains two instructions, then the maximum speedup is only two, assuming an infinite number of processors in both models and assuming that instructions within each thread can execute concurrently. A larger thread size can exploit more locality and also reduce the overhead.

On the other hand, bigger threads do not always result in a higher speedup, as the comparison of BBA and DSA methods show. Creating a larger cluster can increase input latency, thus delaying the execution of the whole computation. If this cluster is on the program's critical path, the total execution time could suffer. Therefore, it is important not only to increase cluster thread size but to minimize input latency.

Pebbles and Hybrid Code Generation. The experiments in cluster execution provide the motivation to develop a new multithreaded execution model and a hybrid code generation strategy. The Pebbles execution model has non-blocking threads and uses framelets as a basic storage unit. Code is initially generated top down from a program data-dependence graph and then optimized in a bottom-up method. The optimization techniques include local and global scopes. Local optimizations consist of traditional techniques such as dead code elimination and copy propagation, while global (inter-procedural) optimizations, which follow a bottom-up style, include merging of multiple threads into one thread. The objectives of the global optimizations are to increase the thread size and reduce the cost of synchronization.

Dynamic quantitative intra-thread measurements on the optimized code yield: (1) an average thread size of to 16.2 instructions per thread, (2) an average parallelism within a thread of 3.6 instructions per cycle, (3) an average number of inputs per thread of 7.6, and (4) an average synchronization cost of 0.47 matches per instruction.

Optimizations at the program level produces the following reductions: 31% in the total number of matches, 23% in the total number of instructions executed, 17% in the total thread count and 18% in the average critical path of programs (measured in threads). The total execution time was decreased by about 25-30%.

The overall results indicate a good match between the Pebbles model and thread characteristics (such as the thread size and internal parallelism) generated by the code generator; thus this is a viable approach to multithreading.

Analysis and Cost Reduction Techniques. Other aspects of multithreading are examined, with emphasis on identifying and quantitatively measuring each important aspect and developing strategies that can improve the overall performance. The issue of aggregate inter-thread locality in a multithreaded execution is first addressed. A storage hierarchy for the Pebbles model is developed to take advantage of locality and to reduce the cost of synchronizations.

The results show that the reference stream contains a great deal of locality: a cache size of 256K is sufficient to bring the capacity miss rates below 1% for most benchmarks. By varying the cache replacement policy, it is found that FIFO performance is almost identical to that of LRU, thus indicating a reference stream pattern in which the *least recently* referenced block is often the *earliest* referenced one.

A scheme that significantly reduces capacity miss rates is introduced, which keeps a list of free-blocks in the Framelet Cache. The reduction results from avoiding replacement of cached framelets wherever possible. With low capacity misses, compulsory misses become dominant. To deal with this, a scheme was devised where one free cache block, called a reserve block, is kept on reserve and used when a compulsory miss is detected. Hence, the cost is reduced to nearly that of a cache hit when there exists a reserve block. The resulting scheme produces *total* miss rates is well below 1% for all but one benchmark. The above schemes reduce the effective total miss rates by a factor of 6 to 2400, and require a smaller cache size.

Another multithreading issue examined concerns inter-processor communications. For this purpose, the sources of inter-processor communications are measured and their utilities are determined. Then input locality is introduced and studied to observe the temporal relationship among inputs in a thread. This data may help guide scheduling operations and help in the design of hardware. Finally, output locality of threads is introduced and measured.

The results indicate that roughly one-third of the messages read and write to the structure store, another third are involved in parallelism management, and the final third are involved in communication within a function or loop body. However, there are some significant differences among benchmarks. Also, there is a significant degree of locality among thread inputs; in about 30% of the cases, all inputs to a thread arrive within 10 cycles of each other. This indicates that a cache can be efficiently utilized and confirms the earlier results in the study of the memory hierarchy. In addition, a thread typically sends 3 or 4 tokens to each target thread, implying a relatively high degree of output locality.

