The Design and Analysis of a Stateless Data-Flow Architecture

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science

April 1993

David F. Snelling

Department of Computer Science
Technical Report Number UMCS-93-7-2
July 1993

To My Father

Table of Contents

Abstract

Data-Flow computing is approximately 20 years old, and progress in that time has been steady but not dramatic. As new technologies have emerged they have been incorporated into new Data-Flow systems, however, Data-Flow has held onto many of its original tenets. One of these tenets, contrary to the Data-Flow ideal, is the use of explicit state for storing data structures, arrays in particular. By imposing a shared memory model, this explicit notion of state creates a fundamental barrier preventing traditional Data-Flow systems form being scalable.

In this thesis, a dynamic, tagged token Data-Flow architecture, SDFA (Stateless Data-Flow Architecture), is presented which diverges from other systems in that it does not support an explicit notion of state. In the absence of explicit state, it is possible to exploit the advantages of Data-Flow, such as uniform load balancing, latency hiding, and good performance on small problems, within a distributed memory architecture. With the shared memory barrier removed, the SDFA system is scalable.

The thesis sets out the case for a stateless model, develops an architecture based on this approach, presents a detailed, simulation based analysis of the system, and proposes further research into stateless Data-Flow computing.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

List of Figures

The Author

The Author received his Bachelor and Masters of Science from the University of Denver, Denver Colorado in 1979 and 1981, respectively. Since that time he has worked in industry and academe researching and teaching parallel processing. He has published frequently and edited one book on the subject.

The Story

Once upon a time, more performance was needed for running scientific applications, and there were three ways of getting it. Reduce the operation count. Reduce the cycle time. Go parallel. Data-Flow chose to go parallel.

Years passed. Memory Hierarchy became a lord in the kingdom. Data-Flow machines, like vector machines, didn't pay their taxes to the new lord. The RISC serfs did.

Then it came to pass that wise men realized that Data-Flow faced imprisonment, owing to its traditional belief in explicit state which prevented it from paying taxes. SDFA, a young Data-Flow monk, abandoned explicit state and paid taxes to the lord.

The price could be high; he could go to hell.

Acknowledgements

I would like to thank my supervisor, John Gurd, for his advice and direction. Thank you, also, to my proof-readers Dr. Mike O'Boyle and Dr. Clarence Snelling, and members of the Centre for Novel Computing. Thank you Penny and Tobin for tolerating me while I was writing. And a special thanks to DonDon.

Chapter 1
Introduction

1.1. Motivation

Since the time of Newton, science has improved steadily in its ability to model the universe accurately. In the last two decades, this improvement has been accompanied by a sudden and substantial increase in compute power. This combination has resulted in several new technologies which fuel the need for even more computational resources. Primary examples include: visualisation, which has increased the volume of data that a single individual can interpret; interactive development, which has shortened the development cycle for new ideas; and integrated VLSI processors, which have greatly reduced the cost of computing. Together these, and other factors, ensure a growing demand for ever faster computer systems.

This ever increasing need can be addressed in three ways.

First, the number of operations, required to solve a given problem, can be reduced.[1]

Second, the cycle time of the computer can be reduced, thus, shortening the time to solution.

Last, parallelism can be introduced into both the application and the computer system.

The first two approaches are limited in their ability to satisfy continuing demand. Once a compiler has removed all except the necessary operations, no further improvement is possible; many algorithms, now in use, are highly efficient; at some point compromise results in inaccuracy. Likewise, the speed of light is fixed, as is the size of an atom and, hence, a gate must occupy some minimum area. Therefore, cycle times have a fixed minimum.[2] Only parallelism is seen as a limitless resource, which, properly exploited, has the potential to yield ever improving performance. Under certain circumstances (see Section 1.2.2) parallelism, too, has its limits. Parallelism can only be exploited effectively if the system and software can overcome the time delay, or *latency*, associated with communication and synchronization. Also, efficient parallel processing is possible only when the computation can be scheduled uniformly across multiple processors. These issues are presented in Section 1.2.

Data-Flow, in common with many branches of computing, uses parallelism as the primary mechanism to achieve high performance. However, during the past 20 years, since the earliest work on Data-Flow architecture,[3] implementation technology has changed dramatically. Primarily, the difference in speed between the processor and the

---

1 Compiler optimizations, algorithm development, and compromises in the science contribute to reducing the number of calculations that need to be performed.

2 Although persuasive, this argument seems to be defeated, in practice, every year or two. Nonetheless, the view that clock speeds cannot increase forever has contributed to the motivation behind parallel computing.

3 The earliest theoretical work on Data-Flow dates back into the 1960s [Karp66a], but the first implementations weren't being discussed until the 1970s [Denn73a, Gurd77a, Davi78a,].

main memory has widened significantly. In addition, the increased density of VLSI has allowed the memory and processor to be packaged on the same chip, thus, effectively widening the gap further.

The importance of memory hierarchies has, therefore, grown. Systems are common today which have three, or more, levels in their memory hierarchies. These systems rely on the ability of the software and programmer to schedule the computation such that data near the top of the memory hierarchy is accessed more frequently than data farther down.

Data-Flow research, however, has continued to develop only shallow memory hierarchies. Today, even where memory hierarchies exist, the underlying Data-Flow model assumes an explicit notion of state. Data structures, stored in this explicit state, appear identical to all processors in the system, thus, imposing a shared memory model on the architecture. This shared memory model presents a fundamental barrier to the effective exploitation of deep memory hierarchies. These ideas are developed fully in Chapter 2.2.

In this study, the notion of explicit state is abandoned, and a Stateless Data-Flow Architecture (SDFA) is developed. SDFA is a scalable distributed memory architecture, with a four level memory hierarchy and facilities to exploit that hierarchy. Abandoning state has the potential for increasing the cost of the system and the complexity of the application software; therefore, a careful analysis of the architecture is also undertaken.

The remainder of this chapter is devoted to a summary of the fundamental requirements for achieving efficient parallel performance. These issues are relevant to all branches of parallel computing, including Data-Flow.

1.2. Fundamentals

Once parallelism is identified in an application, there are three issues central to achieving high performance in parallel systems.

First, latency, due either to remote memory accesses or synchronization, must be tolerated.

Second, the hierarchical nature of memory technology must be exploited, as far as possible, to reduce the need for latency tolerance.

Last, mechanisms must exist for controlling, scheduling, and evenly distributing computation such that processor resources are effectively utilized.

These issues, and the associated approaches to addressing them, are presented in the following section.

1.2.1. Latency Tolerance

Since the early 1980s, researchers and engineers have tried to isolate the key factors affecting performance of parallel systems. Arvind and Iannucci's seminal paper, "Two Fundamental Issues in Multiprocessing" [Arvi87a], focuses on two forms of latency, memory access latency and latency due to synchronization. A third form of latency, included here for generality, is pipeline latency.

Memory access latency is the time needed to fetch a data item from somewhere farther down a memory hierarchy.[4] At the top level of the hierarchy, access to data is part of the arithmetic pipeline and usually takes one cycle. Requests for data farther down the hierarchy may take several cycles to perform.

In pipeline latency, data values are unavailable because they have yet to emerge from the arithmetic pipeline. Since high performance processors usually have relatively long arithmetic pipelines, this can be a significant factor contributing to a system's failure to achieve its potential level of performance.

Memory access and pipeline latency are common in all computer systems. In parallel systems, synchronization causes an additional form of latency. Synchronization involves communication between processors and may require at least one of the processors to wait.

The approach to tolerating latency is universal; latency is hidden behind (or masked by) other computational tasks. The following are several approaches employed to mask latency.

Non-Blocking-Reads provide a simple mechanism for masking a small amount of memory access latency, although they do not address the other forms of latency. When a non-blocking-read is issued, instructions following the read can continue to execute while the read is outstanding, as long as they do not require the data value fetched by the read. This facility is provided on most systems and has been available for many years [Hama90a].

Pre-Fetch has two varieties.
The first allows a programmer, or compiler, to issue a request for an item of data in advance of its use. Data is thus moved up the memory hierarchy so that, when it is eventually needed, the delay is shorter. This applies to many systems, both parallel and sequential.

The second form applies to parallel systems where the desired data is on another processor. This form of pre-fetch moves the data to the processor that needs it in advance of the actual request. This feature is common in systems supporting a virtual shared memory model [KSR91a, Leno90a].

Communication-Overlapping serves the same purpose as pre-fetch on systems employing an explicit message passing protocol for data exchange. The programmer, in the knowledge that a collection of data values is already available on one processor and required later on another, initiates the transfer early and overlaps it with computation on both processors. This approach is common in distributed memory systems, such as those built from the INMOS Transputer [Inmo89a].

Scalar-Pipelining accomplishes, for computational pipeline latency, what non-blocking-reads accomplish for memory access latency. Instructions are scheduled, by the programmer, compiler, or special hardware, and executed concurrently in different stages of the pipeline. This practice is common in modern RISC systems [Bako90a, Hama90a].

---

4 Any hierarchy can be viewed from either direction. In this case it is assumed that the storage closest to the computational function units is at the top of the hierarchy. This is the terminology used by [Hama90a] in a computer architecture text.

Vector-Pipelining takes advantage of the fact that a large number of applications require the same operation to be performed on many data values. This parallelism provides a simple method of masking both memory access latency and pipeline latency. This technique is most effective in uni-processors, but memory designs become increasing complex if multiple processors are used [Hwan89a, Hock88a].

Multi-Threading is a generalization of other latency hiding approaches. Rather than pipelining independent operations of a single thread[5] of execution, as in both scalar and vector pipelining, multiple threads of execution are maintained simultaneously, and instructions from independent threads are interleaved in the execution pipeline. This approach can be extremely effective at masking all three types of latency, when there is adequate parallelism available. The earliest multi-threaded system was the Denelcor HEP [Smit78a, Dene81a]. Others have emerged since [Smit90a, Cull92a, Nikh92a].

Data-Flow-Pipelining takes the pipelining approach to its logical extreme, effectively treating each instruction as a separate thread. Since any enabled instruction can execute in the next pipeline slot, all types of latency are masked, in all parts of the computation, and throughout the machine. As with multi-threading, there must be adequate parallelism available to fill the pipelines.

All these techniques can be used to hide the latency that exists in a computation. There is a limit to the effectiveness of latency hiding. Therefore, efforts are made by programmers and compilers to create locality in applications, thus exploiting the memory hierarchies within architectures and actively reducing latency.

1.2.2. Latency Reduction

Memory hierarchies arise because of differences in the technology used to implement memory components. At the top of the hierarchy, resister files, frequently multi-ported, are implemented using fast, static memory cells. These cells are built using transistors, and the multiple ports are supported by replicating data paths. This results in a low density, high power memory. Therefore, only a small quantity of it can be put into a processor. The second level of the hierarchy, usually cache, is also built from static memory, but slower and requiring less power. Packaging can be more dense and the size greater. The bottom of a typical memory hierarchy is occupied by dynamic memory which, since its cells consist of a single transistor and capacitor, are much more dense.[6] They are substantially slower, partly due to the basic speed of the technology and partly due to the need to be refreshed. At all levels, the trade-off between size and speed is also driven by addressing requirements. The larger a memory the longer it takes to address a particular element [Hwan89a].

Since faster memories require more space than slower memories, only a small amount of data can be kept close to the computational function units, where speed is necessary. Thus, in order to exploit a memory hierarchy, data must be systematically moved up the hierarchy used and preferably re-used. This is the process of extracting locality from a computation.[7]

---

5 Thread is a general term referring to an independent stream of instructions and its associated, private state. In fine grain Data-Flow, each instruction is a thread.

6 Magnetic or optical memory can be viewed as another level in the hierarchy.

7 The importance of exploiting locality has been highlighted by applications developers as well as computer architects. Provided with systems exhibiting an explicit memory hierarchy, algorithm

There are two types of locality in computation, temporal and spatial. It is temporal locality that permits the exploitation of memory hierarchies. It is extracted by scheduling the computation so that data is used shortly after it is created, or used repeatedly once it has been moved to the top of the memory hierarchy. However, making effective use of temporal locality, within a single stream of computation, does nothing to remove delays resulting from communication and synchronization between streams. These delays can be reduced if spatial locality is present in the application. Exploiting spatial locality requires scheduling a computation so that data needed by one processor is always "near" that processor. Spatial locality is frequently exhibited directly by the application through its data structures. For example, the data communication patterns in a weather model follow the flow of air in the atmosphere [Hoff88a, Ritc88a].

Effective use of a memory hierarchy can reduce the amount of memory access latency that must be tolerated and, hence, the effectiveness of the system. It has been argued recently (by David Culler in a critique of the "Two Fundamental Issues" argument [Cull92a]), that the cost, in time and processor real estate, of latency hiding mechanisms, poses fundamental limits on the performance of Data-Flow and multi-threaded systems. Therefore, the effective use of a memory hierarchy is essential to high performance computing and not merely desirable.

The essence of Culler's argument is that, in order to mask latency, independent threads of execution must be active simultaneously. Each of these threads occupies some of the processor's limited, top level memory. This sets a maximum on the number of outstanding remote accesses that can be masked with useful computation. Also, returning requests must synchronize with the associated thread, at a cost in time or hardware complexity. This cost is not fixed; rather, it is an increasing function of the number of active threads. For example, in a multi-threaded system, either more register sets are needed or more frequent, time consuming context switches are required. In a Data-Flow system, the size of the matching store needs to increase as longer latencies are encountered.

The trade-off is that either there exists a fixed latency that can be tolerated (thus limiting the size of the system), or continually increasing the number of active threads, used to mask latency, results in increasing synchronization costs (either in hardware or time).

Alternatively, given a fixed number of active threads per processor, the execution time between remote accesses must be increased, and the latency of these accesses decreased, in order to maintain scalable increases in performance. Therefore, a necessary aspect of a scalable system is effective exploitation of a memory hierarchy.

In parallel computation, Data-Flow in particular, this amounts to a scheduling problem. A parallel program is a collection of partially ordered actions.[8] Executing these, potentially concurrent, actions sequentially can, in effect, decrease the number of parallel threads and, hence, the amount of top level memory required. Alternatively, executing too many parallel actions simultaneously, as with Data-Flow's greedy scheduling policy, can cause top level memory to become full (requiring costly "context switching"[9]).

---

developers can achieve substantial performance improvements by exploiting the locality. The best known of these efforts is the development of LAPACK [Dong88a].

8 The Data-Flow graph of a program represents a partial order for the execution of all instructions in the program.

9 In Data-Flow systems, this is usually in the form of a matching store overflow.

1.2.3. Scheduling

Developing effective techniques for scheduling computation in parallel systems is currently the greatest challenge facing system designers. The goals of a scheduling scheme are to ensure correct execution, extract locality, create adequate parallelism for both latency hiding and parallel performance, balance computation across many processors, and avoid explosions in resource consumption.

The scheduling problem exists at all levels of the program, from instruction scheduling, performed by the compiler, to job scheduling, performed by the operating system or system operators. The techniques used at each level are different and specialized to the particular environment. For example, compiler instruction scheduling is static and aims to maintain efficient use of processor pipelines within a fixed number of registers. An operating system schedules jobs dynamically and aims to provide a fair service to all users.

The basic approach to scheduling used by Data-Flow, at all levels, is dynamic and greedy. Any instruction which can be enabled for execution is enabled. This approach leads to resource management problems [Gurd87a, Cull87a]. Solutions, all of which have weaknesses, do exist and are discussed later in this chapter.

Except for instruction and job scheduling, conventional parallel systems provide little or no support for scheduling. The task of scheduling parallel sections of a program is left to the programmer. Even automatic parallelizers request the user to specify a scheduling policy [KSR91b]. Research in this area, however, has made substantial progress [Snel85a, Poly87a, Poly90a, Alve92a, Yang92a].

Many of the issues in scheduling trade-off against each other. Extracting locality causes sequentiality; massive parallelism causes increased memory use; etc. The question of scheduling, therefore, remains a major issue in all branches of parallel computing.

1.3. Summary

Current Data-Flow systems rely heavily on latency hiding to achieve sustained parallel performance, and mechanisms exist which create well balanced computation in multiprocessor systems. However, to date, Data-Flow research has not aimed to reduce latency through the use of memory hierarchies. This thesis explores this issue and is organized as follows.

Chapter 2 surveys Data-Flow and related work with respect to how they address the issues outlined above, highlighting many of the weaknesses in traditional approaches. The chapter concludes with an analysis of explicit state and how it has imposed a fundamental barrier to current Data-Flow computing which prevents the development of scalable systems.

Chapter 3 describes the Stateless Data-Flow Architecture. This includes a general computational model and its application to a physical architecture.

The experimental apparatus of this study is described in Chapter 4, including simulation software and a summary of the benchmark programs used in the evaluation of SDFA.

Chapters 5, 6 and 7 contain a collection of experiments analyzing SDFA, and Chapter 8 is devoted to conclusions and a discussion of further research. There are also three appendices.

Chapter 2
Survey of
Data-Flow Computation

2.1. Survey of Data-Flow Computing

In this section the basic Data-Flow model of computation is presented, followed by a discussion of the perceived weaknesses of Data-Flow computation generally. A classification of key aspects of Data-Flow systems follows, and the section concludes with a summary of current work in Data-Flow and related research.

2.1.1. The Data-Flow Model of Computation

In contrast to the von Neumann model of computation,[10] the execution of an instruction in the data driven, or Data-Flow, model relies on the availability of its operands rather than on a pre-defined sequence. Even in parallel versions of the basic von Neumann model, sequencing of instructions is controlled explicitly by the programmer or compiler. In a data driven system, the compiler generates dependency graphs, which describe how one instruction depends on the results of other instructions and, from these, generates an executable code encapsulating this dependency information. In a Data-Flow system, the selection of instructions, for execution, is performed by the hardware at execution time and is constrained only by the partial order implicit in the program's data dependency graph. The resulting computation is fine grained and exhibits a much higher degree of parallelism than code written for parallel von Neumann machines. The primary advantage of the Data-Flow model is that parallelism is exposed easily and automatically by the compilation system.[11]

In the Data-Flow model, data values, rather than being stored at particular addresses in the system's memory, are *tagged*. The tag includes the address of the instruction for which the particular data value is destined and other information which uniquely identifies the value. The data value, together with its tag, is called a *token*. Tokens in a Data-Flow system have no fixed location but flow through the system existing only from the instruction that created the data value to where that value is used in further computation.

In order for an instruction, requiring two operands to execute, both tokens must exist and be brought together. This synchronization process is called *matching*. Once these input tokens are matched, the instruction is performed, and the result tokens sent on to subsequent instructions. Note that tokens which do not require matching may go directly to the execution unit. These tokens are called *by-pass* tokens.

Viewed abstractly, a Data-Flow computation is a directed graph, which may have cycles. The nodes of the graph represent instructions, and the directed arcs carry tokens from one instruction to the next. The semantics of a node are that it may execute when all of its inputs are available, and that it executes in some finite, but non-zero,

---

[10] See [Hock88a] pages 53 to 80 for a survey of this terminology.

[11] Many modern compilation systems for parallel von Neumann systems [KSR91b] can extract parallelism from standard programming languages, however, users are frequently forced to re-structure their code in order that the compilation system can perform its duty effectively. This is often more complex, though usually less tedious, than parallelizing the code explicitly.

time. The action of a node depends only on the values of the incoming tokens and it has no side effects.[12]

The semantics of the arcs give rise the major distinctions between Data-Flow models. Since Data-Flow graphs can be cyclic,[13] more than one token may be present on a given arc. It is necessary to ensure that a token, travelling on one input arc of an instruction, pair up with the correct token on the other arc. Such correctly matched tokens have the same *context*. There are several mechanisms ensuring this, and these approaches, and their impact on Data-Flow architectures, are discussed later in this chapter.

2.1.2. Weakness of Data-Flow Computing

Many researchers have discussed the weaknesses of Data-Flow. As might be expected, some of the strongest criticism has come from the Data-Flow community itself. In the following paragraphs some of these weaknesses are summarized, and similar discussions can be found in [Arvi88a, Arvi91a, Gott91a, Gurd87a, Nikh89a].

2.1.2.1. Cost of Synchronizing Data

Data-Flow computation relies on the matching process to synchronize data values prior to computation. Data-Flow systems , therefore, execute instructions from many parts of the program in parallel, giving Data-Flow its advantage with respect to parallel computation. This represents a trade-off between the time and space domains. Data-Flow systems provide memory in which unmatched tokens wait. An associative, or pseudo-associative, memory is incorporated into Data-Flow systems which performs the match. The cost of this *matching store* has two aspects: 1) it is associative, and 2) it must hold both the data value and its tag.

Associative Memory performs the synchronizing function in a Data-Flow machine. It can be constructed in several ways, but there are two basic classes, directly associative memory and pseudo-associative memory. Directly associative memory, sometimes called content addressable memory, is more expensive than address based memories. Associative memory is commonly used in address translators, where the virtual address is used as the key when searching for the corresponding physical address.

Directly associative memory operates by performing a comparison of some of the bits in a reference register against the corresponding bits in all words of the memory simultaneously. Output from the memory is enabled only for the word that matches all reference bits.[14] The speed of an associative memory is governed by the number of bits in the comparison part of the word and the size of the memory. Since all words in the memory are read simultaneously, no single register can be used as a basis for refreshing a dynamic memory cell, and therefore, associative memories are based on static memory technology.[15] This increases their cost and decreases their size. For a general discussion of associative memory, see [Hwan89a].

---

[12] This generalization is not held universally by Data-Flow systems. Many nodes in actual systems are dependent on the general state of the system and can have side effects, e.g. the storage nodes in the Manchester Data-Flow Machine [Sarg86a].

[13] This occurs in re-entrant functions and loops.

[14] Commercially available associative memories have a more general set of functionalities [Hwan89a], but only a unique exact match of all tag bits is relevant in Data-Flow computing.

[15] They could, in theory, be built from dynamic memory, but each memory cell would require its own refresh circuit, and would not be competitive.

Because of the cost of directly associative memories, pseudo-associative memories are sometimes used to perform the matching function. Even though these memories can be built from conventional dynamic RAM, they still are complex and highly specialized. They usually employ one or more of the following techniques.

Hashing: The tag of an incoming token is sent through a hardware hashing unit which produces an address in a random access memory. This address is then read and the actual contents of the location compared to the incoming token.

Re-Hashing: If the initial hash fails to find a match and the slot is already occupied, a second hashing function can be used. This process can be repeated as long as subsequent searches are needed. The more times a re-hash is applied the slower the matching process.

Banking: In conjunction with hashing, multiple banks can be searched simultaneously at the same hash address. This avoids the need for frequent re-hashing since several locations are available at the same hash address.

Overflow: At some point, the hashing process may fail to locate an empty location in the memory. An overflow memory is then required. This is typically a slow software managed facility.

Except for rehashing, all of these techniques were used in the Manchester Data-Flow Machine [Wats82a]. The weaknesses of pseudo-associative memory include the long pipeline required for computing hashing functions, the overheads associated with unused entries in the hash table,[16] and the slow overflow.

Tag Size in Data-Flow systems causes an additional overhead in storage space. Each tag contains the equivalent of a program counter, a stack pointer, and a loop index. This overhead is carried by every data value active in the system. Techniques such as activation name recycling [Rugg87a] can allow part of the tag to be reused. This can force some sequentiality into a program. Also, for some computations, if the tag space is too small, deadlock can occur when all available tags have been used.

2.1.2.2. Communication Cost

Data-Flow systems have been criticized for their reliance on high bandwidth communication, since most tokens must travel through some type of network to reach their destinations. In order to achieve good load distribution, Data-Flow systems perform a hashing operation on the tag, determining to which processor a particular token is sent. In most cases, the design of this hashing function does not attempt to exploit spatial or temporal locality, contributing substantially to the bandwidth requirement.

The second communication problem results from the tag size. When conventional systems make a request for a data location, the request contains only the address, which is occasionally multiplexed with the data. In Data-Flow systems, the entire tag and data value are sent through the communication network.[17]

---

16 Hashing is more efficient if the hash table is sparsely filled, as this avoids clashes.

17 With modern massively parallel systems, this distinction is fading. Large address spaces and return addresses in messaging protocols are enlarging the message header in these systems.

2.1.2.3. Memory Management and Execution Control

In Data-Flow computing, memory management and execution control are closely related. Most Data-Flow systems have some form of explicit storage mechanism for data structures. This structure storage must be managed dynamically, usually with a combination of hardware and software, creating added complexity and increased access times. The amount of this memory occupied at, any given time, is a function of the execution schedule of the program. The basic greedy scheduling policy of Data-Flow creates large numbers of temporary results which occupy the memory of the system. Controlling these resource "explosions" is the responsibility of the execution control mechanism of the system. These mechanisms are complex and not fully general. This is an active research area and is discussed later in this chapter.

2.1.2.4. Sequential Bottle-necks

Because Data-Flow systems have inherently long pipelines they can perform poorly on sequential computation. Even short segments of sequential code can cause problems in what are otherwise highly parallel algorithms.[18]

2.1.2.5. Specialized Languages

The most likely reason why Data-Flow has not seen commercial success is its reliance on specialized languages. In order for compilers to extract maximal parallelism, programming languages for Data-Flow systems must exhibit "clean" semantics. These semantics are similar to those of functional languages. Data-Flow languages are covered in more detail in Chapter 3.2.

2.1.3. Anatomy of Data-Flow Architectures

In the following sections, Data-Flow architectures are analyzed with respect to a few primary design approaches. The primary distinction between Data-Flow systems is whether they are static or dynamic. The SDFA system is dynamic, and therefore most of this section pertains to the subclass of dynamic Data-Flow architectures. The features discussed are matching technology, the support provided for data structures, the strictness or leniency of functions, and scheduling techniques or "throttling".

2.1.3.1. Static versus Dynamic Data-Flow

Static and Dynamic Data-Flow execution models are distinguished by the mechanism used to handle re-entrant code, either in the form of multiple entries to a function (as in recursion) or multiple invocations of a loop body. The basic firing rule of a node, that all inputs must be present, is the same in both cases. In dynamic Data-Flow, additional information is included in the tag field of each token. This information is called the *colour*. Assignment of colours to tokens guarantees that the colour field is unique for each invocation of a loop or function body. This colour determines the *context* in which the token is being used.

Although details vary, the basic principle behind static Data-Flow is the incorporation of an additional requirement to the firing rule for a node, that the output arc(s) of the node be empty. In the MIT Static Data-Flow Architecture [Denn80a], additional arcs

---

[18] This issue is highlighted by Wim Böhm in a study of the Fast Fourier Transform algorithm [Böhm92a]. Using a sequential approach to creating the initial data array, the MIT Tagged Token Data-Flow Machine runs sequentially for a major part of what should be a highly parallel application.

are added to Data-Flow graphs which act as a feedback mechanism. When a node fires, a feedback token is sent to all nodes from which the executing node received input data. If these feedback tokens are treated as input tokens, the rule for static and dynamic Data-Flow becomes essentially the same. Two details make the static model complete. First, by the definition of their function, the arcs carrying feedback tokens can never have more than one token at a time and, therefore, do not require feedback arcs themselves. Second, the input 'slots' for feedback tokens must start with entries already present to initialize the feedback process. Successful implementations of static Data-Flow exist, including [Denn80a], and more contemporary systems have been designed [Denn91a].

2.1.3.2. Matching

Matching tokens, in preparation for execution, is at the heart of all Data-Flow systems, and the process exhibits little variation across systems. Most systems perform only binary matching, since performing multiple matches on a single tag complicates the structure of the matching store itself. Nonetheless there are systems that implement multiple matching [Abra88a].

The binary matching process takes place in an associative memory. However, more recently, Data-Flow research systems have been developed which take different approaches. The most notable is the Explicit Token Store (ETS) strategy as manifested in the Monsoon architecture [Papa88a].[19]

The construction of the matching store is critical to any Data-Flow machine, both in terms of speed and cost. It must perform the matching action, essentially an associative search, and hold tokens between their creation and the creation of their matching token. The latter can result in large storage requirements, a contradictory requirement to the former. The need for speed, and the nature of the matching process, dictate that ideally a matching store should be constructed from directly associative memory. However, this has meant too high a cost when the second requirement is taken into account. Matching stores have, therefore, usually been implemented using a pseudo-associative memory. The Manchester Data-Flow Machine's matching store, described above and reported in [daSi83a] is typical. Another approach, that of the CSIRAC machine from the Royal Melborne Institute of Technology [Abra91a],[20] incorporates multiple levels.

The matching store in the CSIRAC system is two level. Initially there is a 16K directly associative cache. If a match is not found, and control bits indicate that there might be a match in the main matching store, the main store is then searched.[21] The hash table, of the Melborne system's matching store, functions in a similar way to the Manchester machine's except that there are no parallel banks. The hash space is much larger than the expected need, and therefore, clashes are rare. However, they do occur and the

---

[19] Briefly, in ETS tokens exist within "Activation Frames", and each instruction has an explicit address within the activation frame where the first of a pair of tokens is stored. Each location has associated with it a presence bit used in determining if a match has occurred, at which time the corresponding instruction is enabled and scheduled for execution. The management of activation frames is handled by the run time system, using a mechanism based on K-Bounded loops [Arvi86a]. Each activation frame is enabled when all its input tokens are available.

[20] This was a joint project with the Commonwealth Scientific Industrial Research Organization, Division of Information Technology.

[21] There is evidence to suggest that there is temporal locality of this type in Data-Flow computation, see [Abra91a, Brob86a].

mechanism for handling them is straightforward. An entry in this table is either a token or a pointer to a linked list of entries, which is searched and appended to as needed.[22]

2.1.3.3. Data Structure Support

The management of data structures, arrays in particular, is one of the major problems in Data-Flow research.[23] Given that the semantics of Data-Flow languages are basically functional in nature, and assuming that arrays behave like a collection of scalar data values, the modification of a single element of an array necessitates the creation of another array, identical to the original, except for the altered element. Multiple references to an array require multiple copies of the array, even when only one element is needed. Solutions are varied and depend largely on the architecture, but several approaches predominate.

Data structures have two modes of reference, to the data structure as a whole and to the individual elements. The data structure is passed, as a unit, from one part of the program to another, and references to, and modifications of, data structures are performed on the elements. This dichotomy gives rise to various semantics for data structures. The following techniques are representative of Data-Flow computing in general and are drawn from the two major Data-Flow languages Id [Nikh87a] and SISAL [SISAL85].

SISAL streams are one dimensional arrays, created element by element in order. Once elements have been written they cannot be modified; they can only be removed from the head of the stream; but they may be referenced randomly by an index. They are non-strict, in the sense that elements in a stream may be referenced before they have been created. At this point, computation is suspended until the element is created.[24] Mechanisms for managing reference-before-creation semantics are discussed later in this section.

I-Structures, and non-strict SISAL arrays, are both non-strict in the same sense as SISAL streams, however, they are of a fixed size and shape.[25] Elements may be modified in both, however, this implies that the entire array is copied and the element modified in the process. There are compilation techniques to avoid excess copying of arrays [Cann88a, Sarg86a].

Strict SISAL arrays are distinct, from the non-strict variety, only in that before the pointer to an array is released the entire array must have been created. In this case, no references can occur prior to the creation of a given element, since a reference requires a pointer. This mechanism is supported in software and requires no special hardware support, but all elements of an array must synchronize before the pointer can be released. This can prove costly [Arvi88b, Egan91a].

---

22 This matching store is substantially more complex than indicated above. It is able to store queues of values, representing vectors, as well as single tokens. Details can be found in [Abra91a, Abra88a, Abra90a].

23 Data structures can, in most Data-Flow models include records and unions as well as arrays, however most Data-Flow research has been with respect to arrays. Therefore, in what follows arrays will be examined in more detail. In most cases, techniques which apply to arrays also apply to other forms of data structure.

24 These streams should not be confused with the lazy streams of functional languages, in which the creation of elements in a stream is driven by requests for them.

25 There is provision for enlarging them, but this implies making a new copy with the additional element appended.

M-Structures, part of the Id language, are the most explicit form of state in Data-Flow computation. They are initialized explicitly and then exist until destroyed. Elements of an M-Structure may be modified asynchronously in a variety of ways, however, they are implemented using techniques which avoid race conditions. M-Structures are a major deviation from the pure Data-Flow model; in particular they make programs containing them potentially non-deterministic [Bart91a].

All of the above approaches implement arrays by incorporating an explicit notion of state at some level in the computation. An array resides at a particular location and has no colour. It is stored without a context or destination; this information is provided by read instructions, which execute in context using the pointer and index as input. In this way an array may be referenced in several places without storing extra copies.[26]

As is apparent from above, data structures in Data-Flow systems require special treatment. In most systems, there is an additional, specialized function unit (e.g. the structure store in the Manchester machine) that provides the storage and performs token colouring.[27] The most important aspects of this mechanism are the use of pointers and reference counting. For each array, a pointer is created containing the location and size of the array. This pointer is then passed from one function to another, thus postponing re-colouring the entire array until it is referenced. Reference counting is used to ensure that all outstanding references to an array have completed before the storage associated with it is reclaimed.

In a system with multiple banks in the structure store, a given array can be allocated in several different ways. The entire array can be placed in a single bank; the array can be partitioned across banks in chunks; or elements can be completely interleaved. Each approach has its advantages and disadvantages. Allocating an entire array to a single bank creates a bottle-neck in the system, if the array is accessed frequently. However, distributing the array requires a central mechanism to control the allocation of memory. In the Manchester machine, both approaches are used, the former for "small" data structures,[28] and the latter, using a single element interleaving. For a detailed description of the structure management in the Manchester machine see [Sarg86a], and for a summary see [Gurd92a].

2.1.3.4. Strictness Class

By controlling the semantics of function calling mechanisms, through various levels of strictness, the degree of parallelism exposed to the architecture can be regulated. In sequential computing, all functions (and procedures) are fully strict by virtue of the sequential semantics of the underlying model. All arguments to the function (and the entire state of the program by default) must exist before the function can execute. By relaxing this constraint, parallelism can be exposed. In Data-Flow computing, there are several classes of strictness commonly adopted.

Functions, in a pure Data-Flow model, exhibit lenient semantics. A token arriving at a lenient function may cause the execution of instructions within the function, and the function may produce results, before all the other arguments have arrived. Fully strict

---

26 Copies of this array are, nonetheless, created as a result of read operations.

27 Data structures are usually stored as un-coloured tokens. They are coloured, and routed to the appropriate instruction, as part of a "split-phase" operation, half in the processor and half in the structure store.

28 "Small" is an adjustable architectural parameter, which cannot be altered dynamically, but is approximately 10.

functions must have all arguments before they can execute. The strictness class of a Data-Flow model is determined largely by the mechanisms used for function call management. A theoretical discussion of strictness and parallelism can be found in [Wdal88a].

In most Data-Flow models, all tokens entering a function must be re-coloured to distinguish them from other invocations of the same function. In the Manchester Data-Flow Machine, an instruction is executed which generates a special token, the trigger, whose value is a unique colour. The trigger token is used to re-colour all arguments to the function. All functions are therefore strict in this, invisible, argument. Within the Manchester system, there is also provision for functions strict in all their arguments [Sarg85a].

Functions in the Id system [Nikh87a] are managed similarly. Arguments are coloured based on the execution of specialized instructions. The colour is obtained by a mechanism known as K-Bounded Loops [Arvi86a, Cull89a]. In this approach, each function body is allocated K colours, where K is a number determined by the system software and user directives. As arguments arrive at a function boundary, they are coloured with these K values until there are K active invocations of the function. At this point, arguments are blocked until all the results of an earlier invocation have been produced, synchronized, and the old K value released. This K value is then re-used to colour another invocation of the function. This technique was used on the MIT Tagged Token Data-Flow Machine [Arvi86a] and the Monsoon system [Papa88a].

The above represent three common levels of strictness:

Lenient - Any argument to a function may begin execution of the function dependent only on the data dependencies within the function itself.

Limited - The function is strict in some of its arguments, usually the trigger is used to re-colour other arguments to the function.

Strict - All arguments to the function must be present before execution of the function may begin.

In practice, only systems with limited strictness have been successful. Strict systems exhibit limited parallelism, an issue well illustrated in [Cull87a]. Lenient systems, like the one proposed later, are difficult to control, since function invocation provides a convenient handle with which to limit the parallelism in a system.

2.1.3.5. Throttling

Throttling is the term used by the Data-Flow community to refer to the process of controlling and scheduling the execution of Data-Flow programs. Data-Flow programs expose large amounts of parallelism, and Data-Flow systems implement a greedy scheduling policy. Therefore, they can create large numbers of temporary results, which can cause a "resource explosion". The aim of throttling is to reduce resource usage without reducing performance. There are two basic approaches.

The Manchester approach relies on the fact that function and loop bodies are strict with respect to their trigger token. The trigger token is the result of a split-phase operation which produces a new colour, called an Activation Name (AN). The AN is generated by a special function unit, called the Throttle. There is only one throttle per system and it maintains a data structure of all ANs, which includes their ancestor relationships in the call tree. Since each AN represents a body of computation, which may include requests for additional ANs, the decision to release an AN, or suspend its release, can have a direct effect on system load. To provide the throttle with information on which to base this decision, the total length of the Token Queues in the system is used as the dependent variable in a simple "bang-bang" control system. Briefly, the control function attempts to predict when the workload on the system will need to be increased in order to keep the pipelines full. This technique has proved extremely effective at both reducing store use and maintaining highly efficient execution. A detailed report of the most recent work can be found in [Gurd92a] and earlier work in [Gurd87a, Rugg87a, Teo89a, Teo91a]. A similar approach was adopted for the SIGMA-1 system [Hira91a] and is currently being studied with respect to the CSIRAC II system [Egan91a].

The other major approach is called K-Bounded loops [Arvi86a], described earlier. The MIT Tagged Token Data-Flow Machine [Arvi89a] and the Monsoon [Papa88a] control parallelism using this technique.

There are several other techniques which are less common, but have been attempted at one time or another.

- Particular functions can be coded sequentially. This reduces the parallelism and the store requirements. It is, however, manual and static.[29]

- Particular functions can be coded strictly. This has an effect similar to the above, with the same disadvantages.

- Code for either of the above can be planted along side parallel code. The system can then choose, at run time based on some system load indicator, which version of the code to execute.

- At a fine grain level, priority can be given either to matched pairs of tokens or by-pass tokens. The assumption here is that by-pass tokens can only increase the system load, whereas matched tokens can, potentially, reduce it. The problem is that matching and by-passing tokens exist uniformly throughout the code, sections of which cause an increase in load where others cause a decrease.

- In the token queue of a system, the usual technique, of inserting executable instructions into a FIFO queue, can be replaced by a stacking technique. The assumption is that recently created tokens will be consumed first, thus reducing system load. The problem is that not all codes behave in this way.

- The token queue and the token stack can be combined in a dynamic mechanism that attempts to choose between the two, intelligently. This technique failed to prove robust enough for a variety of applications.

- A forerunner to the Activation Name approach, currently used in the Manchester machine, was the "T-leaf" approach. Based on ANs, the system attempts to maintain a fixed number of active "leaf" processes. A leaf process is associated with an AN at a leaf of the AN tree. This approach was superseded by the current approach.

Surveys of these, and other techniques, and research involving the regulation of parallelism can be found in [Böhm90a, Fole89a, Rugg87a, Teo91a, Arvi86a, Egan91a].

---

[29] The disadvantage of a static scheme is that it cannot respond to changes in system load.

2.1.4. Related Work

The current trend in Data-Flow research is to incorporate increasingly more explicit notions of state into the architecture. In the process, many of the Data-Flow ideals are being left behind and are being replaced by abstractions that are more conventional. At the same time, conventional computing has been incorporating many of the techniques of Data-Flow, particularly through efforts to improve the latency hiding aspects of modern multi-threaded systems. Below are some developments in Data-Flow and related research which highlight this trend.

2.1.4.1. Data-Flow

- The two primary fine grain Data-Flow systems are the Manchester Data-Flow Machine [Gurd85a, Gurd87b] and the MIT Tagged Token Data-Flow Machine [Arvi89a]. Many of the features of these machines are discussed above. The most recent work on the Manchester system is focused on throttling and resource management [Snel92a, Gurd92a]. The current work at MIT has shifted to the Monsoon project, see below.

- The CSIRAC II is a tagged token Data-Flow machine with an additional feature in the architecture, not present in the Manchester and MIT machines, which forces tokens, sharing an arc, to remain in order. This reduces the extent to which tagging and matching are required and increases the power of the matching functions. In particular, direct, matching unit support is provided for most stream functions and for scalar expansion [Egan91a].

- Derived from the MIT Data-Flow machine and running the same Id software, the Monsoon is based on the Explicit Token Store model mentioned above. In this approach, large grains of computation, called Activation Frames, are scheduled to run when all the inputs to their slots have been filled. Several of these frames are allowed to execute concurrently [Papa88a].

- Other Data-Flow projects include the DTN Data-Flow Computer from the Netherlands [Veen86a], the Sigma-1 [Yuba84a, Hira91a] and the EM-4 projects from Japan [Yuba90a], the Datarol project also from Japan [Amam91a], and the Epislon project from Sandia National Laboratories [Graf91a]

2.1.4.2. Multi-Threaded Architectures

- The Denelcor HEP [Smit78a, Dene81a] is a machine with a close affinity to Data-Flow and was the first of the multi-threaded architectures. The system maintains a collection of 128 active processes (each with their own program counter) and hides latency by removing processes, which have outstanding memory accesses, from the processor pipeline. More recently, Tera Computers has begun development of an architecture based on similar principles [Smit90a]. The locations in memory of both systems contain an extra presence bit. This bit provides support for data driven computation.

- The P-RISC [Nikh89a] architecture is the first major step, by the MIT Data-Flow group, into the field of multi-threading. It has no matching store at all, has a RISC like ALU and enhancements to support multi-threading. In particular, the instruction-set includes fast fork and join instructions that allow the processor to switch from one thread to another in less time than a remote memory access. Remote memory accesses also incorporate a split phase form of read and write, which further aids in latency hiding. The execution mechanism is very much like that of the Denelcor HEP, except that the context switch occurs under instruction control.

- A direct follow on from the P-RISC project, the *T architecture [Nikh92a] supports multi-threading by software controlled context switches. The added features in *T are multiple processors for performing the functions involved in supporting multi-threading (computation, remote request servicing, and synchronization of returning requests) and queueing between these processors.[30]

- The ACRI [ACRI92a] architecture supports multi-threading by partitioning the instruction-set into independent instruction classes - control, memory, and floating point. Three separate processors run independent threads of instructions. As in *T the processors communicate by queues and parallelism is maintained as long as the processors remain de-coupled.

- Other multi-threaded systems include Alewife [Agar90a] and DASH [Leno90a].

2.1.5. Summary

The primary benefit of Data-Flow is its ability to exploit parallelism transparently. This parallelism, because it is fine-grained and abundant, can be used to hide latency and improve performance through parallel execution. However, this fine-grained abundance creates problems in resource management, and its transparent nature implies the use of specialized languages, a major factor inhibiting commercial exploitation.

In the next section, the notion of explicit state in Data-Flow systems is analyzed and the SDFA system introduced.

2.2. The SDFA Approach

It is clear that latency hiding and the exploitation of memory hierarchies are essential to efficient execution of parallel programs, particularly in scalable systems. These two issues are, therefore, central to the design strategy of the Stateless Data-Flow Architecture. These are incorporated into a framework which allows existing and novel scheduling techniques to be implemented.

The SDFA architecture has no explicit storage for data structures, arrays in particular. Arrays are not stored in a logically central memory, but migrate around the memories of the system dynamically. This distributed memory model provides a basis for developing techniques which improve the performance and scalability of Data-Flow systems.

---

30 These processors can be viewed as virtual and multiplexed onto a single physical processor.

Synchronization of data (matching) in SDFA is also distributed, thus substantially reducing the impact that a limited top level memory has on the ability of the system to hide latency. Unlike other Data-Flow systems, matching is distributed throughout a multi-level memory hierarchy. By using a fine grain Data-Flow model, both calculation and synchronization can be dispersed more widely through the system than is possible in systems based on coarser models, such as thread based systems [Smit90a, Nikh92a, Cull91a].

SDFA provides an environment in which scheduling can be studied at three different levels. The fine grain, greedy, Data-Flow scheduling lies at the bottom of this hierarchy. Above this, mechanisms exist for enforcing a partial order on the computation based on a static interpretation of a "likely" computation order. At the highest level, both static and dynamic schemes can be implemented that schedule coarse grain segments of the computation.

2.2.1. The Question of State

In this section the issue of explicit state is addressed directly. Initially some historical background is provided. This is followed by a discussion of the inherent barrier in conventional Data-Flow, present as a direct result of explicit state, which prevents scalable implementations. Last, a proposal for the Stateless Data-Flow Architecture, which is freed from this barrier, is presented.

2.2.2. The History of Explicit State

Data-Flow machines have traditionally been built, or designed, with some notion of explicit state. The early Manchester machine had an explicit notion of state, even before a structure store was incorporated. At this time, data structures were stored in the matching store, at what were called "storage nodes" [Sarg86a]. An array of values, distinguished by their index fields, formed a collection of tokens on one of the arcs of the storage node. Approximately 7 to 10 instructions were required to obtain a copy of one of the elements of an array. This, by itself, was sufficient motivation to build the structure stores, although several other reasons were cited, including the high cost of large matching stores. In the MIT Machine, I-Structures, which incorporate a notion of state explicitly, were part of the original design methodology [Arvi80a]. Later Data-Flow systems followed the lead set by these two systems.

The reasons for including state explicitly can be outlined easily:

- The cost of directly associative memory is much higher than standard RAM, and alternatives, such as pseudo-associative memory, are complex and, hence, not competitive.

- The performance penalty paid for re-colouring entire arrays on entry to a function, led designers to look for a pointer based mechanism for colouring data structures.

In recent years, both of these factors have shifted in favour of a stateless approach. The development of VLSI techniques have meant that the creation of large, custom associative memories is within the reach of technology [KSR91a]. These large custom memories are still more costly than dynamic RAM, but the situation has improved.

Even with data structures stored in explicit memory, copying is required in most Data-Flow programs. Modern compiler techniques [Cann88a, Cann89a], aimed at reducing the number of copies, should be applicable to a stateless approach as well.

There is a price to pay for explicit state in Data-Flow systems. Dynamic memory management is necessary at both hardware and software levels. Without hardware support, the time penalty would be prohibitive. In any case, the software must request data structure allocation, maintain reference counts, and initiate garbage collection. The Manchester machine uses a hardware supported garbage collection scheme, which can be disabled during compilation. Comparing both approaches provides an indication of the cost of dynamic memory management. A 1K element quick sort requires 2370 allocations. The number of instructions increases by 3% when garbage collection is used. However, most of the effort of garbage collection is in the hardware, where there is a 55% increase in the number of memory accesses. Garbage collection is necessary, given that, with garbage collection, the maximum memory used is 10K locations. This rises to over 300K locations without garbage collection. Improved compilation techniques will continue to reduce the software cost of garbage collection, but the hardware must continue to provide specialized support. There is no dynamic memory management required in a stateless approach.

However, the primary justification for a stateless model is the fundamental barrier that explicit state imposes on the architecture's ability to scale.

2.2.3. Fundamental Barrier in State-Based Data-Flow

The most important conclusion, from the summary of David Culler's recent work (see Section 1.1.2.), is that a system without a memory hierarchy cannot be scaled indefinitely without a drop in performance or a substantial increase in cost. The scarcity, and high cost, of shared memory multiprocessors is a testimony to this fact. With the exception of the the Tera machine [Smit90a], which is not yet realized, the largest number of processors connected to a shared memory is less than 50. Systems, with many more processors, have been constructed from simple processors, arranged in a distributed memory configuration. Distributed memory systems exhibit a multiple level memory hierarchy, usually manifested as non-uniform memory access times, and are programmed using a distributed memory programming model.[31] To obtain high performance on these systems, both temporal and spatial locality must be present to exploit the memory hierarchy. Temporal locality ensures high performance on the individual RISC style processors; spatial locality reduces the inter-processor communication traffic. Therefore, high performance (within a reasonable cost) requires a multiple level memory hierarchy, probably in the form of a distributed memory architecture.

Data-Flow systems do not exist, at present, that exhibit a distributed memory architecture. There are systems built where the memory is packaged with the processors [Papa88a, Abra88a], but data structures are stored as global objects, distributed across the memories of the system. Therefore, all accesses to arrays appear as if they are made to a separate, globally shared memory. If these systems are scaled, there will come a point at which the access latency, to these shared data structures, can no longer be hidden, and performance will begin to degrade.

The solution is to ensure, by some mechanism, that the data is almost always on the processor where the computation is to be performed. This is not possible in existing Data-Flow paradigms, which support an explicit notion of state for storing data structures.

In what follows, a distributed memory configuration is assumed, with the main structure memory distributed around the processors.

---

[31] Recently these systems have visualized their address spaces, creating a shared memory appearance, but high performance is only available through careful placement of data and computation. To this end, language extensions have been proposed such as High Performance Fortran [Love93b].

The key point here is that data structures are stored without colour. They are accessed by pointer and coloured by the memory unit, before the value is sent to the processor for use in a computation. The colourless status of data structures means that during a computation, requiring a particular data structure element, the computation (which is always coloured) will not be the same colour as the data. Since colour is usually a fundamental part of the distribution function, it is highly probable that the computation is on a different processor. Therefore, the state based Data-Flow model is inherently a shared memory model.[32]

However, by removing the explicit notion of state, data structures and their associated computation are always in the same context (are the same colour) and are, therefore, in the same processor. Since, in order for two tokens to match they must have the same tag, data created in one processor and re-tagged (e.g. as part of an index shift) migrates to the processor on which it was required. The communication, implied by this migration, is that associated only with the structure of the algorithm and not as the result access to global data. This is true regardless of distribution functions or throttle mechanisms employed by the architecture.

It is argued here that, by removing the notion of state, latency hiding, load balancing, and locality can be achieved. If successful, the stateless model is better suited for constructing massively parallel systems than traditional Data-Flow models. It should be noted that massively parallel systems could possibly be built from traditional Data-Flow processors, but the communications network would require greater bandwidth and longer pipelines. This would increase the minimum problem size that could be run effectively on the system. Likewise, the amount of high performance local state required, on the processor, to mask the latency of access and synchronization, would increase.

2.2.4. Proposal for a Stateless Data-Flow Architecture

In the remainder of this thesis a Stateless Data-Flow Architecture is developed and analyzed. This architecture is capable of exploiting the high degrees of parallelism available in Data-Flow programs; exploiting the locality (spatial and temporal), thus providing the basis of a scalable architecture; and using excess parallelism to mask latency in access, synchronization, and pipelining. There are, however, several details that are beyond the scope of this thesis. These are given below accompanied by a brief justification of each assumption. This is followed by a statement of the goals of this study.

2.2.4.1. Assumptions

Directly associative memory, for high performance sections of the matching store and processing element, can be designed cost effectively. Directly associative memory is based on static memory circuits, where each memory cell requires several more gates than a random access static cell [Hwan89a]. This is a constant factor, and therefore, associative memories, of comparable size to fast static RAMs, should be constructable. Speed might present more of a barrier, since look-up time is proportional to the logarithm of the tag length, rather than the size of the address. Pipelined designs could

---

32 It might be argued that, if the context is not used in the distribution function, computation could be performed on local data. This is perfectly true. However, the explicit distribution and alignment of data and computation are then either the responsibility of the compiler or the programmer. In either case, data structures must be explicitly located on particular processors, thus destroying the dynamic nature of Data-Flow computation. If compilation techniques fail to produce adequate results, HPF [Love93b] style directives, for distribution and alignment, would be required in Data-Flow languages like SISAL and Id. The generality and elegance of these languages would be lost.

then be considered. Large and fast associative memories exist in systems currently available, where they are used for cache management [KSR91a].

In addition to the associative memory, modern processor complexity and performance have been assumed. The technology assumed is similar to the Alpha [DEC92a] processor from DEC.