Based upon these results, two simple techniques called *grouping* and *bundling* are studied. They are designed to reduce the amount of inter-processor communications and the associated overheads by exploiting communication locality. The *grouping* technique eliminates communication between threads within a function or loop body by assigning these threads to the same processor. The *bundling* technique vectorizes messages sent to the same processor in order to reduce both the communication and overhead costs. The overall results indicate that about 80% of non-memory related inter-processor messages can be eliminated, and that more than 50% of all inter-processor messages are eliminated.

Finally, this thesis explores ways to control parallelism and to take advantage of data locality. Chunking and slicing techniques are introduced that address the above issues. Slicing works by generating a variable number of sequential tasks to a fixed number of workers. Thus, maximum space utilization is made problem size-independent; however, the execution time increases. Chunking in essence vectorizes the inner loop by sending and receiving a chunk of data from the structure store. Thus, the execution time is improved and the load on the synchronization unit is reduced; however, the resource usage minimally affected. The combination of the two complementary techniques can help balance the execution time with the resource usage.

6.1 Future Directions

Multithreading still represents a deeply fertile ground to be explored. As far as code generation/optimization is concerned, there are many more techniques that can applied, including profiling, loop unrolling, and different grouping techniques. In addition, the study of various scheduling policies (software or hardware) could be explored.

Another important area of future research concerns data distribution; proper data distribution can significantly reduce the number of remote reads, thus reducing the dependence on latency tolerance mechanisms. This requires a further static analysis at the language level to determine a proper data partitioning and/or it requires the use of global caching.

REFERENCES

[ABM87] Arvind, S. Brobst, and G. Maa. Evaluation of the MIT tagged-token dataflow architecture. Technical report, MIT, 1987.

[AC86] Arvind and D.E. Culler. Dateflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Portfield, and B. Smith. The Tera computer system. In *Int. Conf. on Supercomputing*, pages 1–6. ACM Press, 1990.

[ACC+91] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Portfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proc. Workshop on Multithreaded computers*. ACM Press, 1991.

[AE90] D. Abramson and G. K. Egan. The RMIT Dataflow Computer: A Hybrid Architecture. *The Computer Journal (British Computer Society)*, 1990.

[AHH88] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating systems and multiprogramming. *ACM Trans. on Computer Systems*, 6:393–431, November 1988.

[AHO90] K. Aziz, M. Haines, and R. R. Oldehoeft. Purdue parallel benchmarks in SISAL (Revised). Technical Report CS-90-101, Colorado State University Computer Science Department, Fort Collins, CO, February 1990.

[AI87] Arvind and R.A. Iannucci. Two fundamentals issues in multiprocessors: the data-flow solutions. Technical Report LCS CSG 226-6, Laboratory for Computer Science, MIT, Cambrige Massachussett, May 1987.

[AN90] Arvind and R. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. on Computers*, 39(3):300–318, March 1990.

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.

[AT80] Arvind and R. E. Thomas. I-structures: An efficient data type for functional languages. Technical Report LCS/TM-178, MIT, Laboratory for Computer Science, 1980.

[BG90] W. Böhm and J. R. Gurd. Iterative instructions in the Manchester dataflow computer. *IEEE Trans. on Parallel and Distributed Systems*, 1(2):129–139, 1990.

[BGT89] W. Böhm, J. R. Gurd, and Y. M. Teo. The Effect of Iterative Instructions in Dataflow Computers. In *Int. Conf. on Parallel Processing*, 1989.

[BNR89] L. Bic, M. Nagel, and J. Roy. Automatic data/program partitioning using the single assignment principle. In *Int. Conf. on Supercomputing*, Reno, Nevada, 1989.

[BNSR93] W. Böhm, W. A. Najjar, B. Shankar, and L. Roh. An evaluation of coarse-grain dataflow code generation strategies. In *Working Conference on Massively Parallel Programming Models*, Berlin, Germany, 1993.