A compiler for this architecture does not exist, and the eventual development of an efficient code generator is assumed. Since the base language is SISAL, a language with transparent semantics, code generation should be fairly straight forward. Compiler writers working with SISAL, and other functional style languages, have found them a relatively simple basis for code generation and optimization [Cann88a].

2.2.4.2. Goals

The overall goal of this study is to justify a distributed memory Data-Flow computer, based on a stateless model of computation. To accomplish this, the architecture must exhibit the following properties:

- It must reduce latency through the use of locality to exploit a memory hierarchy.
- It must hide latency due to synchronization, communication, and pipelining.
- It must be possible, in the presence of adequate parallelism, to distribute the workload uniformly throughout the system.
- It must be possible to regulate the computation so as to minimize resource consumption without degrading performance.

2.3. Summary

It has been argued that scalable performance is only possible in a distributed memory environment, and that all current Data-Flow implementations support a colourless structure memory which results in a shared memory model. By abandoning explicit state, it is proposed that scalable performance can be achieved without a major sacrifice in performance. The remainder of this thesis describes and analyzes a Stateless Data-Flow Architecture.

Chapter 3
A Stateless Data-Flow Architecture

3.1. Summary of the Model

This chapter discusses the execution model of the Stateless Data-Flow Architecture (SDFA) and a physical architecture based on modern hardware technology.

SDFA is a dynamic, tagged token Data-Flow architecture. It is fine grain, both in terms of the level of parallelism and the structure of the instruction-set. All the instructions in the instruction-set are simple and based RISC principles. There are no "iterative" or vector style instructions producing more than two tokens per execution. Operations controlling the flow of data through a program and operations on the data are separate within the instruction-set. In particular, there are no dynamic arcs,[33] and operations on the tag fields are independent of operations on the data field. Thus, there is the possibility of enhancing the SDFA model to incorporate a more complex instruction-set based on a horizontal instruction format.

As its name implies, the SDFA system has no explicit notion of state. There are no structure stores, and only *extract-wait* functionality is provided in the matching store.[34] As a result, all computation is functional and deterministic.[35]

The SDFA model provides computational support for basic Data-Flow graph evaluation and procedure invocation. Higher order functions could also be supported where they can be in-lined at compile time.[36]

3.2. The SDFA Execution Model

In this section, the token format of SDFA and the two primary mechanisms required by any Data-Flow system are described, namely the management of functional abstraction and data abstraction. The SDFA model has explicit support for functional recursion (linear and multiple) and blocked, multi-dimensional arrays.

---

33 Dynamic arcs are used in many Data-Flow systems to support re-entrant code. A function entered from multiple locations in the code carries, in addition to all its input data tokens, a token containing the "return address" for the current function invocation. For each result of the function a routing instruction is executed which combines the result value with the return address and sends the result to the proper place in the code. For an example of this approach see [Gurd85a].

34 More complex machines support other matching functions such as *decrement-defer* which will decrement the value of the token waiting in the matching store or, if there is no matching token already in the matching store, "defer" and try again on its return visit [Gurd87a] .

35 Non-deterministic computation is beyond the capability of this model and thus beyond the scope of this thesis. However, Chapter 8.3.7. describes how a non-deterministic mechanism could be added to the SDFA system.

36 This approach is also taken within Id [Nikh87a].

3.2.1. The Token Format

All data values in SDFA are carried in the form of tokens. Each token consists of a data field, a tag field, a matching flag, and a port indicator. There are sub-fields in the tag for the destination and the context of the associated data. The general format of a token is given below.

Token ::

Field	Size	Description
Data	64 Bits	Real, integer, or boolean data value.
Match	1 Bit	True indicates that token needs to match.
Port	1 Bit	Input port at destination instruction.
Tag		
ProcessTag	Fixed length	Explicitly controllable tag field.
TagType	Fixed length	Indicates format of remaining fields
Destination	Variable length	Destination instruction address.
RecTag	Variable length	Indicates depth of linear recursion.
RecTagStack	Variable length	Stack of all previous RecTags.
IndexX	Variable length	Index in 1st dimension.
IndexY	Variable length	Index in 2nd dimension.
IndexZ	Variable length	Index in 3rd dimension.
IndexT	Variable length	Index in 4th dimension.
IndexS	Variable length	Scratch index for reductions etc.

The Data field is any value which can be represented in 64 bits. For example, in all that follows, this field contains real, integer, or boolean data. The Match and Port fields are single bits indicating whether the token requires a matching token and if it is the left or right operand to an instruction.

The ProcessTag is a fixed length field which must be set explicitly. This could be part of the user's code or a run-time system function. The ProcessTag can serve many explicit control functions such as throttling, processor network partitioning, locality and distribution of data in the network, protection domains, activation names, system service calls, etc.

The TagType is a number indicating the format of the remaining fields. These formats are user definable and allow different programs to use the tag space differently. For example, a program which used only very large, one-dimensional arrays might require many bits in the IndexX field and none in the higher dimensional index fields.

The Destination field is the address, possibly relative to a protection domain, of the instruction for which this token is intended.

The RecTag field distinguishes the context of simple recursive function calls. The RecTagStack manages the history of initial calls to recursive functions, and all multiply recursive function calls. These mechanisms are discussed in more detail later.

The Index fields provide direct support for 1, 2, 3, and 4 dimensional arrays and reduction across any dimension, without requiring the use of techniques such as "context sensitive" loops.[37] These fields are discussed in detail later.

---

[37] "Context sensitive" is the term used by the Manchester Data-Flow project for loop bodies containing (non-inlined) function calls or other loops. In these cases, the use of the index field of the tag would be insufficient.

3.2.2. *Management of Function Calls*

In any programming model, functional abstraction is essential to express programs efficiently. At the machine code level, the implementation of functional abstraction can provide spatial and temporal locality.

The two simplest forms of functional abstraction are encapsulation, enclosing a section of code into a function, and recursion, which expresses a computational structure in addition to encapsulating code. However, the most common forms of functional abstraction are formulated as loops, either as iterative instantiations of the same code block or as a means of applying a sequence of operations to an entire array. In SDFA, iteration is formulated recursively, and array operations are accomplished using the index fields of the tag.

The implementation of non-recursive functions is accomplished using code replication for each textual occurrence of the function. This approach is equivalent to the, now common and automated, practice of code inlining, and could, in some cases, cause an increase in the amount of code generated. It is expected that there would not be an excessive growth in code size, and techniques exist to minimize it.[38]

The implementation of recursive functions is accomplished using two parts of the tag field, the RecTag and the RecTagStack. The destination address for the results of a recursive function can be decided based on the value of the RecTag. The RecTag is a counter indicating the depth of recursion. When a return encounters a zero RecTag, the result token is sent to the calling function, otherwise the token is sent to the point where the function was called recursively. In tail recursive functions (linear recursive functions which return only once), the destination for all results is fixed at the point where the function was initially called. Appendix A.1.3 contains a rigourous development of these tagging strategies and the tagging strategy for multiple recursion.[39]

3.2.3. *Management of Arrays*

Arrays are collections of data values, travelling along the same arc of a Data-Flow graph, distinguished by the index fields of their tags. There are five index fields within the tag: X, Y, Z, T, & S. The S field is a scratch field for use in reductions and is not used as an array dimension. Higher dimensions are supported by macros, at the language level, which convert the higher dimensions into longer versions of other dimensions.[40]

An array travels through a SDFA Data-Flow graph on a collection of arcs, one for the elements of the array and one for each dimension of the array. The dimension arcs each

---

38 The simplest approach is to tag the separate calls uniquely using the ProcessTag field. Alternatively, the function could be coded strictly, in the sense that no inputs to one instance could enter the function prior to the previous instance completing. Since inlining would be worst at the outer most level of a program, either approach would not present a significant programming inconvenience and could be accomplished explicitly with the use of compiler pragmas. Likewise, the pragma could direct the compiler to treat a particular function as if it were called recursively.

39 Note that the SDFA model has no direct support for mutually recursive functions. However, by encapsulating all functions involved in a mutually recursive program structure into a single, multiply recursive function, mutual recursion can be supported indirectly. This, like mutual recursion, is not fully general since the number of internal points of invocation is an implementation dependent constant (8 for the current simulator).

40 This is in effect what all conventional systems do, since memory is normally a one dimensional address space. The SDFA model supports several dimensions in the addressing domain. This will reduce the amount of address calculation required in multi-dimensional computations.

carry a single token containing an Index Set Descriptor (ISD), which describes the, possibly blocked, structure of the array in a given dimension. The ISD is discussed in more detail in Appendix A.1.4.3.

3.3. Instruction-set

The instruction-set for SDFA is based on a classical two-input, two-output Data-Flow model. Each instruction encodes an operation to be performed, the nature of the two inputs (tokens, literal, or null), and the two possible destination addresses. The basic philosophy of the instruction-set is twofold; it should be simple and orthogonal. The goal of simplicity (motivated by the desire for speed of computation, ease of compilation, and cost of hardware) is addressed by adhering to a short list of requirements:

Each instruction:

a) has 1 or 2 inputs;
b) must have at least one input which is a token;
c) may have one input which is a literal;
d) produces at most one result value;
e) produces results with exactly one tag value;
f) has 0, 1, or 2 defined destinations.

The goal of maintaining orthogonal aspects to the instruction-set is addressed by organizing the instruction-set into sections which, with the exception of type transfer operations (e.g. Float), are independent of each other. This is motivated by the possibility of exploiting a more complex instruction-set (without substantial redesign) and the ability to exploit specialized function units without replication of hardware. The complete instruction-set and instruction format are provided in Appendix A.1.4.

3.4. Technological Assumptions

The following technological assumptions are the basis of the development of a physical architecture supporting the SDFA model.

- Microprocessor technology is capable of supporting multiple function units, multiple registers (64 to 128, 64 bit registers), and large on chip associative cache (approximately 16 KBytes) [Inmo91a, KSR91a, DEC92a, Bako90a].

- All components of the processor architecture (processor, token queue, matching store banks, and other smaller components such as interconnect chips) can be packaged on individual chips.

- A chip package will allow enough pins to support three full tokens plus two additional data values in parallel, power, external control, and debug.

- One cycle is required to transfer a token from one component of a single processor to another on the same processor. A pipelined approach can also be considered, where latency could be several cycles, but bandwidth would remain the same.

- There exist fast communications networks which may be point-to-point or packet switched, serial or parallel, pipelined or non-pipelined.

- It is possible to build large, relatively fast, direct, associative memories using custom techniques.

- It is possible to build very large, pseudo associative memories from DRAM, using available parts.

- Single module processor designs allow very large, fast memories to be built into the same package as the rest of the function unit hardware. Much of this memory is associative.

- The cycle time is assumed to be approximately 6 nanoseconds, in line with what is currently available [DEC92a]. The design of the various components of the system will use this figure as a guide to deciding how long pipelines should be and how many cycles are required to perform non-pipelined activities.

3.5. Hardware Architecture

The hardware architecture for the SDFA model is a multiprocessor machine connected by a switch network. The simulation system supports an all-to-all packet switch. All processors are homogeneous and contain a token queue, a processing element, a matching store, and a network interface.

In the remainder of this section, the structure of the architecture is described in terms of three simple abstract components: Queues, Pipes, and Arbitrators. The timing characteristics of the all components of the system, including memories, are modelled by these primitives. In each section, the performance parameters of some pipes will be justified by argument and reference; others will be studied in Chapter 6. Likewise, some queue lengths will be discussed here while others are studied later. Appendix A.3. is a listing of the architecture configuration file for the final architecture, based on the results of experiments carried out and discussed in Chapters 5, 6, and 7.

The overall structure of the machine is shown in Figure 3.1.

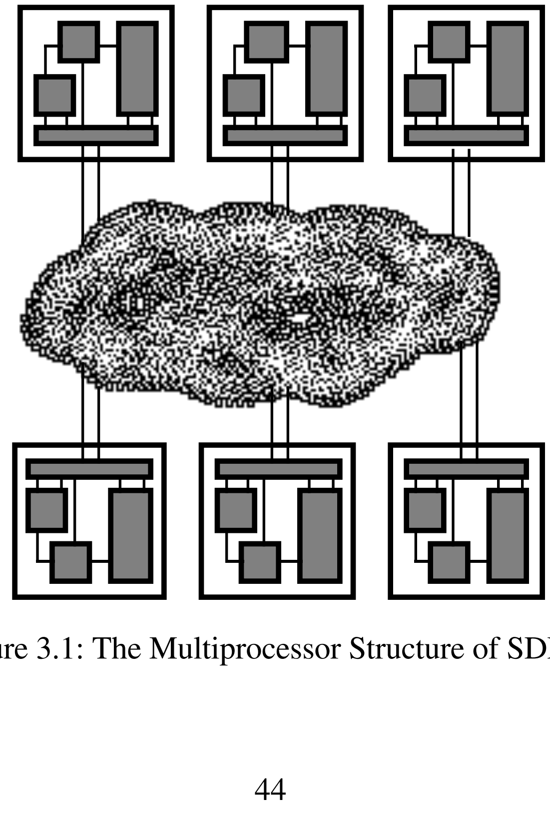


Figure 3.1: The Multiprocessor Structure of SDFA.

3.5.1. Queues, Pipes, and Arbitrators

A *queue* is an abstraction of a FIFO queuing device. Each queue is described by two parameters: its *length* and its *width*. The length of a queue is maximum number of tokens (or token pairs[41]) the queue can hold. The width is the number of tokens (or token pairs) that can be inserted into, or removed from, the queue in a single cycle. This is normally one, but in some cases it is convenient to allow more than one token to follow a given path in a single cycle. A queue, wider than one token, is simply two parallel queues with a simple arbitrator at each end. This generalization simplifies the design process and raises the level of abstraction.

The timing characteristics of a queue are as follows. If a queue is non-empty, tokens can be inserted and removed, according to the width and length of that queue, during each cycle. However, no token can enter and leave a queue during a single cycle. The design of a queue of this nature is straightforward, as long as the length is relatively small [ACRI92a].

A *pipe* is an abstraction of a pipelined function unit. Each pipe is described by two parameters: its *length* and its *beat*. The length of a pipe is the number of "slots" in the hardware pipeline that it models. The beat is the number of cycles between advances of that pipeline. For example, the floating point adder/multiplier advances each cycle and is several slots long, whereas the divider is not pipelined and is therefore only one slot long but must wait several cycles before advancing. A pipe can accept only one token per cycle and then only when the first slot is empty. Thus pipes may "stall" whenever the last slot cannot be emptied, even if there are empty slots inside the pipe.

An *arbitrator* is an abstraction which models the arbitration and distribution functions in SDFA. An arbitrator has no storage associated with it and is transparent in time.[42] All arbitrators are either *round-robin-fair* or *biased*, both of which are simple to construct. *Round-robin-fair* means that the inputs are arranged in some fixed order. At each cycle, priority is given to the input following the input that most recently received a token. *Biased* here means that the inputs are again arranged in some fixed order. At each cycle, priority is given to the first input which can accept a token.

Figure 3.2 describes the notation, for queues, pipes, and arbitrators, used in the diagrams that follow.

---

[41] At various points in the processor, e.g. between the token queue and the processing element, queues and pipes must be able to carry either a single token or a pair of tokens with the same tag. In the remainder of this thesis, "token" may occasionally refer to a pair of matched tokens.

[42] In this design, the time associated with arbitration has been incorporated into the adjacent queues and pipes.

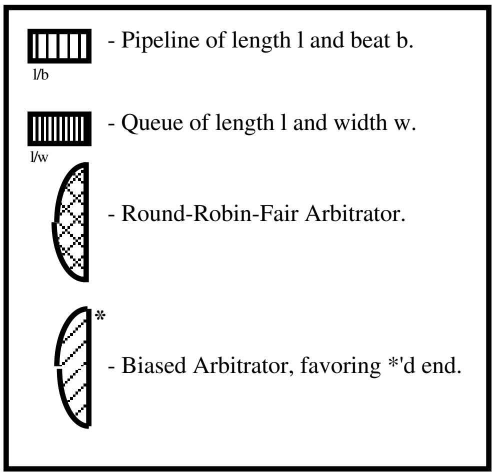


Figure 3.2: Queues, Pipes, and Arbitrators.

3.5.2. Network

The SDFA processor architecture is independent of SDFA network architecture. The only assumptions are that each processor can send and receive a number of tokens simultaneously and that these tokens may take several cycles to transmit. Several network architectures can be supported by the simulation environment. The network architecture studied here is an all-to-all packet switch. Other types of network, which could be modelled, are k-ary n-cubes [Ni93a], hyper-cubes [Seit85a], Fat-Trees [Leis85a], and networks like that of the Tera architecture [Smit90a], which is a torus sparse in two of three dimensions. A discussion of the packet switch follows.

In the packet switch configuration, each processor has one input and one output port, with buffering at each end. Figure 3.2 depicts a simplified abstraction of a packet switch, which is adequate for the size of systems simulated later. A discussion of this simplification can be found in Appendix A.2.1.

The performance of the packet switch assumes that it is possible to route a single token through a 2x2 switch in a single cycle. This is a very high level of performance and allows other aspects of multiprocessing to be studied independently of network performance. The sensitivity of the network to bandwidth and latency is studied in Chapter 7.

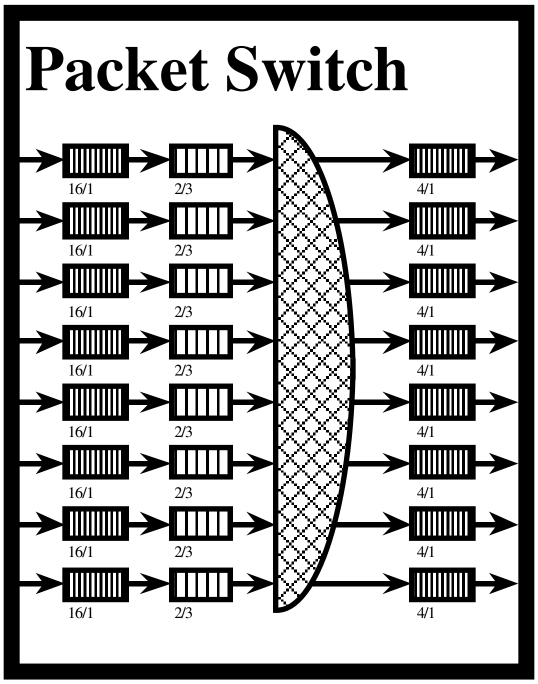


Figure 3.3: Abstract Model of a Packet Switch.

3.5.3. Processor

Figure 3.4 shows the relationships between the components of a SDFA processor. Each of the data paths is one token wide except for the data paths between the matching store and the token queue, and between the token queue and the processing element, which are a token pair wide.

Token arbitration is biased in favour of tokens originating from the local processing element followed by those from the inbound port.

3.5.4. Token Queue:

The function of the token queue is to hold tokens and token pairs, which are ready to execute, until the processing element can accept them. Tokens arriving in the token queue may be inserted into the token store according to an ordering function. This function may be a simple first-in-first-out queue, a last-in-first out stack, or a more complex function. The latter facility can be viewed as support for a throttle. These throttling techniques are investigated in Chapter 6. Figure 3.5 shows the structure of the token queue.

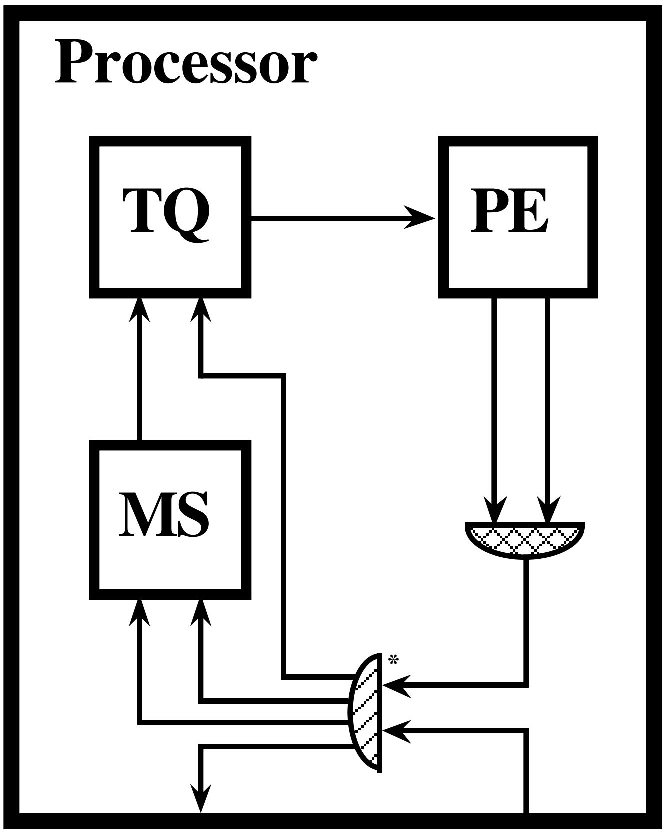


Figure 3.4: A Single SDFA Processor.

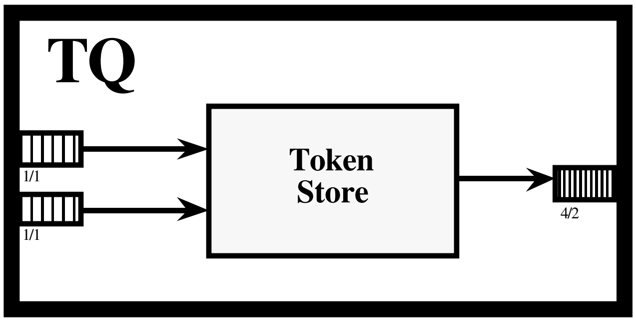


Figure 3.5: The Token Queue.

3.5.5. Processing Element:

The processing element is standard RISC style processor, where the registers have been replaced by queues and data cache by the Short Stay Matching Store (SSMS). The components of the processing element are the Instruction Fetch Unit, the Call-Return Function Unit, the Divide-Square-root Function Unit, the Floating Point Unit, two Integer Function Units, the Local By-pass Queue, the SSMS, and the Output Dispatcher. Figure 3.6 shows the overall structure of the processing element.

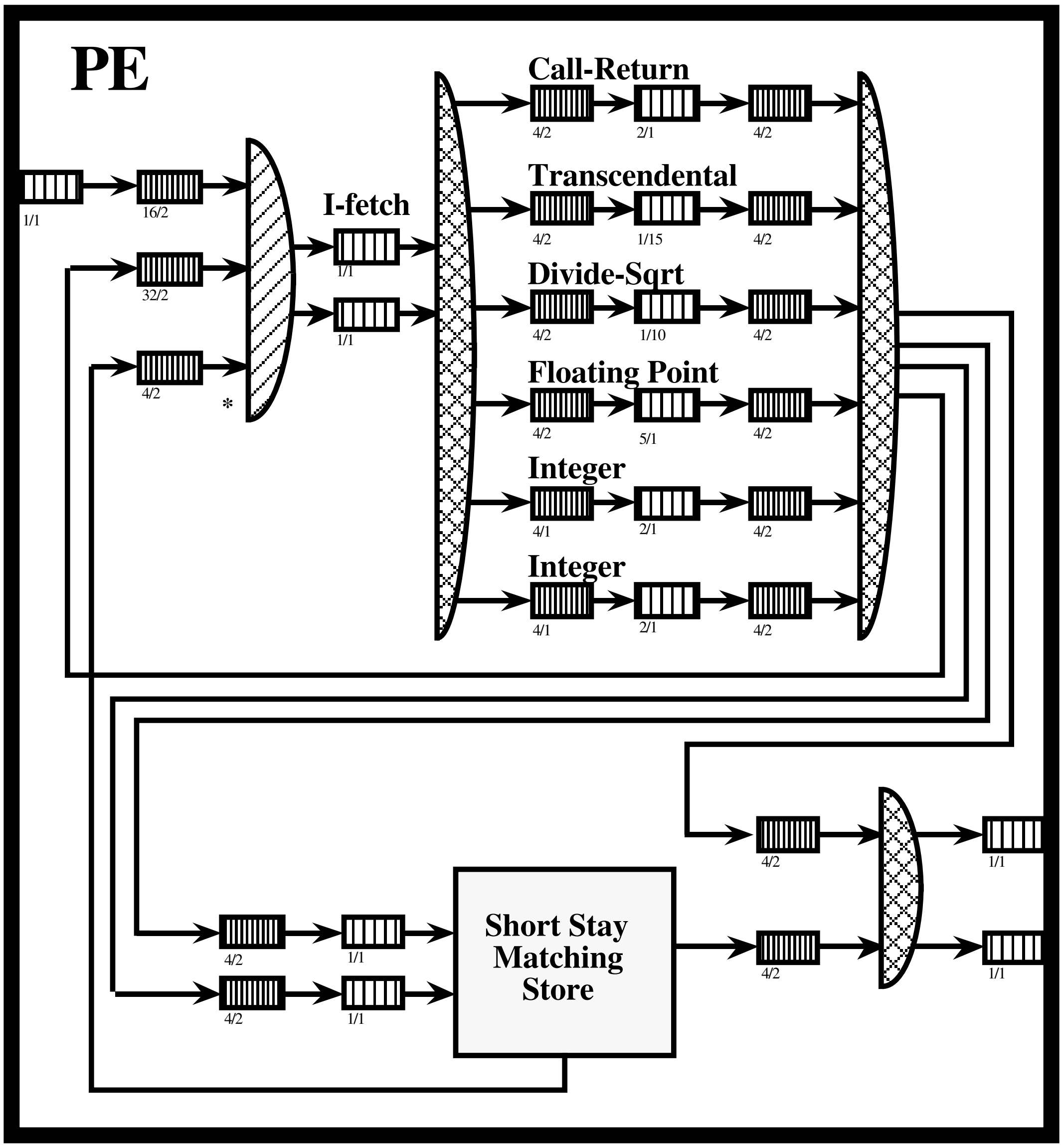


Figure 3.6: The Processing Element.

3.5.5.1. Instruction Fetch Unit

Tokens are accepted from the token queue and queued for instruction fetch along with tokens from the local by-pass queue and from the SSMS. The arbitration strategy in the instruction fetch unit is biased in favour of tokens originating from within the processing element. This enhances the exploitation of locality. Given the locality exploiting aspects of the processing element, the processor can issue two instructions per cycle even though only one token (or token pair) can enter the processor per cycle. As a result, the instruction fetch unit requires a dual-ported instruction cache.

3.5.5.2. Function Units

For the purposes of this study, a detailed analysis of the function units would be both inappropriate and premature. Therefore, the instruction-set has been divided into sections such that all operations requiring a similar number of cycles to execute are assigned to the same function unit.[43] Detailed variations within a function unit are ignored, e.g. an integer add is assumed to take the same amount of time as an integer multiply, even though former could be done faster.[44] The assignment of instruction classes to function units is given in Appendix A.2.2.1.

The timing characteristics of the function units have been chosen such that they could be implemented using current technology. The table below contains a selection of the performance times for several instructions taken from the T9000 processor [Inmo91a], the DEC Alpha processor [DEC92a], and the SDFA processor. These times include only the computational phases of the operation in question.

Instruction	T9000[45]	DEC Alpha[46]	SDFA
Integer Add	1	2	1
Integer Multiply	2 - 5	21	1
Integer Divide	5 - 12	-	10
Integer Remainder	6 - 13	-	10
Real*8 Multiply	3	5	5
Real*8 Add	2	5	5
Real*8 Divide	15	63	10

The arbitration strategy for input to the function units is round-robin-fair. The queuing just before each function unit achieves the same effect as (or possibly a better effect than) dynamic scheduling of the function units.[47] Queueing inputs to function units has been adopted by the ACRI design team [ACRI92a]. The queueing after the function units avoids short term blocking of pipelines, when several function units complete in a single cycle. This could be as high as six in a single cycle, but on average cannot exceed the instruction fetch rate of two per cycle.

The integer function units are simple, performing at their most complicated integer multiply. It was found during early experimentation that, if a two instruction fetch approach was taken, there needed to be two integer function units. Improved code generation might obviate this requirement.

---

43 This approach is quite common [Smit90a, ACRI92a, Fish88a], although not universal [Inmo91a].

44 This approach was used in an extreme form in the Denelcor HEP, in which all instructions, except divide and non-local memory accesses, required 8 pipelined cycles [Kowa85a].

45 The variations in the expected execution times of several instructions are due to other design features of the T9000, such as stack conflicts.

46 Apparently, in the design of the Alpha processor, compromises were made in the design of some sections. The integer multiplier is not pipelined and has a long latency, and integer divide and remainder are not basic instructions. The floating point divide also has a long latency.

47 Dynamic scheduling of instructions is possible in the SDFA model, since several instructions are usually queuing at the input to instruction fetch. Note also, that since Data-Flow instructions are always accessed randomly, bits in the destination field of the tag could be used to indicate to which function unit the next instruction was destined. In this way dynamic instruction scheduling could be done before instruction fetch, rather than after, as in processors like the T9000 [Inmo91a] and as in simulated dynamic scheduling such as reported in [Henn92a].

The floating point and the divide function units are standard.

The only function unit unique to SDFA is the Call-Return function unit. Its function is to generate the RecTagStack field, which maintains the call-return structure of an executing program. The function unit includes a cache which contains the parent-child relationships between RecTagStack values. It also supports the distribution of this information to other processors in a network.

Most operations, to this function unit, require only a single access to the memory. The exceptions are those, relatively rare, instructions which actually set entries in the table. The function unit, therefore, has two stages (one for memory access and one for tag reconstruction). It is assumed that, the exceptional write is performed in a second cycle, and the pipeline stalls for only one cycle.[48] The details of the Call-Return function unit are contained in Appendix A.2.2.2.

3.5.5.3. Local By-pass Queue

This queue holds tokens that do not need to be matched. In many applications the number of these by-passing tokens can be equal to the number of matching tokens. As a result, locality can be exploited easily by routing these tokens directly back to the instruction fetch unit. In Chapter 6 there are some results on how the length of this queue effects system performance.

3.5.5.4. Short Stay Matching Store

The principle behind the Short Stay Matching Store (SSMS) is the assumption that many tokens, requiring a match, will either wait for a relatively long time (such as data being passed from one function to the next) or a very short time (such as the data involved in the evaluation of a simple expression) [Gott92a, Egan91a]. The long-term success of this approach will depend on the extent to which this type of temporal locality exists in Data-Flow programs, given that the underlying mechanism is fully eager. This issue is explored further in Chapter 6.

The SSMS acts like a high speed cache, holding recently created tokens for a specified length of time. If a match should arrive within this time, the token pair is sent directly to the instruction fetch unit, thus exploiting temporal locality. However, tokens cannot be kept in the SSMS indefinitly. Eventually, the SSMS becomes full of unmatched tokens, and "older" tokens need to be forwarded to the main matching store, making room for "younger" tokens. Hence, a token's match may already be in the main matching store when it first arrives at the SSMS.

Therefore, tokens are forwarded, to the main matching store, after a fixed number of cycles in the SSMS. Once a token has been forwarded, there are still two options. It may be deleted from the SSMS or a copy may be retained, as long as space allows. In the latter case, when one of these forwarded tokens is matched locally, a *clearance* token is sent to the main matching store, removing the original token. The waiting time and the size of the SSMS are studied in Chapter 6. Note that, tokens intended for other processors are not sent to the SSMS but are sent directly to the main matching store of the other processor.

As is clear from Figure 3.6, the SSMS is built as two banks and is capable of doing two extract or wait actions every cycle. If this level of performance were to prove too great a design challenge, the SSMS could be banked, or pipelined, further. This would, of course, increase the latency in the pipeline.

---

[48] The stall, resulting from these rare write actions, is not modelled by the simulator.

Note that, one of the primary weaknesses in Data-Flow computing is that the long, circular pipeline of the system causes poor performance in sequential bottle-necks of a computation. This weakness is directly addressed by the SSMS.

3.5.5.5. Output Dispatcher

Tokens intended for other processors, tokens not able to find matches in the SSMS (within a reasonable time), and local by-passing tokens unable to fit into the Local By-pass Queue are queued for departure from the Processing Element. The output arbitrator is round-robin-fair.

3.5.6. Matching Store:

The matching store holds un-matched tokens until their matching token arrives. It supports only the extract-wait and clearance matching functions. It is a two level, multiple bank, associative memory. At level one, all banks would be implemented using directly associative memory. The level two store is much larger and would be implemented as pseudo associative memory, based on cheaper technology. This approach could be extended to several levels, where each level could be implemented using the same technology as the previous level (to increase size only), or using a cheaper technology. Only two levels have been built into the simulator to demonstrate the concept.

Arbitration into, and out of, the banks of the matching store is round-robin-fair. Within the matching store there are two processes at work, the matching process (high priority) and the migration process (low priority). The matching process is carried out whenever a token arrives at the input arbitrator of a matching store bank, and the migration process is carried out whenever the bank is otherwise idle and the current occupancy of the level one store is greater than a given threshold. In the migration process, a token from level one is chosen, at random,[49] and sent to the second level. This creates additional space in level one, allowing for further exploitation of locality.

The two levels of the matching store are packaged separately. The second level would be constructed from standard parts, but the first level would be custom designed. Because of the (single part) pin-out limit on the first level, the connection between the two levels is bidirectional and multiplexed. To ensure that it is possible to build this bidirectional connection, a slower transfer rate is assumed, than that between other major components of the system. The following rough guide, taken from David Culler [Cull92a], is used in setting the performance characteristics of the memory hierarchy:

> "... at each level [of the memory hierarchy] the latency increases by a factor of five, the capacity increases by a factor of a hundred, and the bandwidth decreases by a factor of two."

The following table gives the performance characteristics of the SDFA memory hierarchy:

[49] This is equivalent to the random replacement strategies used in conventional processors [Inmo91a, KSR91a].

Memory Level	Latency in cycles	Bandwidth in tokens/cycle	Capacity in tokens[50]
SSMS	1	1	512
Level - 1	4	0.5	64K
Level - 2	20	0.25	8M

Figure 3.7 presents the overall structure of the matching store. The number of banks and the effectiveness of the two level approach are studied in Chapter 6. Figure 3.8 depicts the internal structure of each matching store bank. The dashed line indicates the division between the two levels of the matching store. Details of the matching and migrating processes can be found in Appendix A.2.3.

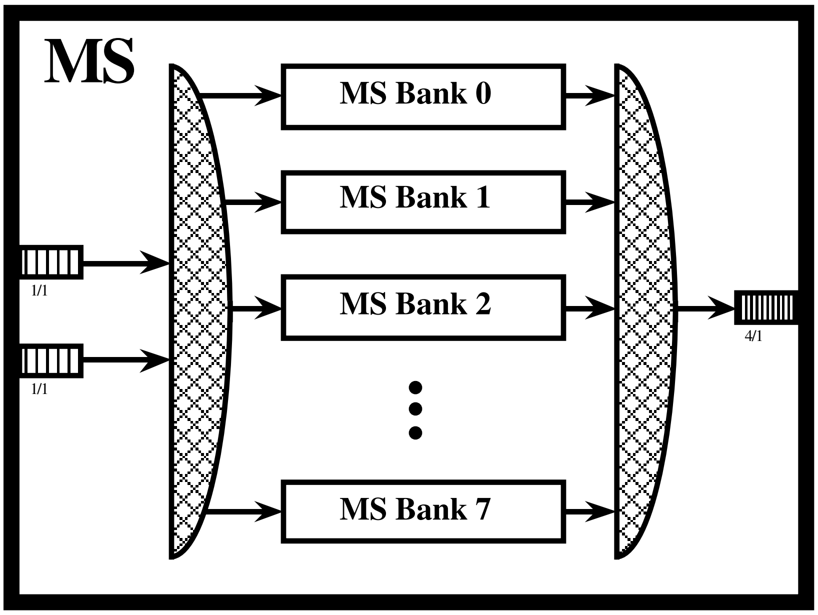


Figure 3.7: The Multiple Bank Main Matching Store.

50 These figures represent the sizes in an implementation of SDFA, but not that in the simulator. The figure of 8 million tokens per level two bank is unreasonable in a work-station based simulator and all programs tested use less than 64 thousand tokens per bank. Therefore, in simulations, these figures are adjusted in order to examine the behaviour of the matching store.

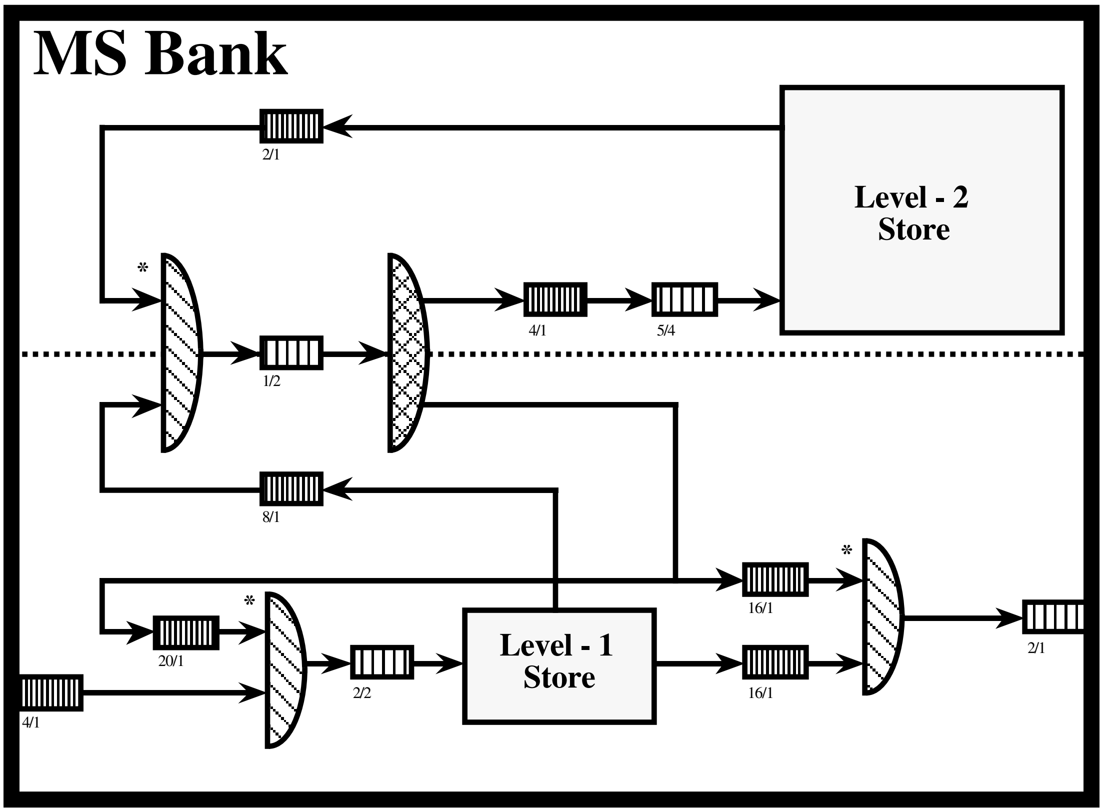


Figure 3.8: The Matching Store Banks.

3.5.7. Deadlock

SDFA, like all Data-Flow systems, has a circular pipeline. In fact it has a hierarchy of circular pipelines. Any time such a cycle exists, there is a potential for deadlock [Dall87a]. Deadlock occurs, in such a cycle, when the number of tokens reaches the maximum capacity of the cycle. Any further attempt to increase the number of tokens results in deadlock.

In the arguments that follow, it is assumed that the main matching store is of infinite capacity. A program that deadlocked, because it filled the matching store to its maximum, would simply be too large for the system and could not be run.[51] However, other potentials for deadlock must be prevented by the design of the system. These are examined in the sections that follow.

3.5.7.1. Network Level

In the SDFA network, there is a finite number of queue or pipeline slots available to store tokens. The packet switch network, however, is unidirectional and contains no cycles. It is therefore free of deadlock, as long as individual processors do not deadlock.

Other networks would require more careful analysis. There is, however, a wealth of literature on network interconnect and routing research. A recent survey can be found in [Ni93a].

51 The matching store could be backed up by disk based storage which would prevent deadlock for a much larger class of programs. However, the performance would be much the same as on a conventional system that had to run from disk.

3.5.7.2. Processor Level

Unlike the matching store, there is no explicit overflow for the token queue. It is therefore possible for a program to deadlock because the token queue has become full. However, the number of tokens in the token queue generally remains small, see Chapter 6. Alternatively, the token queue could overflow into an external memory. This would put an additional requirement on the number of pins on the chip or have a detrimental impact on performance if some pins were multiplexed.

3.5.7.3. Within the Processing Element

Since the Local By-pass queue introduces an additional cycle into the system, the possibility of deadlock exists within the processing element. In this case, the solution is to divert all by-passing tokens to the main token queue whenever the local by-pass queue is full.

Similarly, when there is rapid growth in the number of tokens requiring a match, it is possible for the inputs to the SSMS to become blocked. The remainder of the cycle can then also become full, and deadlock results. Hence, the SSMS has an escape valve built into it which, when both its input and output queues are full, releases tokens to the main matching store skipping the SSMS. This mechanism is implemented using only information local to the SSMS and, therefore, does not represent an implementation difficulty.

3.5.7.4. Within the Matching Store

Two cycles, similar to those in the processing element, arise in the two levels of the matching store, each creating the potential for deadlock. The first arises because clearance tokens from level two must be serviced by the same pipeline as incoming tokens to level one. Should the output from the matching store be temporarily blocked, a chain of full queues and pipes can develop. If the arbitration strategy at the input to the level one store were to select a token from the incoming queue (rather than the clear queue), and level one is blocked trying to send tokens to level two, the pipeline would stall. Deadlock occurs when level two attempts to send tokens, which have clearance actions required, back to level one. These can never be serviced, since the input to level one has committed to accepting an incoming token.

The second situation arises because of the multiplexing of the connection between the two levels. When the arbitrator has committed to sending a token up to level two, token pairs from level two cannot get back down into level one. As before, this can only arise when the output from the matching store bank is temporarily blocked.

In both cases, part of the solution is to design arbitrators so as to ensure that tokens which will relieve the pressure on the system always have priority. Hence, a biased arbitrator is used at both places. This favours tokens which relieve the pressure, thus ensuring that there is always the potential for movement within the matching store bank when the output is not blocked.

This priority mechanism prevents most cases of potential deadlock. There is, however, a further case which can cause deadlock in level one. If the maximum length of the clearance queue is less than the sum of the lengths of all the other queues and pipes in the cycle, deadlock is possible. If all these "upper" queue and pipe slots fill with tokens, which eventually require clearance actions in level one, and the output from the matching store bank becomes blocked for long enough, the clearance queue can fill up thus blocking the intra-level pipeline. With this pipeline committed to a downward transfer, the path upward could also become blocked which would, in turn, block the clearance queue.[52] Deadlock can be avoided by ensuring that there are enough slots in the clearance queue to accommodate all possible clearance actions from the level two store. No extra overflow is required, since clearance tokens will eventually have priority.

The choice of output arbitrator is arbitrary with respect to deadlock, but a biased one was chosen which grants precedence to matched pairs from level two over those from level one. This contributes further to relieving the pressure in the matching store bank.

This concludes the discussion of the SDFA architecture. The next chapter describes the experimental environment used in the study of the SDFA architecture.

3.6. Summary

In this chapter, a stateless Data-Flow model of execution and an architecture implementing it have been presented. Many of the features described above are motivated by the need to exploit locality, in application programs, to take advantage of memory hierarchies. The memory hierarchy in SDFA consists of the input queues to the function units (registers), the Short Stay Matching Store (cache), a multi-banked, two level main matching store (main memory), and multiple processors (remote memories). These features, it is argued, combine with the basic advantages of Data-Flow, namely latency hiding and load balancing, to create a system the can be scaled. The next chapter describes the experimental framework for an analysis of SDFA.

52 This complex collection of circumstances has only been seen experimentally when the clearance queue is short, 1 or 2 slots. This is because of the "infinite" nature of the token queue, which will always be able to accept tokens from the matching store. The only delay is, therefore, the round-robin of the matching store banks. However, since an implementation of the token queue might have a slow overflow facility, no such assumption can be made, and deadlock is structurally avoided within the matching store banks.

Chapter 4
Experimental Environment

4.1. Introduction

This chapter presents the research apparatus required to analyze the Stateless Data-Flow Architecture. In the absence of physical hardware, a research apparatus is a collection of software tools and abstractions that provide an experimental environment in which questions about the architecture can be posed and answered. The questions posed in Chapters 5, 6, and 7 range in detail from deciding the optimum queue length for the input to the floating point unit, to investigating the ability of a large multiprocessor configuration to execute application programs efficiently. The apparatus must be able to address all of these questions.

The experimental framework in which these questions can be approached must include a language interface to SDFA, a mechanism for instrumenting SDFA, a simulation system for the hardware, a collection of benchmarks, and an outline for experimental reports. These are each outlined in the sections that follow.

4.2. High Level Languages

At an early stage in this project, it was decided that a compiler would not be developed. Instead, an assembly language, SDAL (Stateless Data-Flow Assembly Language) was developed, and all benchmarks were written directly in SDAL.

Nevertheless, it is important to note the features of high level languages required by the SDFA model. One high level language, SISAL (Streams and Iteration in a Single Assignment Language) [SISAL85], is used as a base, from which the assembly language versions of the benchmarks are developed. These SISAL programs are also used in the experiments in Chapters 5, 6, and 7, in which the SDFA instruction-set architecture is compared to that of the Manchester Data-Flow Machine [Gurd85a].

4.2.1. Language Features

The single most important feature of a high level language, intended for use with SDFA, is that it must be essentially functional. Functional languages provide the referential transparently and freedom from side effects that allow programs, written in them, to be translated directly into Data-Flow graphs. This allows code generators to plant code that is free of extraneous sequentiality.[53] Also, programs which run on SDFA must be free of constructs which imply a modification of global state.[54]

---

53 It should be noted that this process is not ideal. If it were, programs describing identical algorithms would exhibit identical parallel behaviour. In practice the textual formulation of functional programs has a strong impact on the final performance characteristics.

54 At a theoretical level, this requirement is not really necessary, since modification of an element of state can be viewed as a complete recreation of the entire state. The problem is one of efficiency only.

The SDFA model assumes that the only mechanisms for replicated computation are recursion and data structure (array) creation. Therefore, a high level language which supports these features directly is preferable. The Data-Flow model is essentially eager; therefore, lazy functional languages would be inappropriate.

4.2.2. SISAL

A recursive subset of SISAL is used for representing programs in the SDFA experimental environment. The sections that follow describe those features of SISAL which are used later. There is no fundamental reason why the full SISAL language could not be supported by SDFA. These restrictions are employed for simplicity of illustration and manual code generation. "$\text{SISAL}_{\text{SDFA}}$" is used in the following sections to refer to this subset. All these features are legal SISAL, but there are many others not discussed. For a full definition of SISAL see [SISAL85]. A good short summary can be found in [Gurd87a].

4.2.2.1. Data Types

$\text{SISAL}_{\text{SDFA}}$ supports the scalar types, boolean, real, and integer, and the structured data type, arrays. The element type of an array may be any of the scalar types or another array type. The size of an array is determined at run time and may vary dynamically. $\text{SISAL}_{\text{SDFA}}$ allows the creation of user defined type names, and the types of identifiers can be determined implicitly. See figure 4.1. for examples.

```
i, j, k    : integer;      % Three scalar integers.
x, y, z    : real;         % Three scalar reals.

type Vec = array [real]; % Names 'Vec' as a type for a real array.

vec        : Vec;          % Declaration of a real array.
matrix     : array [Vec]; % Declaration of a 2D real matrix.
```

Figure 4.1: $\text{SISAL}_{\text{SDFA}}$ Declarations.

4.2.2.2. Single Assignment

As its name implies, $\text{SISAL}_{\text{SDFA}}$ supports a limited form of the assignment operator, in addition to function and expression evaluation. As with all functional languages, all statements and functions in $\text{SISAL}_{\text{SDFA}}$ can appear as terms in expressions. In addition, identifiers may be assigned values once within a basic block. This facility can be combined with a LISP like 'let' construct to build more complicated expressions. See figure 4.2 for examples.

```
let
   xsq := x*x;
   ysq := y*y;
   sum := xsq + ysq
in
   sqrt (sum)
end let
```

Figure 4.2: The Let Construct and Assignment.

4.2.2.3. Functional Abstraction and Recursion

Functions in $SISAL_{SDFA}$ can be used to encapsulate expressions and provide support for recursion. All functions are pure, in the sense that parameters are value parameters and each function returns one or more results. Both linear and multiple recursion can be expressed. Control of expression evaluation and management of termination in recursive functions is accomplished with an 'if' construct. See figure 4.3. for an example of each.

```
function fac (n: integer returns integer)  % Compute n!
   if (n < 2) then
      1                                    % Return 0! or 1!
   else
      n * fac (n-1)                        % Compute n * (n-1)!
   end if
end function
```

Figure 4.3: Selection and Recursion.

4.2.2.4. Array Creation and Reduction

Arrays can be created in $SISAL_{SDFA}$ using a 'for loop'. For example, in the matrix multiply (see figure 4.4.), the outer loop creates a two dimensional array whose elements are the reduced sums of the dot product of the rows of 'a' and the columns of 'b'.

```
for i in 1,n cross j in 1,n
returns array of
   for ka in 1,n dot kb in 1,n
   returns value of sum (a[i,ka] * b[kb,j])
   end for
end for
```

Figure 4.4: Matrix Multiply

4.3. The Assembly Language

The SDAL (Stateless Data-Flow Assembly Language) assembly language is intended to provide both a convenient vehicle for writing Data-Flow code for SDFA directly, and as a potential target language for compilers. The loader is assumed to perform all necessary code replication and in-lining, but the programmer, or compiler, is expected to plant all necessary instructions for calls to recursive functions. The BNF syntax definition of SDAL is given in Appendix B.1.1. Some example SDAL programs are in Appendix B.1.2.

4.4. Instrumentation and Measurement

Measurement is central to all empirical and experimental science. In this experimental framework, a uniform system of statistics is employed, based on two abstractions *events* and *magnitudes* (see below). A statistics module, which supports these abstractions, is incorporated into the interpreter and the simulator. The output from the statistics module can be examined both textually and graphically.

There are two basic types of data to be monitored in a simulation: the occurrence of events, the magnitude of some property in the system. Each of these data values (called a *statistic*) can either be reported directly or aggregated in time or across processors. Although collected every cycle, these statistics are reported at intervals, selected by the user, and at the end of the simulation run. At the end of each reporting interval, and at the end of a simulation, the following information about each statistic can be output by the statistics module.

Event Statistic
: The Total number of, Maximum number of, Average number of, and Standard Deviation in the number of occurrences of an event in a single cycle, can be reported for the system as a whole or by individual processor. The aggregates, maximum, average, and standard deviation, are also computed with respect to the number of processors. This information is reported at each report step and at the end of a simulation.

Magnitude Statistic
: The Maximum value of, Average value of, and Standard Deviation in the value of the magnitude during a single cycle, can be reported for the system as a whole or by individual processors. The aggregates,maximum, average, and standard deviation, are also computed with respect to the number of processors. This information is reported at each report step and at the end of a simulation. In addition, a histogram is maintained for each magnitude indicating the number of cycles that the statistic fell within a particular range.

It is possible to set specific times to start and stop data gathering during a simulation run. All statistical output is structured around the components of the system, and the level of detail reported can be controlled.

4.5. The Interpreter

The SDFA interpreter provides low level details about how various Data-Flow programs execute and exhibit parallelism, assuming that an arbitrary number of instructions can execute in a single cycle and that all other parts of the machine are transparent. The interpreter supports the stateless Data-Flow model outlined in Chapter 3. The module structure and top level pseudo-code for the interpreter are given in Appendix B.2.1.

4.6. The Simulator

The SDFA simulator models, at the "register transfer" level, a stateless Data-Flow architecture. It provides insight into the problems that might be encountered in developing a real machine that would implement a stateless Data-Flow model. The simulator supports the stateless Data-Flow model outlined in Chapter 3.