[Can89] D. C. Cann. Compilation techniques for high performance applicative computation. Technical Report CS-89-108, Colorado State University, 1989.

[CCMH91] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W. W. Hwu. Computing static and Dynamic Code Scheduling for Multiple-Instruction-Issue Processors. *Proc. 24th Int. Symp. Microarch.*, 1991.

[CF89] D. C. Cann and J. T. Feo. SISAL 1.2: An alternative to FORTRAN for shared memory multiprocessors. Technical Report UCRL-102263, Lawrence Livermore National Laboratory, Livermore, CA, November 1989.

[CGSvE93] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. Tam – a compiler-controlled threaded abstract machine. *J. of Parallel and Distributed Computing*, 18:347–370, March 1993.

[CSS+91] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[CSvE93] D. E. Culler, K. E. Schauser, and T. von Eicken. Two fundamental limits on dataflow multiprocessing. In Cosnard, Ebcioglu, and Gaudiot, editors, *Proc. IFIP WG 10.3 Conf. on Architecture and Compilation Techniques for Medium and Fine Grain Parallelism*, Orlando, FL, 1993. North-Holland.

[Cul85] D.E. Culler. Resource management for the tagged token data flow architecture. Technical Report TR-332, Laboratory for Computer Science, MIT, January 1985.

[Cul89] D. E. Culler. *Managing parallelism and resources in scientific dataflow program.* PhD thesis, MIT, June 1989.

[Cul94] D. E. Culler. Multithreading: Fundamental limits, potential gains, and alternatives. In R. Iannucci, editor, *Multithreading: A Summary of the State of the Art*, pages 97–138. Kluwer, 1994.

[DFK+92] W. J. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[EG91] P. Evripidou and J.-L. Gaudiot. *The USC Decoupled Multilevel Data-Flow Execution Model.* Prentice-Hall, 1991.

[Feo87] J. T. Feo. The Livermore Loops in Sisal. Technical Report UCID-21159, Computing Research Group, Lawrence Livermore National Laboratory, Livermore, CA 94550, August 1987.

[GDHH89] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The ϵpsilon dataflow processor. In *Proc.* 16^{th} *Ann. Int. Symp. on Computer Architecture*, pages 36–45, Albuquerque, New Mexico, 1989. Sandia National Laboratories.

[GE91] J-L. Gaudiot and L. Bic (Editors). *Advanced Data-Flow Architectures.* Prentice-Hall, 1991.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester Dataflow prototype. *Communications of the ACM*, 28(1):34–52, 1985.

[GN88] J.-L. Gaudiot and W. A. Najjar. Macro-actor execution on multilevel data-driven architectures. In *Proc. of the IFIP Working Group 10. 3 Working Conference on Parallel Processing*, pages 277–290, Pisa, Italy, 1988.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[GP92] M. Girkar and C. D. Polychronopolous. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–177, 1992.

[Gro83] T. R. Gross. Code Optimization Techniques for Pipelined Architectures. *Proc. IEEE Computer Society Spring Int. Conf.*, pages 278–285, 1983.

[HDGS92] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile time partitioning of a non-strict language into sequential threads. Technical report, Sandia National Laboratory, 1992.

[HG93] H. H. Hum and G. R. Gao. Supporting a dynamic SPMD model in a multithreaded architecture. In *Proc. of Compcon Spring'93*, 1993.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[HTG94] H. H. Hum, K. B. Theobald, and G. R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proc. Int. Parallel Processing Symp.* IEEE CS Press, 1994.

[Ian88] R. A. Iannucci. Toward A Dataflow/Von Neumann Hybrid Architecture. In *Proc.* 15^{th} *Ann. Int. Symp. on Computer Architecture*, pages 131–140, 1988.

[Ian90] R. A. Iannucci. *Parallel Machines: Parallel Machine Languages*. Kluwer, 1990.