In Chapter 3, two abstractions are introduced to describe the SDFA architecture: *queues* and *pipes*. These are implemented as abstract data types in the simulator. They model the timing behaviour of the system in a two phase process. In the first phase of a simulation cycle, all components of the system are examined and tokens moved between them. In the second phase, each queue and pipe is "advanced", e.g. tokens move from one stage of a pipe to the next. It is this advancing of the queues and pipes that effects the timing of the simulation. The first phase models the activities, in the hardware, that occur between clock pluses, and the advancement models the clock pulse itself. This makes the simulation truly synchronous and allows for forward simulation of components of the system, rather than backwards.[55] The module structure and top level pseudo-code for the simulator are given in Appendix B.2.2.

4.7. Benchmark Programs

Benchmark programs define the constraints surrounding each question posed in the course of a scientific investigation, in much the same way as control groups do in experiments in biology or social science. These controls must be representative of the environmental constraints expected in reality. Given the self-imposed restriction of coding only in assembly language, these benchmarks are small. However, every effort has been to select a set of codes that, individually, test different aspects of the SDFA system. One of these benchmarks, Shallow, is well known from the meteorological community [Hoff88a].

This chapter concludes with a brief description of each bench-mark program used in this study. A listing of the SISAL code for each is included in Appendix B.3.

4.7.1. Adaptive Quadrature

Adaptive quadrature is a multiply-recursive program that computes the area underneath a function by refining a trapezoid integration method until an error tolerance is reached. Its interesting properties are that it is both numerical and requires multiple recursion. It does not use arrays.

4.7.2. Linear Recursive Factorial

This program computes n! using linear recursion and, along with the tail recursive version, is the simplest test frequently used to debug the simulator. It is also highly sequential.

4.7.3. Tail Recursive Factorial

Similar to the linearly recursive version, this program is highly sequential. However, unlike the linear recursive version, it does not produce a "stack" of partial results.

4.7.4. Gaussian Elimination

Gaussian elimination is a computation with cubic complexity, which uses both the array constructs and recursion. It has bad load balancing characteristics and is, therefore, useful in workload distribution studies. It is a non-trivial program in that it requires the replication and re-tagging of arrays because of the Stateless model. It is, therefore, used in comparisons with the Manchester Data-Flow Computing system.

---

[55] The order in which the components of the system are examined, in phase one, is the same order as token travelling through the system. This approach is more intuitive and less prone to errors. In fact, as long as there is a barrier between the phases of a cycle, and access to each individual queue and pipe is protected, each phase could be parallelized at the lowest level.

4.7.5. Matrix Multiply

Matrix multiply is the most common of all computational benchmarks. It is both trivial and extremely complex. Although load balancing well, it produces high degrees of parallelism and can create "explosions" in resource needs. Matrix multiply can be formulated in ways which exploit more locality than this rather naive implementation. It, therefore, represents an application with global communication requirements.

4.7.6. Queens

Queens is a program which computes the number of arrangements of N queens on an NxN chess board, such that no queen is threatened by another. It is intensively multiply-recursive and uses many small arrays. The load balancing characteristics of Queens are unpredictable.

4.7.7. Saxpy

This program performs a simple matrix times a scalar plus a matrix computation. It has little computation, but allows the distribution of data in a multiprocessor to be studied for relatively large problems.

4.7.8. Shallow

Shallow is the closest to an application program used in this study. It requires several array variables, including temporaries, and uses both array constructs and recursion. It is the only program in the suite that has two factors influencing the problem domain, time and space. It is, also, the largest and most complex code in the suite.[56] A discussion of this bench-mark can be found in [Hoff88a].

4.7.9. Ulam

Ulam is an integer-only program which tests Ulam's conjecture [Laga84a]. The conjecture is as follows: Given the sequence $\{T_0(n), T_1(n), \ldots\}$, where $T_0(n) = n$ and $T_{i+1}(n) = T_i(n) * 3 + 1$ if n is odd, otherwise $T_{i+1}(n) = T_i(n)/2$. For all n there exists an i such that $T_i(n) = 1$. Like the two versions of factorial, Ulam is purely sequential, but can be run for much larger values than factorial. This allows for long sequential experiments.

4.8. Experimental Method and Report

Each experiment in the following chapters is presented according to a fixed formula for experimental reporting. This experimental form follows.

Here, as in all experimental science, an experiment is intended to either test a hypothesis or obtain data to be used in making a decision or judgement about a single, simple, question. Once the question has been stated, a method is required which sets out how the answer is to be obtained, what procedures are to be followed, and what measurements are to be taken. As part of this experimental framework, a collection of assumptions, on which each experiment may depend, must be provided. All that

[56] The SISAL code for Shallow is more than six times longer than the second largest program, Queens.

remains is to present and interpret the results. The basic form of these experimental reports is therefore:

Objective or Hypothesis

Method

Measurements

Assumptions

Results

Conclusions

The experiments in Chapters 5 involve comparisons between the Manchester Data-Flow Machine and the SDFA machine. In Chapter 6, the uni-processor SDFA machine is analyzed, and Chapter 7, multiprocessor configurations are studied. The full form of this report is used in Chapter 5, but only summaries of the experiments are presented in Chapters 6 and 7, the details appearing in Appendix C.

4.9. Summary

This chapter has provided the experimental "apparatus" which will be used in the remainder of this study. The tools and experimental framework are designed to provide detailed information, much of which is not presented explicitly in this thesis. However, the information provided has proved invaluable in the development of the SDFA system as it is presented here. The following chapters are a distillation of a larger collection of design experiments.

Chapter 5
A Comparison of the MDFM and SDFA

Among the original motivations for including explicit state in Data-Flow systems, are the cost of token re-colouring entire arrays at function boundaries and the cost of replicating data structures. These are the primary costs that the SDFA system is likely to pay. On the other hand, the primary motivation behind SDFA is its ability to scale, based on the exploitation of locality. The experiments in this chapter address these two issues in a comparison with the Manchester Data-Flow Machine, MDFM [Gurd87b]. It should be noted that these comparisons are made with respect to the model of execution and not the implementation of that model. Therefore, these experiments are run on the SDFA interpretor rather than the simulator.

5.1. Token Traffic, Comparison to MDFM

The SDFA architecture does not pay a substantial computational penalty for being stateless, when compared to other Data-Flow machines.

It is clear from the description of the SDFA model that the absence of explicit state, carries with it some costs other than that of large associative, or pseudo-associative, memories. These costs manifest themselves in two ways. First, there is the potential that additional copies of data structures might be required in the SDFA machine that are not required in other Data-Flow machines. Second, since SDFA data structures are passed in their entirety through function boundaries, rather than just the pointers, the cost of re-colouring data structure elements could prove excessive.

There is reason to believe that the impact of these factors is not excessive. In any Data-Flow machine, the bulk of the extra copying and colouring must be performed anyway. However, the part of the machine performing the colouring may vary depending on the computational model. For example, in the Manchester machine (MDFM) colouring of data structures is done in the structure stores rather than in the processing element. Also, these extra costs could be partially compensated for by inlining techniques, and balanced against the costs of garbage collecting in the structure memories. A direct comparison between the SDFA machine and the MDFM is, nonetheless, warranted.

5.1.1. Method

In this experiment, a variety of codes (Aq, Gauss, Mm, Queens, Shallow)[57] are executed on the Manchester Data-Flow Machine Simulator (NMR) [Teo91a, Gurd92a] and the SDFA interpreter and simulator. The aim is to find out how much "work" is done by each system and ascertain if the penalty paid by the SDFA machine is substantial. Each of these programs is run for various sizes. From these data, regression is used to derive functions describing the total workload, with respect to problem size.

57 A description of each of these is provided in Chapter 4.7.

The SDFA codes are hand written in assembler, so every effort must be made to find the best possible implementation, through various compiler options, of the MDFM versions. All the following options are applied at the IF1 to Data-Flow code translation phase of the compilation process [Sarg85a]:

-v	Vectorize the code in loops, eliminating redundant addressing.
-clb 0	Generate code assuming all arrays have zero lower bound.
-inl N	Inline functions until their size exceeds N IF1 nodes.
-blo N	Inline recursive functions until their size exceeds N IF1 nodes.

Their use in the various programs was as follows:

Aq[58]	-blo 250
Gauss[59]	-v -clb 0
Mm	-v -clb 0
Queens[60]	-clb 0 -blo 500
Shallow	-v -clb 0 -inl 1000

Recall that the SDFA instruction-set is simple and therefore, frequently, several instructions are required where one suffices in the MDFM. Recall also, that the instruction-set is designed to support the possibility of a more complex instruction-set architecture. In addition to performing these tests on the interpreter, some are also run using a Complex Instruction-Set (CIS) switch.[61] This is done to provide an indication of how much of the penalty is due to the relative simplicity of the SDFA instruction-set.

---

58 The recursive inlining was tried for increasing values until the performance improvement became constant. In this case, as with Queens below, this was a substantial increase in the number of nodes, namely, an increase of 1180%.

59 This implementation of Gaussian elimination is performed "upside-down" so that the constant lower bound optimization could be applied.

60 In this case, the code size increased by 1560% due to recursive inlining.

61 In this mode of execution, up to three operations can be performed as a result of a single match. The restrictions are that 1) the second two instructions must be simple integer operations (on either the data or tag fields of the token), 2) the second two instructions must each have only one token as input, and 3) no more than two tokens can result from the collection of instructions. These collections may be arranged in any of the following forms and are counted as a single instruction. Also, the tokens passing between these instructions are not part of the total token count.

5.1.2. Measurements

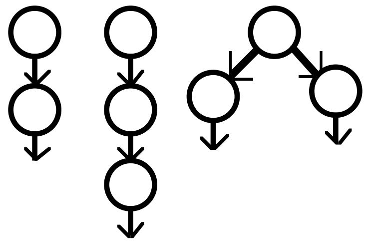
The difficulty in comparing any two systems, as distinct as the MDFM and SDFA machines, arises from the need to measure something in common to both systems that, in this case, represents the amount of work done by the systems. Instruction count is inappropriate since the complexity of the MDFM's instruction-set far exceeds that of the SDFA system, particularly when the iterative instructions are considered.[62] Operation count is ruled out on the basis that defining what constitutes an operation can be rather difficult. The use of cycle count assumes that the architecture is independent of technology, clearly a falsehood. Likewise, using execution time is out of the question.

Since all data, in both systems, are carried on tokens, the number of tokens generated by all components of the system is taken as a measure of the total work. The creation of a token, as an abstraction of work done by a Data-Flow system, is quite natural. This is a standard statistic gathered by the SDFA software and is easily obtained from the MDFM NMR simulator. Notice that this count includes tokens generated by the MDFM's structure stores as well as by its processing elements.

5.1.3. Assumptions

The Stateless Data-Flow Assembly Language (SDAL) codes are hand written, thus creating the potential for an unfair comparison. Two factors limit the degree of concern here. First, for the numerical codes at least, the SISAL compilation system is known to be rather good. This is highlighted by the instructions per floating point operation ratios for the various codes (Aq - 2.6; Mm - 2.0; Shallow 4.6; and Gauss - 3.0).[63] Second, the SDAL code is written for a macro assembler; therefore, a small library of frequently used SDAL code fragments is employed to simplify code generation. Many redundant operations could be removed by techniques such as loop unrolling and loop invarient removal, if the macros were not used. This is demonstrated by a test in which Mm is unrolled, see results below.

Garbage collection, which consumes compute resources in the structure stores,[64] is not included in the computation of the figures for the MDFM, since it does not create explicit tokens. Therefore, the results for the MDFM are slightly optimistic.



Note that this is implemented as a dynamic action taken by the simulator and not a re-coding of the program. It is expected that better results could be achieved if code were rewritten, or compiled, with this in mind.

62 Iterative instructions were included in the MDFM's instruction set to accelerate the creation of arrays and for expanding scalars into arrays for combination with existing arrays [Böhm90a].

63 This ratio has its failings as a measurement, but it has nonetheless been used in the past [Böhm90a].

64 Every location in the structure store has associated with it a presence bit (currently assumed to be part of the main store, but which could be moved to a smaller fast memory [Gurd92a]) which must be cleared when the data structure is garbage collected. With most of these programs, this is equivalent to an extra write access for each normal write access. Garbage collection is not affected by multiple reads. This is due to reference counting.

Traditional Data-Flow machines, other than the Manchester machine, behave similarly. There is little, or no, published data with respect to how these machines compare to each other, but their fundamental behaviour is relatively uniform.

5.1.4. Results

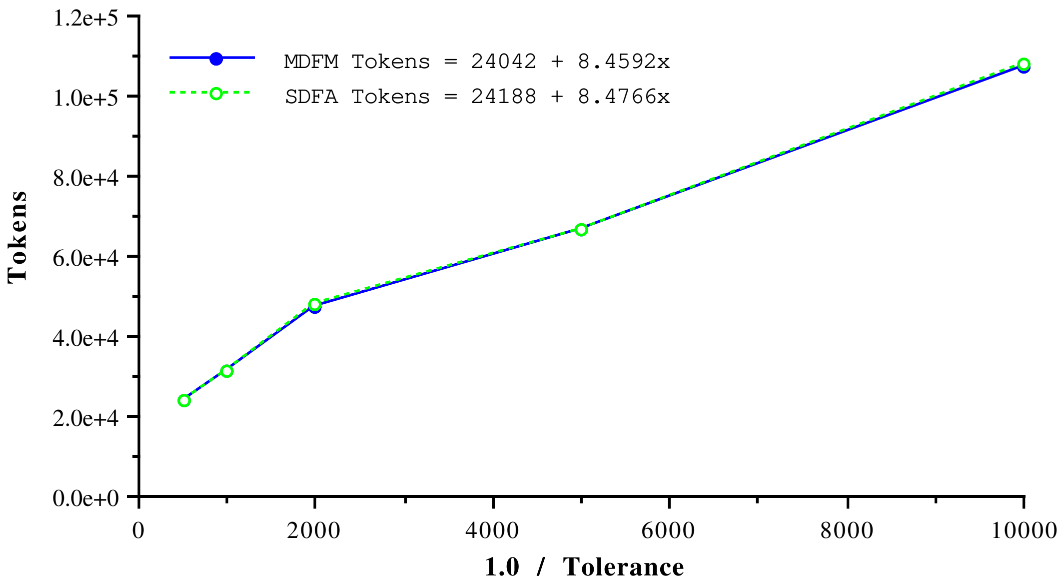


Figure 5.1: Token Count in Adaptive Quadrature (Aq).

Notes:

- The amount of work done by Aq depends on a parameter, called the tolerance, which determines the accuracy of the area computation. The reciprocal of the tolerance is used as the x-axis so that an increasing function can be devised and plotted.

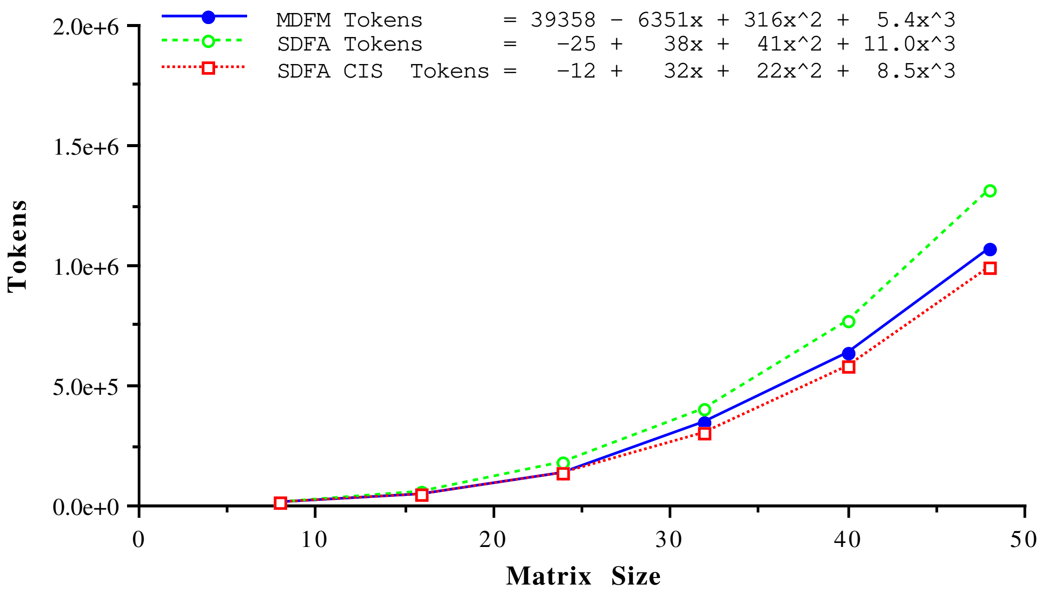


Figure 5.2: Token Count in Gaussian Elimination (Gauss).

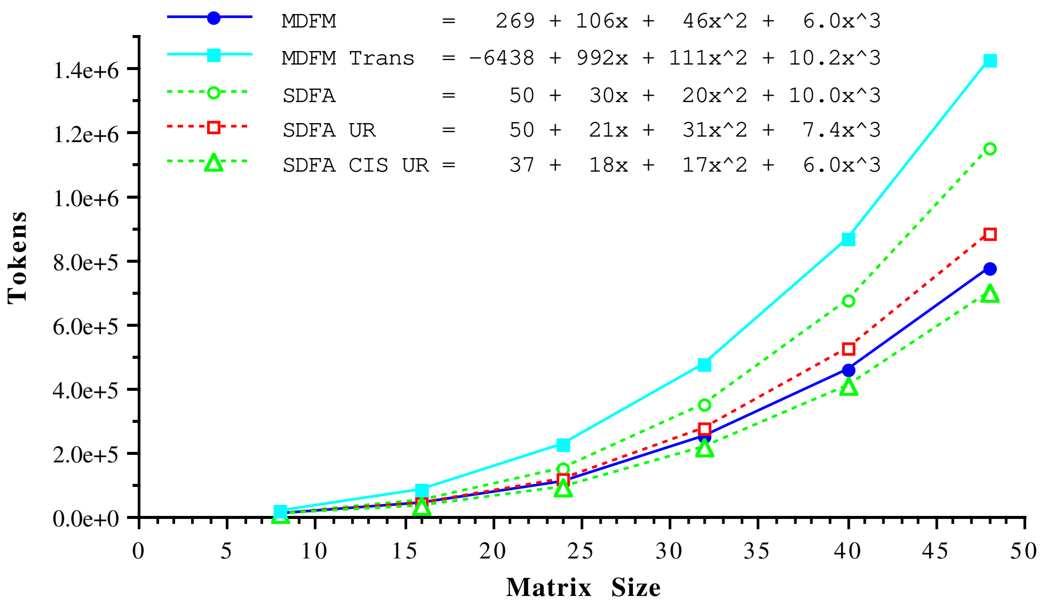


Figure 5.3: Token Count in Matrix Multiply (Mm).

Notes:

- The shorthand in the legend above is translated as follows: The MDFM case is the original matrix multiply used regularly in MDFM studies [Sarg86a, Teo91a]. It is written under the assumption that the second matrix is already transposed. The second case (MDFM Trans) includes the transposition of the second matrix, as do all the SDFA cases. The first SDFA case is the original. In the second (SDFA UR), the inner loops have been unrolled to a depth of eight, likewise for the last test, which also employs the CIS facility.

- The substantial cost penalty, paid by the MDFM, for performing the transpose is a result of the technique used to store two dimensional arrays. Two dimensional arrays are not first class constructs in the execution model, rather they are represented as an array of pointers to one dimensional arrays.[65]

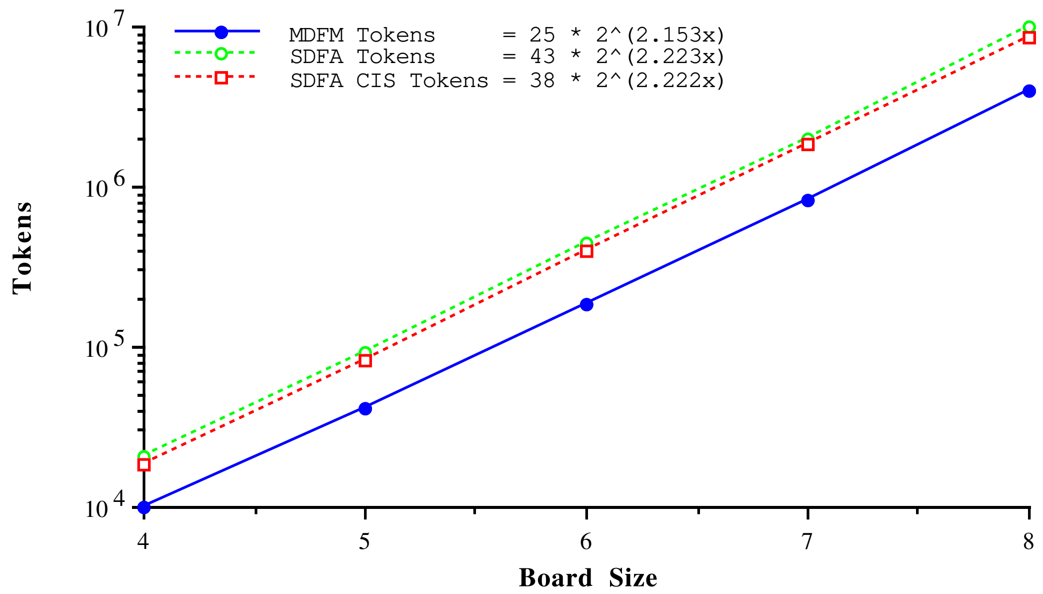


Figure 5.4: Token Count in the N Queens Problem (Queens).

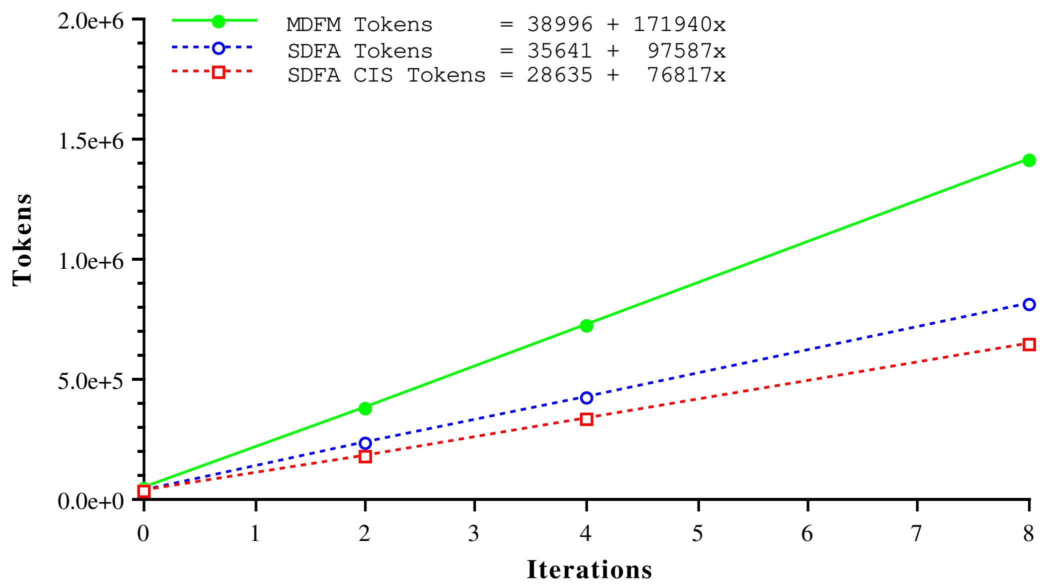


Figure 5.5: Token Count versus Iterations in the Shallow Water Equations (Shallow).

65 This is partially an artifact of the base language, SISAL, which supports this model for arrays. However, if the execution model supported multidimensional arrays, a compiler could convert this to a multidimensional array representation.

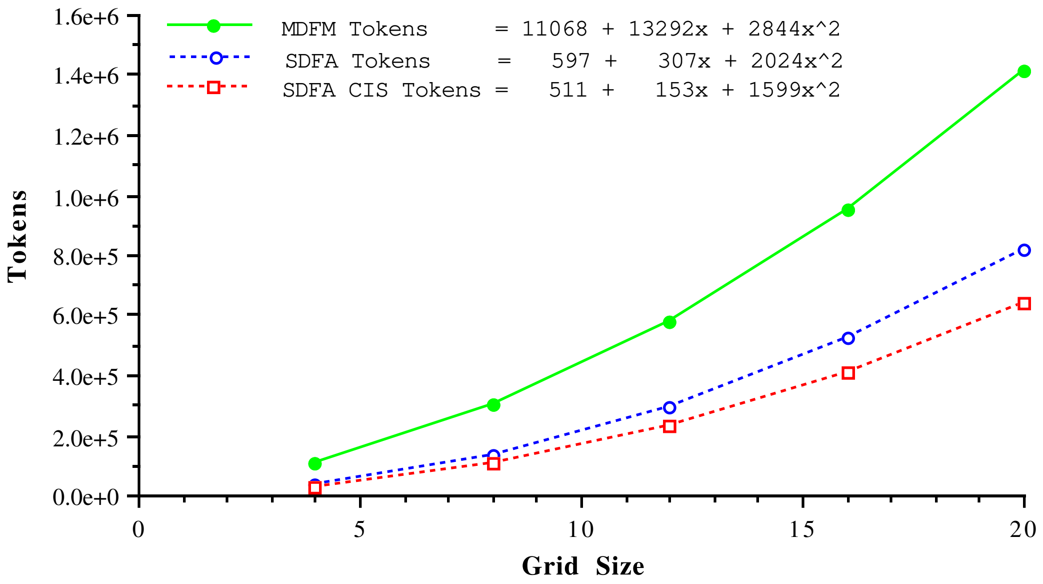


Figure 5.6: Token Count versus Grid Size in the Shallow Water Equations (Shallow).

Notes:

- Shallow has two parameters which control its complexity and raise different issues; therefore, two experiments are presented. In the first, the grid size was fixed at 20, and in the second, the number of iterations (or time steps) was fixed at 8.

5.1.5. Conclusions

Two things are apparent from the above experiments. First, there is a price to be paid for statelessness in several cases, and second, it does seem that part of this price comes from the simplicity of the instruction-set. In all the numerical tests, the SDFA model required, at most, the creation of a few extra tokens in the inner loop, the worst case being Gauss, figure 5.2, in which approximately twice as many tokens are required. This improves to 1.6 times when a more complex instruction-set is considered. In other cases, the SDFA model is superior to the MDFM, particularly with Shallow (figure 5.6.) where the MDFM approach required 1.4 times as many tokens as SDFA and Mm (with the transpose) where the MDFM model required 1.02 times as many tokens. These numbers improve to 1.8 and 1.7 when the optimized approaches are taken on the SDFA system.

Note that because each iteration of Gauss requires a recursive call on the SDFA machine, and not on the MDFM, this experiment represents a re-colouring intensive test.

The poor performance on Queens, figure 5.4, is more serious since, although the difference in coefficients is small, it is in the exponential part of the formula. Techniques such as recursive inlining, as used in the MDFM case, might help some. However, with problems exhibiting non-polynomial growth, the likelihood of a problem being too large to solve solely because of this factor, is quite small.

The most important factors in the future success of a SDFA model are, therefore, improvements to the instruction-set and the ability to compile for it. The former is best highlighted by a closer look at a key factor in the inner loops of most programs. Recall that, in the MDFM there is an instruction which executes for several cycles, issuing one new token per cycle. In SDFA, proliferate is performed by a simple macro (figure 5.7.). This approach requires the creation of three tokens for each result token. This is improved to two when the CIS facility is employed, which is the best possible unless more than two result tokens can be produced by a single instruction.

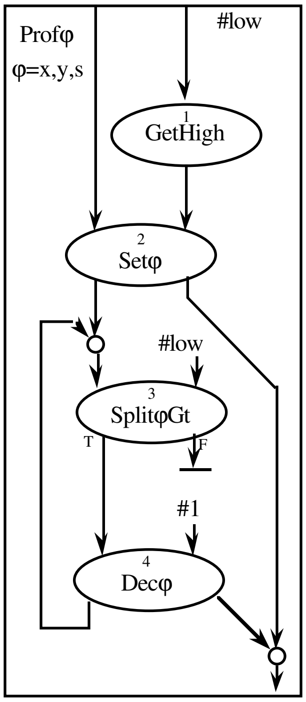


Figure 5.7: Data-Flow Graph for the Proliferate Macro.

5.2. Token Locality, Comparison to MDFM

The distributed memory model underlying the SDFA architecture produces substantially less network traffic than the shared memory approach of traditional Data-Flow systems.

The fundamental motivation behind the SDFA architecture is to benefit from the distributed memory model underlying it. The effectiveness of this is studied, in later experiments, with respect to the SDFA machine in isolation. However, it is appropriate to provide some comparative results.

5.2.1. Method

In most cases, the largest versions of the programs from Experiment 5.1. above are run on a 16 processor MDFM and SDFA systems. The exception is Queens which case a board size of 7 is used. As with Experiment 5.1, highly optimized versions of the MDFM codes are used. Only the original versions of the SDFA codes are used, however.

Both systems use a hashing mechanism to perform token distribution across the multiprocessors. The SDFA function is designed to create spatial and temporal locality. In the SDFA function, none of the bits of the destination address and only the high order bits of the index fields are used. The MDFM distribution function, however, includes all the tag bits. This creates an almost total lack of locality. However, as a result of recent changes in the MDFM design [Gurd92a], tokens intended for the processor on which they are created can take a "short-cut", avoiding the switch.

For the purposes of this study, the important part of the MDFM traffic is that which travels to and from the structure stores, because it is this that will always be global,[66] regardless of what locality facilities are incorporated in the MDFM.

This experiment compares total network token traffic on the SDFA system and total network traffic and structure store traffic on the MDFM. This experiment is run using the SDFA interpreter and the MDFM simulator.

5.2.2. Measurements

For the SDFA system two measurements are taken, the total number of tokens created and the number of these which travel between processors. This is used to compute the percentage global traffic. For the MDFM three measurements are taken, the total number of tokens created, the number of these which travel over the switch network, and the number travelling to or from the structure stores. These values are used to compute the percentage global traffic and the percentage structure store traffic.

5.2.3. Assumptions

Other Data-Flow systems exhibit similar behaviour to that of the MDFM. Since data structures are colourless in other Data-Flow systems, they will have a shared memory architecture and, therefore, should have similar characteristics.

It is acknowledged that the global traffic in the MDFM is a worst case, due to the distribution function and basic architecture.[67] The structure store traffic, on the other hand, represents a best case since it assumes that <u>all</u> non structure store traffic is local.

There is always a danger with normalized results that something is being obscured. In this case, the SDFA system usually generates more tokens, for the same problem, than the MDFM system, and hence has a larger number of tokens which could be local. From figure 5.8. it is clear that, with the exception of Queens, the actual difference is not excessive, nor does the SDFA system always generate more tokens, e.g. Shallow.

---

[66] In this 16 processor experiment, an average 1/16-th of this traffic could be counted as local if the structure stores were integrated into the processing elements. It would be difficult to arrange for more than 1/16-th of the structure store traffic to be local, since data structures are colourless, see Chapter 2. Therefore, all structure store traffic is considered global.

[67] To improve locality in the MDFM, the destination node address could be removed from the distribution function. It should be noted that the latter could have an impact on the machine's load balancing ability.

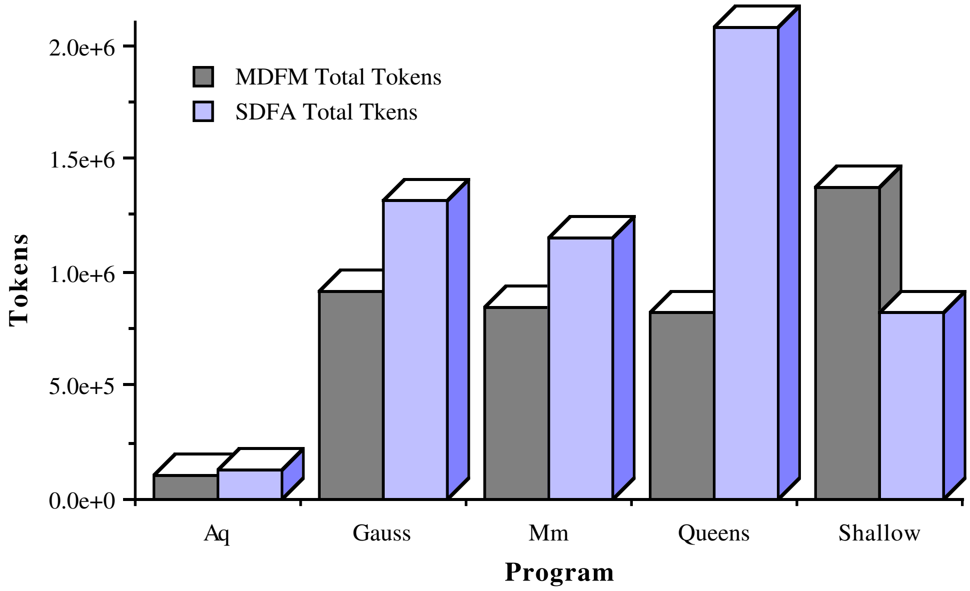


Figure 5.8: Total Tokens Generated by the Benchmarks.

5.2.4. Results

The results obtained from the percentages study are presented in figure 5.9.

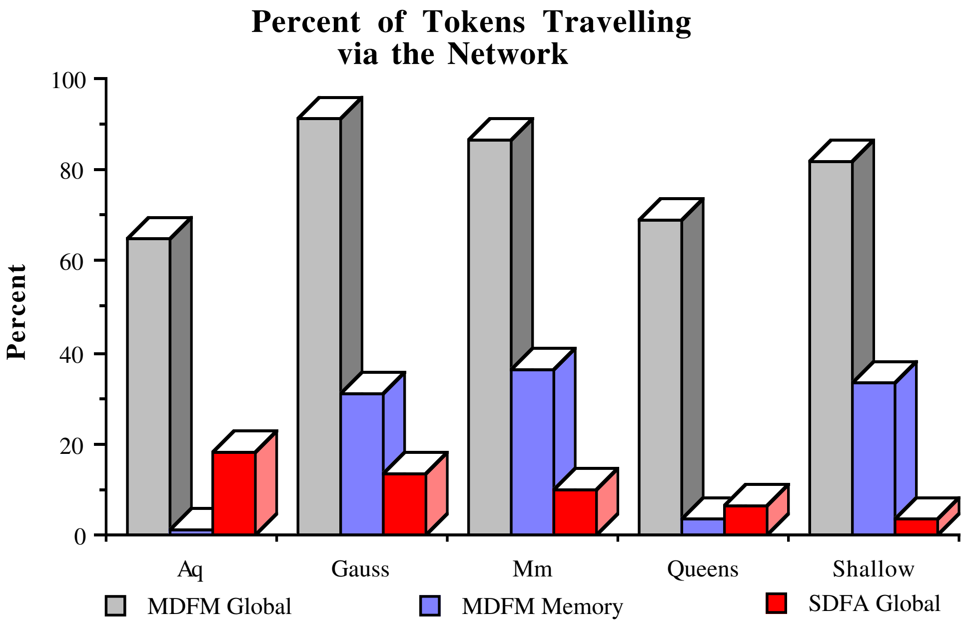


Figure 5.9: Network Token Traffic.

- The low figures for the MDFM Memory traffic on Aq and Queens, are due to the nature of the programs. Aq uses the structure stores only for activation name recycling counters, and Queens has one array for each board, a maximum of 7 elements long.

5.2.5. Conclusions

The provision of a simple distribution function in the SDFA machine can create locality in a variety of programs. In comparing these two systems, it is clear that a much simpler communications network should be adequate for the SDFA system, compared to the MDFM system. Chapter 7.3. investigates the design of a switch based on this ability.

5.3. Summary

Form these experiments, it appears that the cost paid for statelessness is not extreme and that the potential for exploiting locality is substantial. The next chapter investigates the design of a SDFA uni-processor, and Chapter 7 investigates the SDFA multiprocessor..

Chapter 6
Analysis of a
Uni-Processor SDFA System

The primary goal in the design of the SDFA uni-processor is to develop mechanisms for exploiting locality within a memory hierarchy. If these function effectively, the ability of the system to scale, to large numbers of processors, is enhanced. In addition to this fundamental issue, the question of throttling must also be addressed. Experiments relating to these issues are presented in this chapter.

In this chapter, and the next, many of the details of the experiments are in Appendix C. rather than in the main text.

6.1. Effectiveness of Local By-pass Queue

A by-pass queue, inside the processing element, can effectively increase the number of tokens processed locally.

A Data-Flow system, which supports a super scalar architecture, must support local computation, since input to the processor is likely to be more restricted than the instruction fetch. In the SDFA design, input to the processing element is limited to one token (or token pair) per cycle and there are two instruction issues possible per cycle. Therefore, in order to feed the instruction fetch units, some locally generated workload must be processed locally. The local by-pass queue and the Short Stay Matching Store serves this function. In this experiment only the local by-pass queue is examined. The details of this experiment are in Appendix C.1.

6.1.1. Results

The table below contains the data from this experiment. The times are in milliseconds, and the percentage of all by-pass tokens, which used the local by-pass queue, is also included. Two types of local queue are used, one accepting 1 token per cycle, the other 2 tokens per cycle.

Program	No By-pass	1 By-pass/cycle		2 By-passes/cycle	
Mm	1.683ms	0.880ms	63.4%	0.881ms	97.7%
Queens	1.858ms	1.125ms	71.0%	0.953ms	92.0%
Shallow	1.785ms	1.041ms	66.1%	0.940ms	88.2%

6.1.2. Conclusions

The local by-pass queue is an effective technique for exploiting locality. However, its effectiveness in multiprocessor systems is likely to be degraded some,[68] and how it is effected when the Short Stay Matching Store is enabled is demonstrated in the next experiment. Although the time benefit for allowing two by-passes to be handled in one

[68] Unlike the uni-processor, not all tokens can remain local. Some must be sent to other processors.

cycle is small, the increase in the number of tokens remaining local is worth exploiting. In subsequent experiments, the two by-passes per cycle configuration is used.

6.2. Design of the Short Stay Matching Store

Determine effective operating parameters for the Short Stay Matching Store.

As is clear from Experiment 6.1, temporal locality can be exploited effectively with by-pass tokens. A more effective locality mechanism would also exploit matching tokens as well. The Short Stay Matching Store (SSMS) is designed to accomplish this.

The SSMS has two operating modes "simple" and "rectifying". In the simple SSMS, tokens are held up in the SSMS for several cycles and then forwarded to the main matching store. In the rectifying SSMS, the mechanism is more complicated. As before, tokens not matched after the prescribed number of cycles are forwarded to the main matching store. However, as long as space permits, a copy of this token is left in the SSMS memory. Should a match occur in the SSMS, it is extracted and a clearance token is sent to the main matching store to remove the previously forwarded token. Tokens intended for other processors are not sent to the SSMS

The design decisions are, therefore, to determine which style of matching store should be used, how long tokens should wait before being forwarded, and how large the SSMS memory should be. The details of this experiment are in Appendix C.2.

6.2.1. Results

The larger the SSMS is the better it performs. The largest, reasonable size for the SSMS is 512 tokens, see discussion in Appendix C.2. These experiments, therefore, are run assuming an SSMS size of 512. Also, due to the effective latency hiding aspects of SDFA, the execution times vary only marginally, see Appendix C.2 for details.

The following table contains the percentage of all matches which are performed locally for each program, SSMS type, and delay time.

	Simple			Rectifying		
Delay	Mm	Queens	Shallow	Mm	Queens	Shallow
16	1.57%	40.00%	7.73%	3.09%	52.38%	14.09%
32	1.59%	43.93%	9.04%	3.17%	52.76%	13.59%
64	1.65%	46.36%	10.24%	2.41%	52.88%	13.64%
128	1.62%	47.88%	10.84%	2.10%	53.26%	13.68%

Note that the percentage matched locally in the rectifying SSMS remained constant as the delay time was varied, indicating that SSMS size is more important than delay. However, the length of the delay can affect the number of tokens which must be cleared from the main matching store and, hence, the amount of extra token traffic. Figure 6.1. highlights this effect.

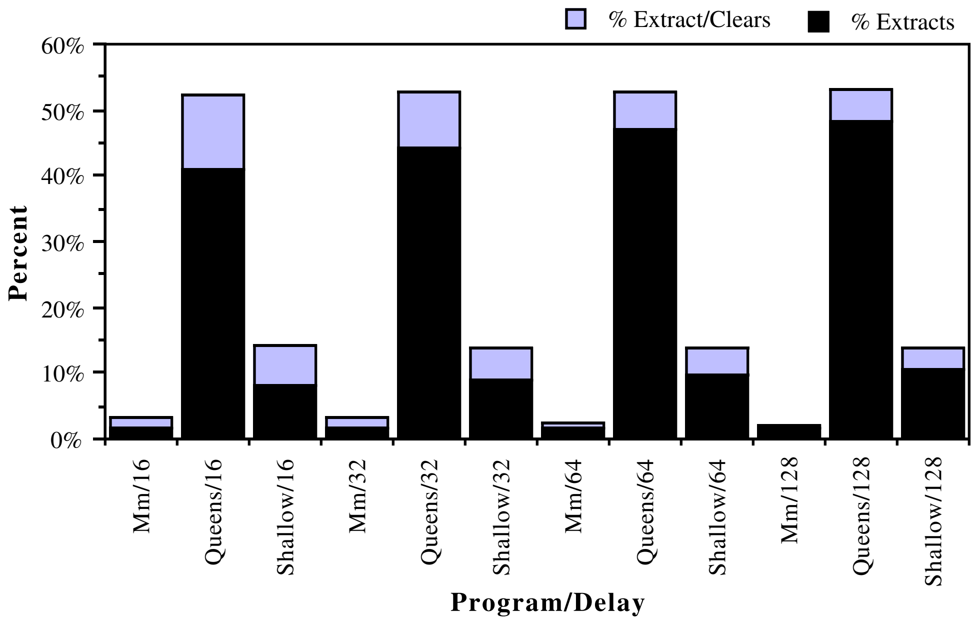


Figure 6.1: The effect of increasing the delay in the SSMS on clear actions.

6.2.2. Conclusions

The rectifying SSMS is clearly more effective than the simple SSMS, and there is little loss in performance due to the increasing delay, but that the number of tokens that must be sent to the main matching store drops steadily as the delay is increased. However, the key factor, effecting the number of local matches, is the program itself.

The locality features of the SDFA system are clearly effective. The design parameters chosen for all further experiments are a rectifying SSMS, of size 512, with a delay of 64 cycles. The overall success, of the locality mechanisms in SDFA, is also clear. In experiments using these design parameters, 60.0%, 68.1%, and 49.2% of all tokens were processed locally in Mm, Queens, and Shallow respectively.

6.3. Tuning the Function Unit Queues

Tune the length of the function unit input and output queues.

In this experiment, the goal is to tune the length of the function unit input and output queues. The details are given in Appendix C.3.

6.3.1. Results

The following table contains the times for the runs with long queues (128) and with queues set to realistic values 16, 8, and 4. All times are in milliseconds.

Program	Long Queues	Length = 16	Length = 8	Length = 4
Mm	0.871	0.870	0.871	0.867
Queens	1.197	1.195	0.947	0.977
Shallow	0.907	0.911	0.923	0.951

6.3.2. Conclusions

The Queens program is exhibiting rapid growth in the number of tokens, see Appendix C.3.4. Without some throttling mechanism, this program simply fills the queues to their maximum and keeps them that way. The other two programs saturate the machine without causing rapid growth in the number of tokens. Relatively short queues suffice in most cases. Therefore, queue lengths throughout the machine, unless altered for other reasons, have been set to 4. Notice that even in the case of Queens, the short queues have had little impact on the final performance. In fact, the short queue versions are sometimes slightly faster.

6.4. Super Scalar and CIS Study

Determine an effective combination of multiple instruction issue and complex instruction-set implementation.

The SDFA simulation system supports both multiple instruction issue and, emulation of, complex instruction formats. The former is a central element of the SDFA design philosophy, and the latter, detailed in Chapter 5.1, can reduce the total token traffic. There is a fundamental trade off between the two approaches. Multiple issue requires more complex instruction cache management, while the complex instruction-set requires replication of function unit hardware. This hardware is needed to perform the additional operations that form part of each complex instruction.

There are four different approaches possible in SDFA: 1) single issue, simple instruction; 2) single issue, complex instruction; 3) double issue, simple instruction; and 4) double issue, complex instruction. In this experiment, the performance characteristics of these strategies are studied and an optimal arrangement selected. The details of this experiment are in Appendix C.4.

6.4.1. Results

Figure 6.2. shows the execution times for each of the programs using the four approaches.

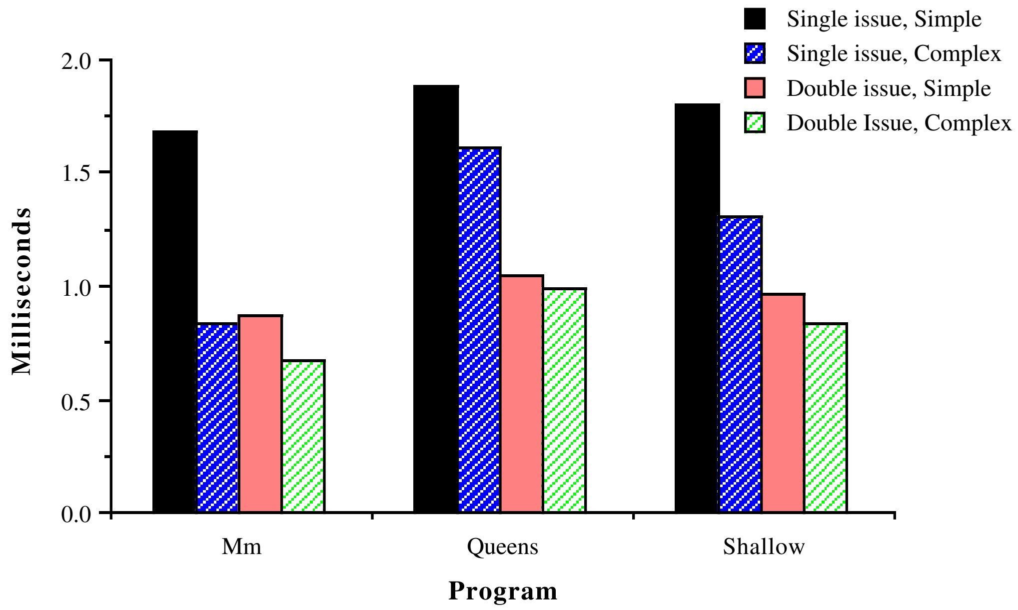


Figure 6.2: Execution times of various instruction issue strategies.

6.4.2. Conclusions

The benefit of double issue is clear. However, the advantages of a more complex instruction-set are not as clear. Although there is a performance improvement, it is at a cost in efficiency, see Appendix C.4. The single issue approaches are the most efficient, issuing an instruction virtually every cycle. The double issue, simple approach is also quite efficient, issuing between 1.8 and 1.9 instructions per cycle. The efficiency of the double issue, complex approach is much lower, ranging from 1.2 to 1.6 instructions per cycle.

The operations per instruction rate of the double issue, complex approach is limited by the number of tokens that can be produced by a single instruction, two in SDFA. Increasing this number would increase the volume of communication from the processing element to the matching store and token queue, which is already strained as indicated by the low efficiency.[69]

Hence, the time benefits do not justify increasing the complexity of the function units to support more a fully general complex instruction-set. The double issue, simple instruction-set model is adopted for the remainder of these experiments.[70]

---

69 This conclusion is confirmed by the fact that the reduction in issue rate is a result of the instruction fetch being blocked 8 to 30 times more frequently than it is starved. This is caused by a failure of the processing element to send tokens to the matching store and token queue rapidly enough.

70 These issues are re-examined in Chapter 7.5, where a couple of instructions are added which substantially improve the performance of scientific codes.

6.5. Sequential Bottle-neck Study

The locality exploiting features of the SDFA architecture improve the performance of sequential code

Sequential bottle-necks, which arise naturally in many parallel applications [Böhm92a, Arvi86a], have a severe impact on the performance of Data-Flow systems, due to the long pipelines. The SDFA architecture's local by-pass queue and Short Stay Matching Store reduce the length of this pipeline when computation is local. This experiment investigates the success of this approach. See Appendix C.5. for more details.

6.5.1. Results

Figure 6.3. displays the percent improvement in the execution time, over the configuration without a local by-pass or SSMS, when these features are added.

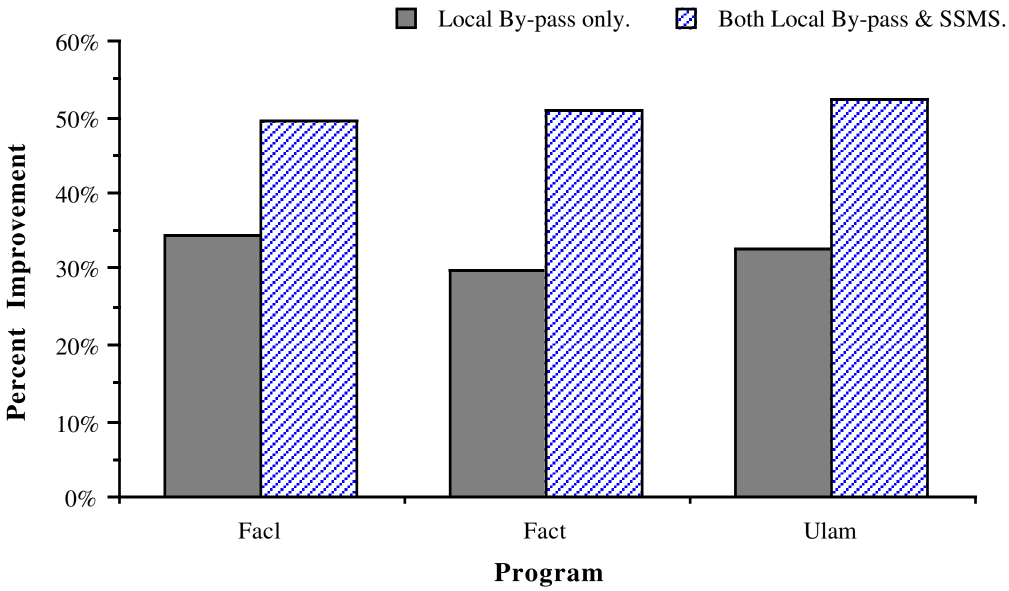


Figure 6.3: Improvement in sequential performance due to locality mechanisms.

6.5.2. Conclusions

There is substantial improvement in the performance of sequential code as a result of the local by-pass and SSMS. Note that the SDFA hashing scheme for distributing the workload across multiprocessors does not include the destination address, as it does, for example, in the Manchester Machine [Gurd87a]. This feature of the SDFA architecture, in most cases, ensures that sequential computation is performed on only one processor, rather than distributing it across multiple processors. The MDFM approach incurs penalties of inter processor communication as well as losing the benefits of local computation.

6.6. The Need for a Throttle Mechanism

As problem size increases, the greedy scheduling policy underlying Data-Flow causes store explosions in SDFA.

A parallel solution to a problem requires more memory than a sequential one, a space versus time trade off. This becomes a problem when memory use increases in excess of what is required to support a given level of parallelism. The memory use should increase in accordance with increases in problem size, but not by more than this. The greedy scheduling policy underlying Data-Flow can cause many partial computations to be performed which occupy memory but are not necessary to sustain pipeline saturation. This is called a "store explosion", and this experiment verifies that store explosions can occur in SDFA. The details of this experiment are in Appendix C.6.

6.6.1. Results

Figure 6.4. shows the sum of token queue and matching store occupancy, and the memory required in an ideal sequential implementation, for Shallow. Both maxima and averages are shown. Similar results were seen for Queens and Mm, see Appendix C.6.