[Jou90] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Ann. Int. Symp. on Computer Architecture*, pages pp. 364–373, May 1990.

[Lam88] M. Lam. Software pipelining: An effective scheduling techniques for VLIW machines. *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 318–328, 1988.

[McM86] F. H. McMahon. Livermore FORTRAN kernels: A computer test of numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[MNB91] W. M. Miller, W. A. Najjar, and W. Böhm. A quantitative analysis of locality in dataflow programs. In 24^{th} *Int. Symp. on Microarchitecture (MICRO-24), Albuquerque, NM*, pages 46–53, November 1991.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language: reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[Muc88] S. S. Muchnick. Optimizing Compilers for SPARC. *Sun Technology, Summer*, pages 64–77, 1988.

[NA89] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc.* 16^{th} *Ann. Int. Symp. on Computer Architecture*, pages 262–272, 1989. Also: CSG Memo 292, MIT Laboratory for Computer Science 545 Technology Square, Cambridge, MA 02139, USA.

[NA92] R. S. Nikhil and Arvind. Id: a language with implicit parallelism. In J.T.Feo, editor, *Special Topics in Supercomputing Volume 6. A Comparative Study of Parallel Programming Languages: The Salishan Problems*, pages 169–215. North-Holland, 1992.

[Nik90] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990. Supercedes: Id/83s (July 1985) Id Nouveau (July 1986), Id 88.0 (March 1988), Id 88.1 (August 1988).

[Nik93] R. S. Nikhil. A multithreaded imlementation of Id using P-RISC graphs. In *6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[NMB92] W. A. Najjar, W. M. Miller, and W. Böhm. An Analysis of Loop Latency in Dataflow Execution. In *Proc.* 19^{th} *Ann. Int. Symp. on Computer Architecture*, pages 352–361, Gold Coast, Australia, 1992.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. 19^{th} Ann. Int. Symp. on Computer Architecture*, pages 156–167, 1992.

[NRB93] W.A. Najjar, L. Roh, and W. Böhm. The Initial Performance of a Bottom-Up Clustering Algorithm for Dataflow Graphs. In *Proc. IFIP WG 10.3 Conf. on Architecture and Compilation Techniques for Medium and Fine Grain Parallelism*, pages 91–102, Orlando, FL, 1993.

[NWD93] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-Machine Multicomputer: An architectural evaluation. In *Proc. 20^{th} Ann. Int. Symp. on Computer Architecture*, May 1993.

[Pap88] G. M. Papadopoulos. Implementation of a General-Purpose Dataflow Multiprocessor. Technical Report TR-432, MIT Laboratory for Computer Science, August 1988.

[PC90] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proc. 17^{th} Ann. Int. Symp. on Computer Architecture*, pages 82–91, June 1990.

[PT91] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proc. 18^{th} Ann. Int. Symp. on Computer Architecture*, pages 342–351, June 1991.

[Ric85] J. R. Rice. Problems to test parallel and vector languages. Technical Report CSD-TR 516, Purdue University, 1985.

[RN95] L. Roh and W. A. Najjar. Analysis of communication and overhead reduction in multithreaded execution. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, 1995.

[RNB93] L. Roh, W. A. Najjar, and W. Böhm. Generation and Quantitative Evaluation of Dataflow Clusters. In *Proc. Symposium on Functional Programming Languages and Computer Architecture*, pages 159–168, Copenhagen, Denmark, 1993.

[RNSB94] L. Roh, W. A. Najjar, B. Shankar, and A. P. W. Böhm. An evaluation of optimized threaded code generation. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'94)*, Montreal, Canada, 1994.

[Roh92] L. Roh. IDIAS: a dataflow machine simulator. Technical Report CS-92-112, Colorado State University, Computer Science Department, 1992.

[RS87] C. A. Ruggiero and J. Sargeant. Control of Parallelism in the Manchester Dataflow Computer. In *Lecture Notes in Computer Science no. 274*, pages 1–15, 1987.