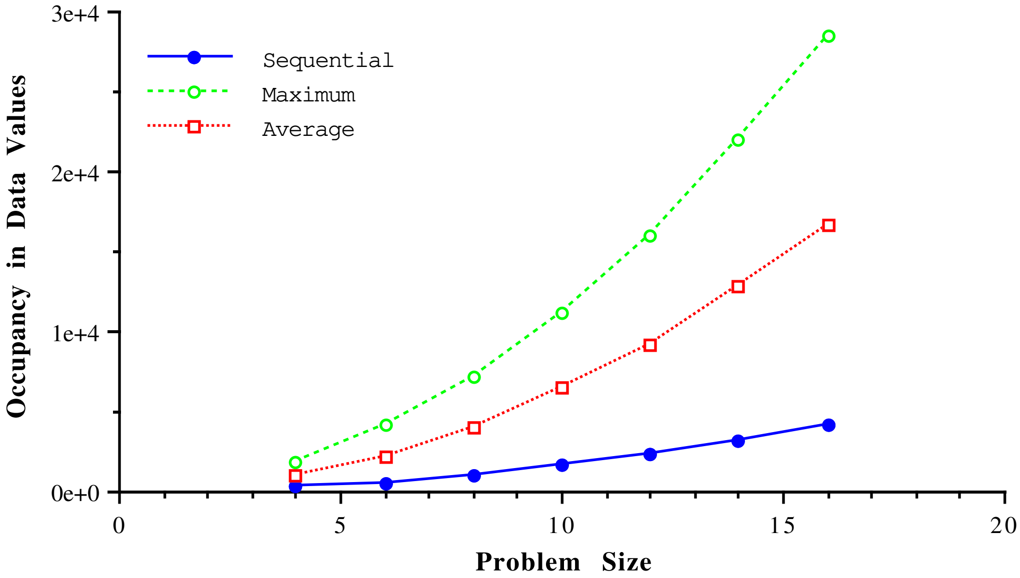


Figure 6.4: Store explosion in Shallow.

6.6.2. Conclusions

A store explosion occurs in all cases. The importance of this cannot be understated. Excessive store usage in a system like SDFA, that relies heavily on associative memory, drives up the cost dramatically. Throttling is the mechanism used in most Data-Flow systems to control store use. The next two experiments investigate two possible throttle mechanisms in SDFA.

6.7. Software Throttling

Existing software-based throttling techniques are effective in reducing store use in SDFA.

There are a variety of software based techniques for reducing store use in Data-Flow programs, see Chapter 2.1.3.5. Two of these techniques are used in this experiment with the program Shallow. In the Experiment 6.6, the rapid growth in tokens occurs because the replication of scalars, for all time-steps (6 steps in this case), begins at the start of the program, thus filling the matching store with values not required for some time. The two software solutions implemented here are: 1) make the main program strict so that only one iteration at a time can execute, and 2) imitate the K-Bounded loop technique, for K=2 [Arvi86a]. It is known that both of these techniques can restrict parallelism. The graph in figure 6.5. depicts the ideal parallelism inherent in all three approaches. The results are similar to those obtained at MIT for the Simple code, when the K-Bounded loop technique was first studied [Cull87a]. The details of this experiment are in Appendix C.7.

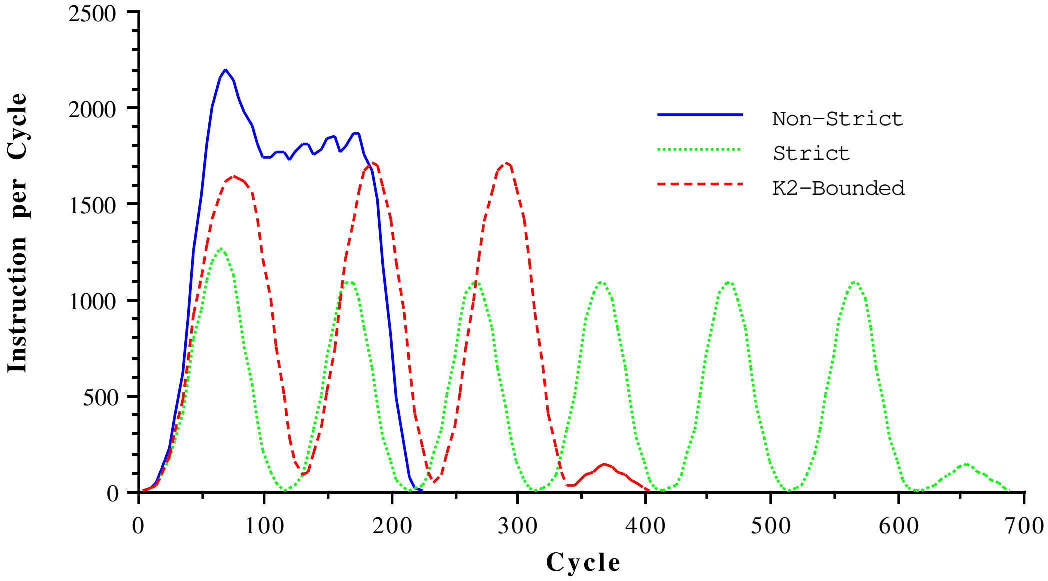


Figure 6.5: Ideal parallelism in Shallow.

6.7.1. Results

Figure 6.6. shows the improvement in store use achieved using these techniques for Shallow.

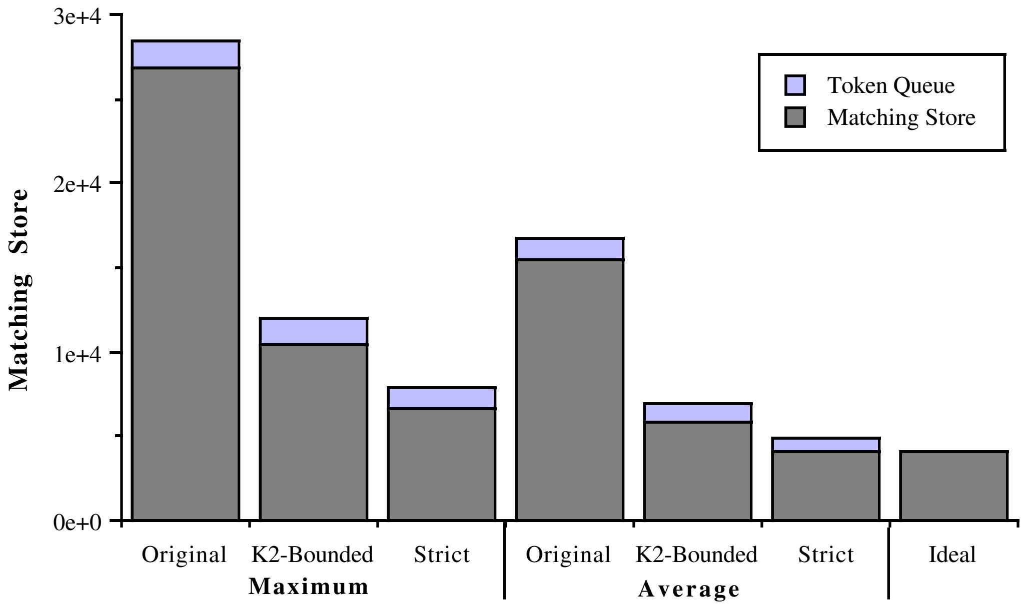


Figure 6.6: Reduction in store use due to software throttling.

The execution times for these tests show little variation. The original version runs in 0.96 ms, the K2-Bounded in 0.98 ms, and the strict in 1.00 ms. The effect would be even less noticeable with larger problems, but more pronounced on multiprocessor systems.

6.7.2. Conclusions

These software techniques are extremely effective at store management, for this problem. Because of the assembly language programming effort required, only Shallow was rewritten, however, the K-Bounded technique has proved successful in a variety of applications [Cull87a].

These techniques do create sequential bottle-necks in the computation, and, on larger systems, might reduce performance more than in this single processor experiment. Judgement is required in deciding which functions, at what level in the program, should be made strict, or in deciding the value of K. As a result, these techniques are not fully general, and control of applications using them, requires user input to the control process.

It should also be noted that these techniques are static and, therefore, are not likely to be as effective at controlling workload as dynamic schemes [Gurd92a].

6.8. Hardware Throttling

An "expected order of execution" can be used to sort tokens as they pass through the token queue, providing an effective throttle mechanism.

Throttles have three basic forms, static, locally dynamic, and globally dynamic. The scheme employed here is locally dynamic. This technique assumes that, for a particular program, there exists an "expected order of execution" within the tag space, for example, deepest recursive depth first and largest index next. This is not dissimilar to the strategy used to issue ANs in the Manchester machine [Teo91a, Gurd92a, Rugg87a].

As tokens enter the token queue, they are sorted according to the fields of their tags. This sort need not be exact, but can be a partial sort. This experiment assumes an exact sort.

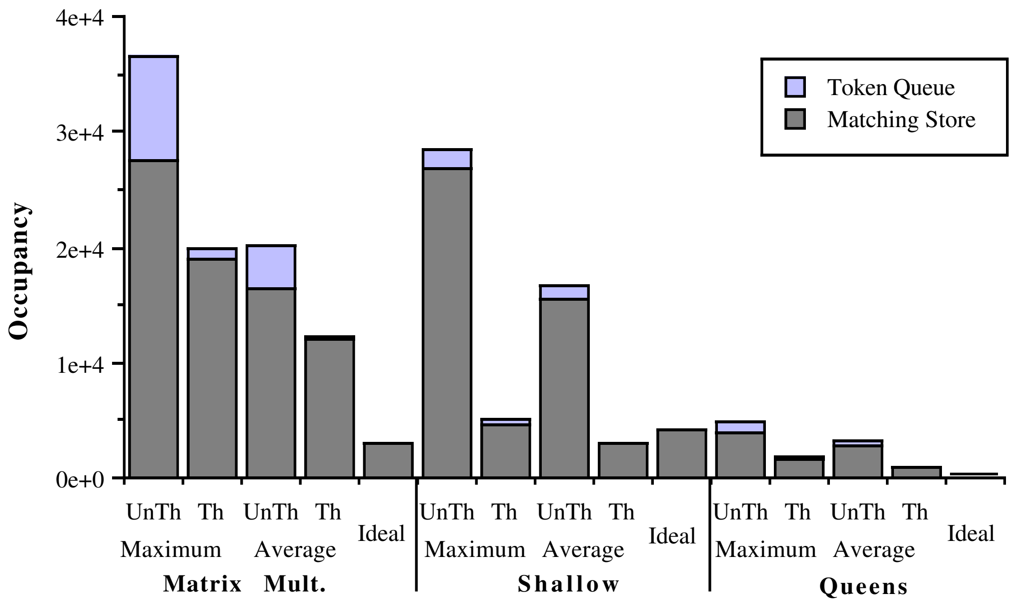
Since this scheme can only act on those tokens that are actually in the token queue, it is indirectly dependent on workload on the local processor.[71] Thus, it is a locally dynamic scheme. This experiment investigates the extent to which this approach can reduce store use without reducing performance in a uni-processor. If throttling is effective, the number of tokens processed locally should increase. This would, in an unsaturated system, improve performance as well. The details of this experiment are in Appendix C.8.

6.8.1. Results

The following table gives the execution times of the throttled and un-throttled cases of the three test programs. Times are in milliseconds and the locality is reported in percent of tokens processed locally.

	Un-throttled		Throttled	
Program	Time	% Local	Time	% Local
Mm	0.8676	60.0%	0.8532	64.4%
Queens	1.0415	68.1%	1.0366	94.4%
Shallow	0.9599	49.5%	0.9413	63.5%

Figure 6.7. shows the reduction in maximum and average store use as throttling is introduced. The ideal store use, based on a sequential algorithm, is included for comparison.



71 It has been found that token queue length is an excellent measure of system workload [Gurd92a].

Figure 6.7: Reduction in store use due to hardware throttling.

6.8.2. Conclusions

This simple throttling technique is remarkably effective at controlling the level of store use. This is particularly true with respect to the token queue length. The case of Shallow deserves special note. The fact that the throttled version is more store efficient than the ideal implies that temporary arrays, required in their entirety in the ideal case, are partially consumed before they have been fully created.[72]

The effectiveness of a single sorting function on this variety of codes supports the hypothesis that an "expected order of computation" exists which is the same for several codes. There will be counter examples, but it is believed that they would be rare or that other mechanisms would complement their use, such as those noted in Experiment 6.7.

In later experiments, hardware throttling is not used. Ideally, any bits, from the tag, used for workload distribution and increasing locality (see Chapter 7.) should not be incorporated into the throttle sorting function. For the problem sizes which can be simulated, this would leave no spare bits on which to sort tokens.

6.9. Matching Store Banking Study

Choose the number of banks in a matching store.

The number of banks in any memory system is determined by the rate at which the memory as a whole needs to provide data and the speed of the component technology. As was noted in Chapter 3.5.6, the timing parameters are based on a rough guide. Given these figures as fixed, the number of banks can be determined experimentally. The details of this experiment are in Appendix C.9.

---

72 A less naive sequential implementation could doubtless reduce these arrays to a small number of scalars, but this would require a rewrite of the program in a less coherent form.

6.9.1. Results

Figure 6.8. shows the execution time of Shallow for varying numbers of matching store banks in all three machine configurations. Similar results were achieved for Queens and Mm, see Appendix C.9.

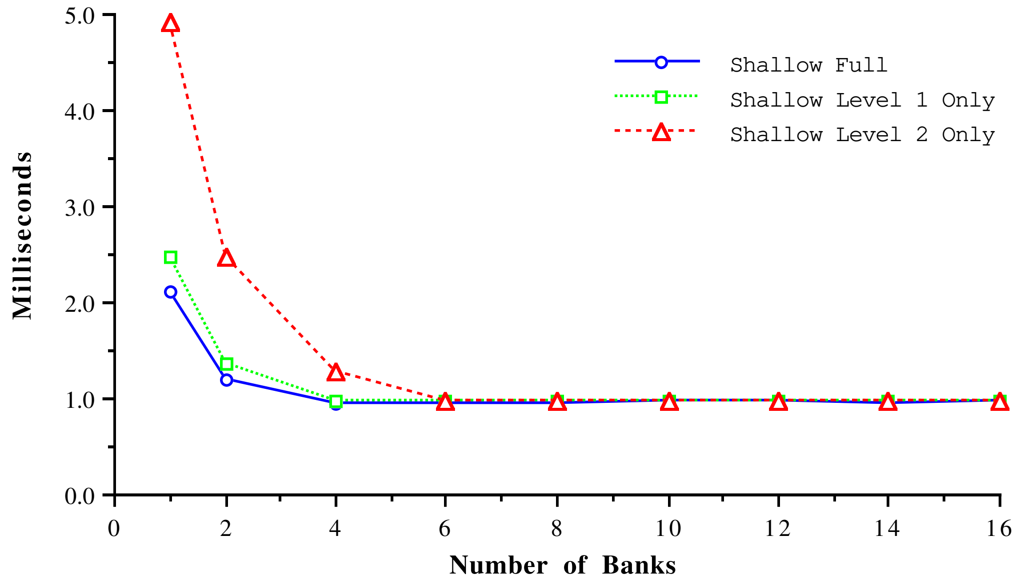


Figure 6.8: The effect of varying the number of MS banks on Shallow.

6.9.2. Conclusions

Clearly there is an optimum somewhere between 4 and 8 banks in a matching store. The value of 8 is selected, since it is likely to prove effective even in the absence of poor locality in the computation. Note that, since tokens not matching in level 1 must, in a realistic situation, always be forwarded to level 2, this is not an overly conservative design parameter. This is not a major technological issue, since 8 banks would be straightforward to implement.

6.10. Matching Store Cache Study

The random migration of tokens from Level 1 to Level 2, in the matching store, is an effective, cache-like mechanism for exploiting temporal locality.

Since scalability is dependent on the ability of the architecture to exploit a memory hierarchy, the effectiveness of the multi-level matching store must be examined. All the experimental applications, which can be reasonably simulated, fit within the default size of the Level 1 store. Therefore, in order to assess the effectiveness of the migrating mechanism, exceptional circumstances must be created within the system. This is accomplished by reducing the size of the Level 1 store to suit each application. The details of this experiment are in Appendix C.10.

6.10.1. Results

The execution times for all programs, as expected, showed little or no change as the threshold parameter varied. The results for caching efficiency, as a function of the threshold parameter, are presented in figure 6.9.

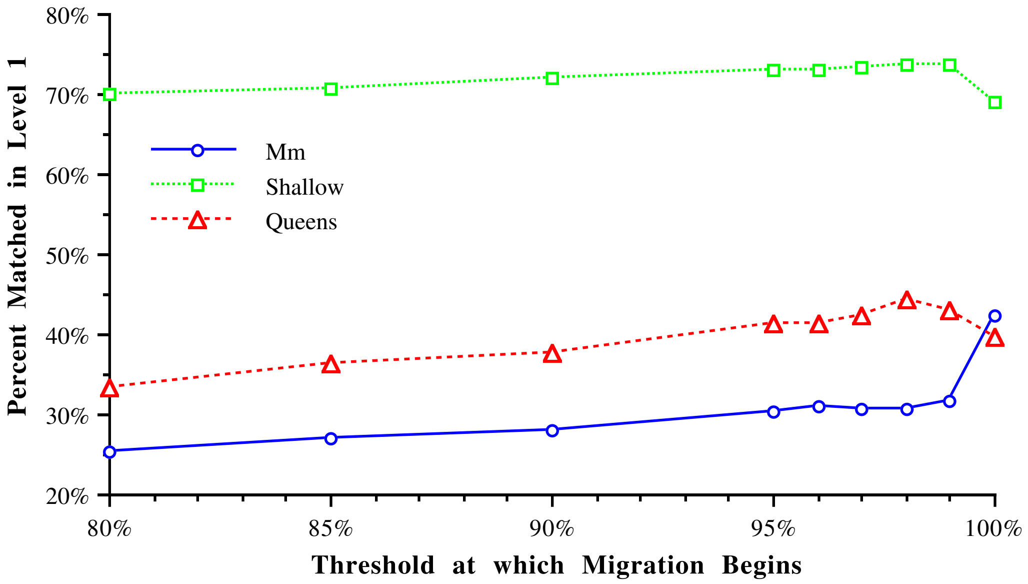


Figure 6.9: Migration threshold and locality in the two level matching store.

6.10.2. Conclusions

The effectiveness of this scheme, as measured by the percentage of main matching store matches occurring at Level 1, shows more dependence on the program than on the threshold parameter. This is due to the fact that, in order to activate the second level, the first level must be small. Therefore, the size of the Level 1 store plays a major role, and the absolute percentage varies significantly with application.

It might seem surprising that matrix multiply shows poor caching behaviour. However, implementations of matrix multiply, which exploit caches, are carefully coded and optimized to the architecture. This is not the case here. Recall from Experiment 6.2. that matrix multiply was the poorest user of the SSMS. The poor results for Queens are likely to be a result of the small size of the Level 1 store used in this experiment, 128 tokens per bank. Recall that the SSMS can hold 512 tokens and that Queens performs approximately 50% of all matches in the SSMS. Queens is, therefore, cache efficient, in spite of the results above. Only Shallow exhibits effective use of the second and third levels of the caching hierarchy. Although the threshold parameter still has relatively little effect.

Given that there is usually space in the Level 1 store, the second token, in a pair of tokens matched in Level 2, has almost always been stored in Level 1. The result is that, for virtually every Level 2 match, a clear action must be performed. Therefore, for every Level 2 match, Level 1 is searched three times and Level 2 twice. The minimum number of searches is two. A redesign of this mechanism is probably necessary. See Chapter 8 for a summary of these possibilities.

In the experiments that follow, the migration parameter is set at 99%. However, given the size of Level 1 compared to the applications used, Level 2 is unlikely to be accessed.

6.11. Summary

As noted at the start of this chapter, the primary design goal here is to develop locality mechanisms. With the exception of the migration experiment, the locality mechanisms in SDFA are effective. Even in this case, substantial numbers of tokens use the hierarchy effectively. It was only that the threshold parameter has little effect on the level of use.

The potential for developing effective throttling mechanisms for SDFA is apparent from Sections 6.7. and 6.8. Although no definitive solutions are obvious, possible directions for further research into throttle mechanisms are discussed in Chapter 8.3.3.

The next chapter examines multiprocessor SDFA configurations.

Chapter 7
Analysis of
Multiprocessor SDFA Systems

In this final experimental chapter, multiprocessor SDFA systems are studied. The preceding chapter presented the effectiveness of the temporal locality mechanisms in SDFA. In order for a multiprocessor to be scalable, it must also exploit spatial locality effectively. The first experiment in this chapter addresses this issue. Following this, the ability of the SDFA system to balance workload is assessed. Before examining the scalability of a SDFA system the packet switch design parameters are determined. The chapter concludes with an investigation into the peak performance of a small SDFA multiprocessor.

7.1. Spatial Locality in a Multiprocessor

The locality factor in the processor hashing function extracts spatial locality from applications, which can then be exploited by the architecture.

The experiments dealing with the various caches in the system investigated the effectiveness of the SDFA temporal locality mechanisms. This experiment deals with the issue of spatial locality in a multiprocessor SDFA system. As was noted in Chapter 5.2, the basic mechanism in SDFA is effective. This experiment focuses on how adjustments to this mechanism effect programs.

Spatial locality is exploited in SDFA through the hashing function used to determine to which processor a token is to be sent. One of the factors in this function is a *locality factor*. It represents how many of the low order bits in the index fields are not used in this function. This insures that most tokens with neighbouring indices are processed on the same processor. This simple mechanism is exploited naturally in many applications, either as a result of template structures, as in finite difference schemes, or algorithmic techniques, such as unit stride vector processing. In SDFA, these structures and techniques can be exploited in several dimensions rather than just the dimension corresponding to the linear organization of conventional memories. The details of this experiment are in Appendix C.11.

7.1.1. Results

Figure 7.1. shows the percentage of all tokens created which are processed locally for the three test programs on a 16 processor system.

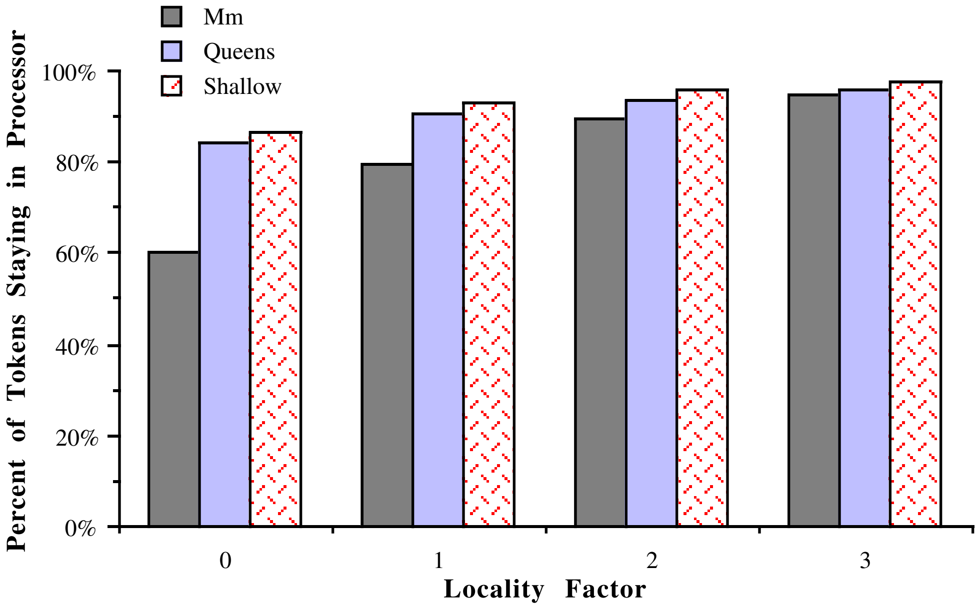


Figure 7.1: Spatial locality in a SDFA multiprocessor.

7.1.2. Conclusions

Two things are clear. First, locality is high in these applications without the locality factor. Second, the locality factor does increase the locality noticeably. With larger problems, the locality factor could be increased beyond three, and would, therefore, be even more effective.

7.2. Workload Distribution in a Multiprocessor

The processor hashing function in SDFA provides an effective mechanism for controlling workload distribution in a multiprocessor.

The constraints of a simulation environment are relevant in this experiment. The mechanisms for workload distribution (load balancing) in SDFA, and Data-Flow in general, are based on the assumption that as problem size increases the distribution of workload becomes more uniform. This is due to the randomizing nature of the hashing functions. However, this is frequently at a cost for smaller problems. This is the case for SDFA.

There are two types of load imbalance in parallel applications, static and dynamic. Static imbalance results when individual processors execute radically different numbers of instructions. Dynamic load balancing includes a temporal element. A system can be staticly load balanced and still perform poorly. Processors all execute the same number of instructions, but at different times. This experiment addresses static imbalance only.

Dynamic load balancing is a function of the application rather than the architecture. It will be addressed only indirectly as a part of later multiprocessor experiments.[73]

To validate the load balancing mechanism in SDFA it must be shown that, as problem size increases, the distribution of workload improves steadily. This must be done for problems exhibiting poor load balancing properties and all types of tagging behaviour. Queens and Gauss are used in this experiment plus Saxpy, which allows larger size matrices to be studied, see Appendix C.12. for details.

7.2.1. Results

Graphs indicating a significant drop in the extent of load imbalance are presented in Appendix C.12. The following is a summary of their meaning.

7.2.2. Conclusions

In the Queens case, the imbalance[74] drops steadily to about $6x10^{-4}$. At this point further simulation became impractical, but it seemed to be levelling off. Assuming uniform temporal balance, this would mean that the expected variation in execution time, between any two processors, would be about 0.06% of the total execution time.

In the Saxpy case, which was run for several *distribution factors*,[75] the smoothest effect is seen with a distribution factor of 1. In all cases, the load imbalance drops rapidly as the problem size increases. For a distribution factor of 1, the average imbalance levels off between $5x10^{-4}$ (worst case) and $3x10^{-5}$ (best case). In practical terms, this would mean that the expected variation in execution time, between any two processors, would lie between 0.05% and 0.003% of the total execution time.

The results for Gauss indicate that, like Saxpy, the imbalance falls off rapidly as the problem size increases. It is expected that the trend would continue as the problem size is increased, just as it does in Saxpy.

Better balance exists where the problem size exactly matches the machine, a feature common to most parallel systems. Programmers seeking exact balance can usually organize their program to meet this requirement.