[Sar87] V. Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. Technical Report CSL-TR-87-328, Stanford University, Computer Systems Laboratory, April 1987.

[SCvE91] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In J. Hughes, editor, *Proc. Symposium on Functional Programming Languages and Computer Architecture*, 1991.

[SG85a] S. K. Skedzielewski and J. Glauert. IF1—an intermediate form for applicative languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[SG85b] S. K. Skedzielewski and John Glauert. IF1: An intermediate form for applicative languages reference manual, version 1. 0. Technical Report TR M-170, Lawrence Livermore National Laboratory, July 1985.

[SKS+92] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proc. 19^{th} Ann. Int. Symp. on Computer Architecture*, 1992.

[Smi81] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE (Real Time Signal Processing)*, 298:241–248, 1981.

[Sne93] David F. Snelling. *The Design and Analysis of a Stateless Data-Flow Architecture*. PhD thesis, Computer Science Department, University of Manchester, Manchester, UK., 1993.

[SR86] K. So and R. N. Rechtschaffen. Cache operations by MRU-Change. *Intl. Conf. on Computer Design*, pages pp. 584–586, October 1986.

[SYH+89a] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a data-flow single chip processor. In *Proc.* 16^{th} *Ann. Int. Symp. on Computer Architecture*, pages 46–53, May 1989.

[SYH+89b] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Proc.* 16^{th} *Ann. Int. Symp. on Computer Architecture*, pages 46–53, 1989.

[SYH+91] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. Pipeline optimization of a dataflow machine. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 225–246. Prentice Hall, 1991.

[Tak87] M. Takesue. A unified resource management and execution control mechanism for data flow machine. In *Int. Ann. Symp. on Computer Architecture*, pages 90–97. ACM, 1987.

[Tak92] M. Takesu. Cache memories for data flow machines. *IEEE Trans. on Computers*, 41(6):677–687, June 1992.

[TCS92] K. Traub, D. E. Culler, and K. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Proc. Conf. on LISP and Functional Programming*, 1992.

[TE94] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proc.* 21^{th} *Ann. Int. Symp. on Computer Architecture*, Chicago, Illinois, 1994.

[THG95] K. B. Theobald, H. H. Hum, and G. R. Gao. A design framework for hybrid-access caches. In *Int. Symposium on High-Performance Computer Architecture*, pages pp. 144–153, January 1995.

[Tra91] K. R. Traub. Multithread code generation for dataflow architetures from non-strict programs. In J. Hughes, editor, *Proc. Symposium on Functional Programming Languages and Computer Architecture*, 1991.

[Vas94] J. Vasell. A Fine-Grain Threaded Abstract Machine. In *Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'94)*, pages 15–24. North-Holland, 1994.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc.* 19^{th} *Ann. Int. Symp. on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[Wal93] D. W. Wall. Limits of instruction-level parallelism. Technical Report WRL Research Report 93/6, Western Research Laboratory, Palo Alto, CA, November 1993.

[WF92] R. Wolski and J. Feo. Program partitioning for NUMA multiprocessor computer systems. Technical report, Lawrence Livermore National Laboratory, 1992.

[WG89] W. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proc.* 16^{th} *Ann. Int. Symp. on Computer Architecture*, pages 273–280, May 1989.

[WS87] S. Weiss and J. E. Smith. A study of scalar compilation techniques for pipelined supercomputers. *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 105–109, 1987.

[WSYR86] M. Welcome, S. Skedzielewski, R. K. Yates, and J. Ranelleti. IF2: An applicative language intermediate form with explicit memory management. Technical Report TR M-195, University of California - Lawrence Livermore Laboratory, December 1986.

[YSHK85] T. Yuba, T. Shimada, K. Hirakai, and H. Kashiwagi. Sigma-1: A dataflow computer for scientific computations. In *Computer Physics Comunications*, pages 141–148, 1985.