7.3. Multiprocessor Network Study

Determine cost-effective design parameters for the packet switch in SDFA.

The important design trade off in a switch network is between bandwidth and cost. Data-Flow systems are capable of masking latency effectively, and therefore, the latency of the switch is less critical. Since the network, described in Chapter 3, is based on 2x2 cross bar switches, the number of "hops" in the switch is the base two logarithm of the number of processors in the system. The actual latency is a function of

---

73 A good survey of load balancing techniques, in fine-grained machines, can be found in [Sarg86b], albeit with respect to reduction architectures.

74 "Imbalance" is the standard deviation in the number of tokens created (with respect to the number of processors), normalized by the total number of tokens created. See Appendix C.12.2. for a full description.

75 The *distribution factor* effects the behaviour of the distribution hashing function and is described in Appendix C.12.1.

the bandwidth of the 2x2 switches. The design objective, therefore, is to determine how slow these 2x2 switching elements can be made without detrimentally impacting the overall performance of the system. The performance of the switch is determined by the *beat* of the pipeline which models it in the simulator, see Appendix C.13. for details.

7.3.1. Results

Figure 7.2. shows the execution time for Shallow as a function of the beat parameter in the switch. Similar results were obtained for Queens and Mm, see Appendix C.13.

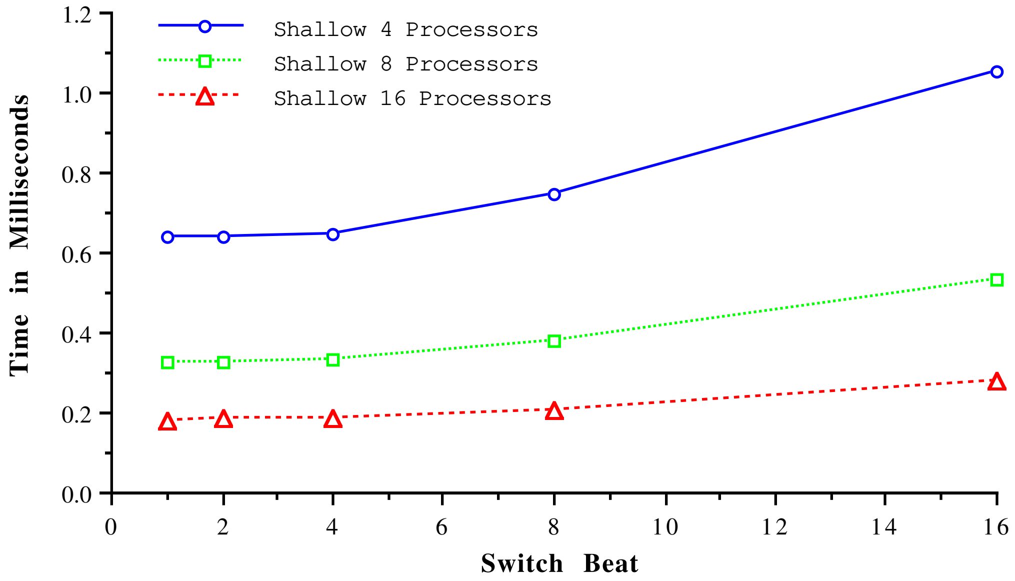


Figure 7.2: Shallow execution time as function of switch performance.

7.3.2. Conclusions

There is a clear point at which performance begins to degrade. This point is independent of the number of processors in the network, and only partially dependent on the application. The former is critical, since a system which required faster switches in larger configurations would not be scalable. The partial dependence on application is due to the level of locality exhibited by the various applications, Mm having the lowest locality in its current formulation, see Chapter 4.7.5.

Relatively small locality and distribution factors have been used to allow the use of small applications in these studies. Larger values of the locality factor would allow lower performance switching elements to be used. This is supported by the fact that those programs with greater locality don't start to degrade until larger values for the switch beat.

In all further experiments, a beat of two will be used for the switch.

7.4. Scaling Study

The SDFA system is scalable.

In a scaling study, based on simulation of relatively small applications, two things must be established, that the problems could run efficiently and that the system actually runs them efficiently. Since SDFA is designed to exploit locality, which detracts from the ability of a single application to fill larger systems, the choice of application size, locality factor, and distribution factor determine the maximum sized system the application will fill. Beyond this, some processors will receive no work at all. Therefore, problem sizes have been carefully selected to ensure that balanced computation is achieved, which (by the results of the previous experiment) would happen with larger problems in any case. See Appendix C.14. for an elaboration of this.

7.4.1. Results

Figure 7.3. contains the temporal performance results for Shallow (16x16, 32x32, and 64x64 all for 4 time steps) as a function of increasing number of processors. Similar results were obtained for Queens and Mm, see Appendix C.14.

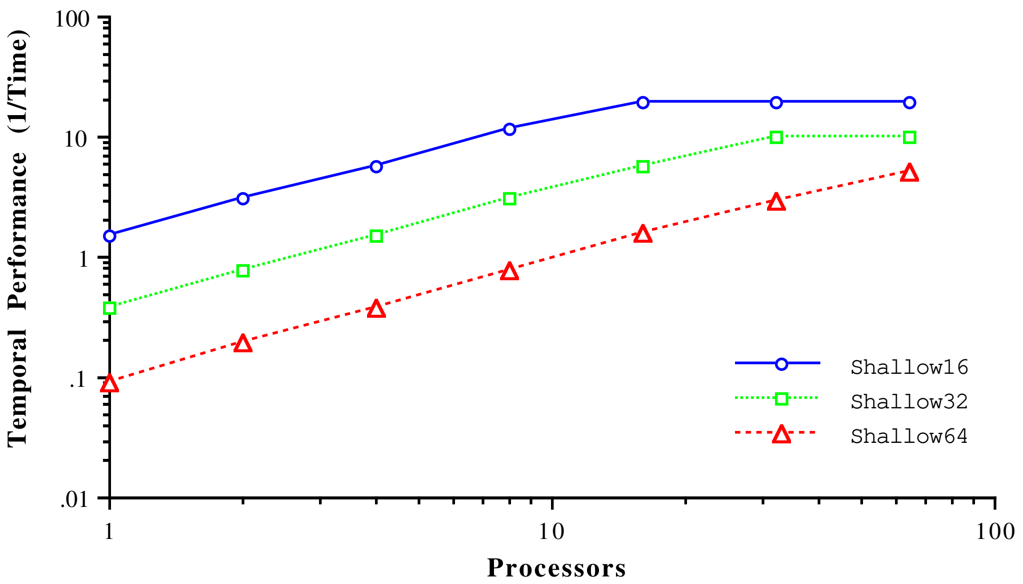


Figure 7.3: Scaling performance of Shallow.

7.4.2. Conclusions

For this collection of small problems, the SDFA system scales remarkably well. Since the locality factor limits the size of system that can execute a given problem, it is encouraging that the system scales, almost linearly, up to the point at which it can scale no further.

7.5. Toward Peak Performance

Determine the Effective Peak Floating Point Performance of SDFA.

In this final experiment, the achieved performance, measured in MFlop/s, for Shallow and Mm is compared to the theoretical peak. A 16 processor system is used, which has the theoretical peak performance of 2667 MFlop/s. The basic results are also compared to those that could be achieved if two simple enhancements were made to the instruction-set. Details of how these figures are arrived at are contained in Appendix C.15.

7.5.1. Results

The following table contains the performance results for Shallow and Mm running on 16 processor system. The time is in milliseconds.

Program	Time	MFlop/s	% of Peak
Mm	0.4253	1233	46.2%
Mm Unrolled	0.4067	1289	48.3%
Mm New Instr.	0.3506	1495	56.1%
Shallow	0.5986	1824	68.4%
Shallow Unrolled	0.5837	1870	70.1%
Shallow New Instr.	0.5260	2075	77.8%

7.5.2. Conclusions

In a system which can issue two instructions per cycle, only one of which can be a floating point operation, to achieve between 50% and 80% is a significant level of efficiency. This result is also significant in light of the fact that both of these problems were very small by high performance computing standards. The ability to produce high and scalable performance is a significant achievement.

7.6. Summary

The ability of the SDFA system to scale efficiently, even on small problems, is the most significant conclusion to be drawn from this collection of experiments. In the following chapter, the overall conclusions of this study are discussed along with proposals for further research.

Chapter 8
Conclusions and Further Research

A parallel computer architecture is scalable only if it distributes workload uniformly, reduces latency as much as possible, and hides the remaining latency. The SDFA system addresses these concerns, within a Data-Flow paradigm, by abandoning the notion of explicit state. In this final chapter, the thesis is reviewed, with respect to the primary issues facing the SDFA system; conclusions are drawn, with respect to the performance of the SDFA system; and lastly, directions for further research are explored.

8.1. Summary

Within the traditional framework of Data-Flow, there is a fundamental barrier which prevents the universal exploitation of locality, thus, limiting the scalability of these systems. In this section, the arguments supporting a Stateless Data-Flow Architecture, freed from this barrier, and the architecture itself are reviewed.

8.1.1. Review of the Fundamental Barrier

In Chapter 1.2.2, David Culler's analysis of the limits imposed on Data-Flow and multi-threaded systems leads to the conclusion that a scalable system must possess, and exploit, a memory hierarchy [Cull93a]. This conclusion is based on the need to increase local state in the processor to support an increasing number of active computations, which mask the increasing latency encountered as a system is enlarged.

With this analysis as background, in Chapter 2.2.3, an analysis of existing Data-Flow computational models reveals a fundamental barrier which prevents these systems from exploiting a memory hierarchy. Namely, the colourless nature of explicit state forces a shared memory model onto the architectures. Thus, Data-Flow, if it is to become scalable, must either abandon explicit state or become more explicit with respect to data structure management. Explicit alignment and distribution of data structures are possible with the use of language extensions such as HPF [Love93a, Love93b], but these, at present, require significant effort on the part of the programmer. The inclusion of these constructs in Data-Flow languages, such as SISAL [SISAL85] and Id [Nikh87a], would significantly detract from their elegance and implementation independence.[76]

In this study, explicit state is abandoned, which frees the system to extract naturally occurring locality and exploit a memory hierarchy.

8.1.2. Overview of the SDFA Architecture

The Stateless Data-Flow Architecture supports the basic model of Data-Flow, but without an explicit notion of state. This allows the system to be designed with a multiple level memory hierarchy, and execution control mechanisms, which can exploit

[76] It must be acknowledged that research is continuing that aims to automate the distribution and alignment process [O'Boy92a], but progress is slow, and commercially provided systems are ineffectual.

it unencumbered by a shared memory model. The SDFA system also incorporates other current architectural features which enhance performance.

A completely implicit mechanism for managing data structures is the primary, and unique, aspect of the SDFA system. Data structures, usually arrays, are represented as conventional tokens tagged with one or more index values. They are stored in the matching stores along with scalars. Thus, all data storage in the SDFA system is in the associative memory (direct or pseudo) of the matching stores. Associative memory, of either type, is more costly than conventional RAM; therefore, the development of a cost effective matching store would be essential to the feasibility of a SDFA system, see Section 8.3.8. below.

As with other Data-Flow systems, the SDFA system uses fine-grain parallelism to hide the latency associated with pipelining, memory access, communication, and synchronization. A fully pipelined approach to all components of the system makes it possible for the SDFA architecture to hide latency effectively. The primary criticism of fine-grained systems is the cost of synchronizing the many parallel activities. This cost is manifested in the design and production costs of matching stores. Although, the SDFA system has larger matching stores than other Data-Flow systems, the stateless model allows the matching process to be distributed throughout the system. Distributed synchronization (matching) in addition to distributed computation provides SDFA with the basis of a scalable system. Notice that, unlike other Data-Flow systems, the distribution of synchronization is not merely horizontal across processors, but vertical in the levels of the memory hierarchy as well.

The presence of these features is not enough to ensure effective computing; it must be possible, and preferably convenient, to exploit them. The SDFA system incorporates semi-automatic[77] mechanisms which exploit both horizontal and vertical distribution in the architecture. In SDFA, the semi-automatic, balanced distribution of workload across processors is accomplished by hashing the tag to generate a processor number. The particular bank number within the matching store is, likewise, computed by hashing.

The vertical distribution of synchronization requires the exploitation of temporal and spatial locality. The locality features of SDFA (local by-pass and SSMS in the processing element and multi-level matching stores) provide support for the locality inherent in Data-Flow programs. Throttle mechanisms (both software and hardware) can increase the locality in programs.

The SDFA system, also, incorporates the principles of super scalar processor design and RISC instruction-set design. The former allows the inclusion of multiple instruction issue in a single cycle. This, in turn, provides the processor with balanced computational resources which balance the local by-pass and SSMS. As with most RISC designs, careful partitioning of the instruction-set allows function units to be built with relatively short pipelines. Thus, the system has the ability to perform well on sequential computation.

Lastly, the SDFA processor is designed to be independent of network design, although only a single ported, packet switch is studied. SDFA systems could be built from simpler network technology, e.g. lower bandwidth but more ports into the network per processor.

---

77 Since the details of the hashing functions are under the control of the programmer, this process is not strictly automatic. However, for larger problems, it is likely that most problems will execute efficiently with a set of default values.

8.2. Conclusions

In Chapters 5, 6, and 7, experiments were performed which explored the ramifications of a Stateless Data-Flow Architecture. The issues investigated are pertinent to both Data-Flow and conventional parallel computing. The important conclusions, drawn from these experiments, are summarized in the following sections.

8.2.1. Comparison to MDFM

The SDFA system has only a small price to pay for being stateless, when compared to traditional Data-Flow systems, such as the Manchester Data-Flow Machine [Gurd87b]. There is a marked increase in the number of tokens required to solve a given problem on SDFA, when compared to MDFM. However, this increase is a constant factor in complexity equation and <u>not</u> an increase in the underlying complexity. It is possible that these constants could be improved through compiler techniques such as loop unrolling.

Also, as noted in Chapter 5.1, a large fraction of these extra tokens can be accounted for by the simplicity of the SDFA instruction-set, which is restricted in its design requirements. Some of these requirements could be relaxed without impacting the complexity of the hardware significantly, see Section 8.3.6. below.

In the second comparison with the MDFM, Chapter 5.2, the workload distribution mechanisms of the two systems are studied. Both systems distribute workload evenly, although SDFA has better mechanisms for extracting locality from the program. Both temporal and spatial locality, inherent in applications, is extracted transparently by the tagging and hashing mechanisms of SDFA. The result is substantially less network traffic. In the experiment pertaining to spatial locality (Chapter 7.1), over 80% of tokens are processed locally, even on small problems. In order to distribute small problems well, in a multiprocessor, each processor works on only a small subset of the problem domain. Thus, the compute-to-communicate ratio is relatively small. As problem size increases, this ratio will increase; therefore, local computation will improve steadily with problem size. This fact contributes substantially to the overall success of SDFA.

8.2.2. Temporal Locality

Temporal locality in SDFA programs is exploited by the memory hierarchy in the form of a local by-pass queue and a Short Stay Matching Store (SSMS) in the processing element. In the experiments performed in Chapters 6.1 and 6.2, these two memories act as caches and increase the number of tokens produced and consumed within the processing element from none to as much as 68%. This facility is necessary to support the multiple issue of the super scalar instruction fetch strategy, since only one instruction per cycle can be supported by the external token queue and matching store.

It is unlikely that dynamic throttling techniques could improve this substantially. This type of locality is fine-grained and, therefore, difficult to manage effectively. However, careful code generation could improve the amount of computation performed locally, see Section 8.3.1. below.

8.2.3. Super Scalar

SDFA, because of the fine-grain nature of Data-Flow computation and its locality mechanisms, is able to use multiple instruction issue more effectively than conventional super scalar systems. In conventional systems, multiple issue instructions must either be scheduled by the hardware dynamically, as in the T9000 [Inmo91a], or staticly by the compiler, as in the KSR [KSR91a]. Both techniques require parallelism to be extracted from a single thread of computation. This is a relatively small pool of potential parallelism when compared to that of the fine-grain approach in SDFA. Both of the instruction fetch pipelines are busy almost continuously in the experiments in Chapter 6.4.

The same argument applies to scheduling the function unit pipelines. In SDFA, as in all Data-Flow systems, the independence of instructions guarantees that there are no pipeline dependences between successive instructions. Parallelism is used similarly in some multi-threaded systems [Dene81a, Smit90a].

An additional advantage, provided by the lack of inter pipeline dependencies, is that the design of the control logic in the processing element is simplified. There are no register clash, branch quashing, or wave-off conditions possible. The combination of all these features allow SDFA to be designed assuming a short cycle time.

8.2.4. Throttling

Throttling in Data-Flow systems comes in three basic forms, static, locally dynamic, and globally dynamic. Only the first two types are studied in Chapters 6.7 and 6.8 respectively. The static technique requires additional instructions which impose varying degrees of strictness on the computation, namely fully strict and K-2 Bounded. In both cases, store use is effectively controlled. It is clear, however, that parallelism is lost in these approaches. This has little, or no, effect on the uni-processor performance. In multiprocessor configurations, it is possible that sequential bottle-necks could form. Increasing the value of K, in a general K-Bounded scheme, would allow a balance to be struck between store use and parallelism. However, the selection of K is not yet an automated process, and directives from programmers are sometimes required.

Dynamic schemes require less input from the programmer. Using some mechanism to measure the workload on the system, dynamic schemes attempt to increase or decrease the workload based on the information provided by this mechanism. This measurement may represent a local or global indication of workload. Only local information is provided by the SDFA mechanism studied in Chapter 6.8. Recall that the technique sorts tokens according to an "expected-order-of-execution" based on the values of their tags. Since the sort can only be performed on those tokens actually in the token queue, it is sensitive to the load on the given processor. Substantial drops in store use were observed, particularly in the token queue. The effectiveness of this technique, in reducing the length of the token queue, lends support to the assumption that the length of the token queue is an effective measure of system workload. Although dynamic, this technique is strictly local. The possibility of global techniques is discussed in Section 8.3.3.

8.2.5. Multi-Level Matching Store

The multiple level matching store is the one aspect of the SDFA design that does not perform to expectations. The mechanism designed to vary effectiveness of the migration process, namely the threshold at which migration begins, has little effect, and the migration strategy itself is fraught with difficulty. This is partially due to the success of the banking strategy. Uniform distribution of tokens across many banks means that only a small number of banks are required. Thus, the banks are able to keep the number of tokens in Level 1 below its threshold; therefore, tokens matched in Level 2 are usually stored in Level 1 and require clearing. This creates the need for frequent, redundant searches of the matching store, particularly Level 1, see Chapter 6.10. In Section 8.3.4. below, alternative strategies are outlined.

Although, multiple banks would allow a SDFA system to be built, which performs well, even in the absence of a multi-level matching store, a cost effective implementation is likely to require a multiple level matching store.

Implementation issues aside, the effectiveness of multiple level matching stores requires that temporal locality exist at the right granularity. Temporal locality at the fine-grained level is clearly present in SDFA programs, as seen in the success of the local by-pass queue and the SSMS. In conventional systems, this type of variable granularity in locality is not always present and must be created by software techniques. The best examples of these techniques are the multiple levels of the BLAS [Laws79a].

There is some indication that this type of locality does exist "naturally" in Data-Flow programs. Experimental studies, such as [Abra91a], and theoretical analyses, such as [Gott92a], indicate that locality exists at several grain sizes.

8.2.6. Latency Hiding

Although a feature of all Data-Flow systems, the ability of the SDFA system to hide latency, in addition to reducing it, must be noted. In the SDFA uni-processor studies, both access latency and synchronization latency are hidden effectively. This is best measured in the instruction issue rate. The percentage of all possible instruction fetches (two per processor per cycle) occurring for the various test programs is 95% for Mm, 88% for Queens, and 90% for Shallow.[78] In multiprocessor systems, latency hiding contributes substantially to the scalability of the system, see Section 8.2.8. below. The unique aspect of the SDFA latency hiding behaviour is that the synchronization, required to support it, is distributed.

8.2.7. Load Balancing

The load balancing technique used in SDFA is static and based on a hashing function of the tag. This technique, although not universal,[79] is effective at providing a good balance in workload across processors. This success is apparent both in terms of the total number of instructions executed and when they execute. By virtue of its randomizing behaviour, this technique improves as problem size increases; thus, even irregular problems (e.g. Queens and Gauss) balance well for reasonably large size cases.

The unique feature that SDFA has, which offers it more flexibility than other Data-Flow systems, is the ability to use several dimensions in the index space when computing the processor number. This allows SDFA to perform distribution in several dimensions.

---

[78] These results were taken from the largest, uni-processor simulations performed as part of the scaling experiment, Chapter 7.4.

[79] Exceptions always exist to static mechanisms, even if highly contrived.

8.2.8. Scaling

The single, most important conclusion of this study is that the development of a scalable parallel machine, based on the Data-Flow model, is a genuine possibility. The applications, run in this study, are small by super-computing standards. Yet, the level of performance exhibited, in terms of speed of computation, scalability, and number of processors, is of a standard expected by the super computing community.

In particular, the raw performance of a SDFA uni-processor system running Shallow (32x32 grid) was 2075 MFlop/s, see Chapter 7.5. A highly optimized version of Shallow (32x32 grid),[80] running on uni-processor KSR-1 system, achieved 13.8 MFlop/s. Assuming ideal scaling, a 16 processor KSR could achieve 220 MFlop/s.[81] If the SDFA clock is re-scaled to match the KSR, the performance of the 16 processor SDFA system would be 249 MFlop/s. This size problem is too small to parallelize effectively on the KSR-1. The performance for larger problems, although scaling well, is extremely sensitive to cache use. The SDFA system, however, scales well up to 32 processors for this size problem, and there is every indication that this trend continues with larger problems.[82]

8.3. Further Research

Understanding complex systems, such as computer architectures, is a continuing process. At no point does understanding become complete, since there are always further questions to ask. This study has focused on many, but not all, aspects of a new architecture. In this section, various areas for future research are explored, namely compiler technology for SDFA, large scale simulation of "real" applications, throttle mechanisms for SDFA, improved approaches to the matching store design, alternative network designs, instruction-set enhancements, the possible addition of limited explicit state, and VLSI associative memory.

8.3.1. Compiler Design for SDFA

The single most important issue, not explored in detail, is applicability of modern compiler techniques to SDFA. From the experiments in Chapter 5, there seem to be mechanisms which allow SDFA to execute efficiently, compared to other Data-Flow systems. Whether these techniques can be applied in general by a compiler is an area for further research. The most direct approach is to build on the work already done using SISAL or Id. SISAL is preferable since it has no explicitly stored data structures. Aside from the general questions of efficient code generation, there are several compilation issues which require special attention in a SDFA compilation system.

Since SDFA stores all data structures in the matching stores, elimination of unnecessary copies of arrays is an important optimization. Research, currently under way, which

---

80 This version of Shallow was developed by the author and George Mozdzynski of KSR. It is optimized for both parallel processing and effective cache use. The size 32x32 case is the best possible size for a single cell KSR-1 processor, and all KSR tests were run for several thousand time steps in an effort to amortise system overheads. The number of MFlop performed by each system was computed from the same formula.

81 In fact, the 16 processor run was slower than the uni-processor because the problem is too small to parallelize on the KSR.

82 It should be noted that, although the codes for both systems, were carefully optimized for the machine in question and all care was taken in preparing this data, comparisons of this type are fraught with vagaries.

aims to reduce the number of copies of an array, required by a given computation [Cann88a], should apply equally well to the SDFA system. This assumption is based on the commonality of the language base and similarity of the basic execution models. In fact, the increased simplicity of the SDFA model could make such techniques even more effective.

There are several techniques, which could be applied by compilers, that could increase locality in computations, such as loop unrolling and loop blocking. A more specialized technique, requiring additional hardware support, would allow the compiler to flag particular results as "non-local". These tokens would not be processed by the SSMS, under the assumption that their match was already in the main matching store, coming from another processor, or not to be created for some time. This technique would allow tokens with higher locality to use the available space in the SSMS.

Because of the uniqueness of the SDFA approach, there is little experience with the architecture generally. This means that the instruction-set could not be designed with compilation techniques in mind. Likewise, hardware support for compilation could not be incorporated into the design of SDFA. A re-examination of the architecture and instruction-set, in light of knowledge gained in this study, should create possibilities for significant improvements in the SDFA system, with respect to compilation.

The existence of a compiler for SDFA would also permit further study into larger applications.

8.3.2. Large Scale Simulation

Since all SDFA programs are written in the SDFA assembly language, large application programs could not be developed easily. The largest program used, Shallow, is only a shell of a single layer, atmospheric model. Simulations of substantial applications would be a necessary part of further research into SDFA.

The design of the SDFA simulation system, see Chapter 4 and Appendix B, included the possibility of parallel simulation, although this was not implemented. If the simulation software were ported to a parallel system, large scale simulations could be performed. The ability of the system to scale and the effectiveness of the current throttle mechanism could then be explored in detail. Likewise, the ability of the system to perform in a multi-user environment could be investigated.[83]

8.3.3. Throttle Mechanisms

As just noted, the effectiveness of the sorting throttle on multiprocessor systems needs to be ascertained. It was effective in reducing store use in a uni-processor, but, for reasons noted in Chapter 6.8, it was not used in the multiprocessor experiments. A more detailed study of this mechanism should include multiprocessor simulations, large application programs, alternative sorting functions, and the design of a hardware mechanism for performing the sort.

The local, dynamic throttle used in SDFA could be expanded to include the processing element. When the decision to route a token, either locally or externally, is made, its tag could be compared with that of the token at the head of the processing element input queue. In this way the processing element could become part of the same sorting function.

83 A pilot project in this area, using an early SDFA design and the MDFM, is reported in [Snel92a].

Global control of parallel machines is one of the major research topics currently being explored by researchers in Data-Flow and conventional parallel computing. In conventional computing, these techniques usually rely on the run-time system or operating system to regulate workload.[84] They are coarse grained and not suited to Data-Flow. The most effective fine-grain technique is that used by the MDFM [Rugg87a]. Recent improvements to this basic scheme [Gurd92a] effectively control individual programs and job mixes [Snel92a].

The incorporation of a global, dynamic scheme into SDFA is not straightforward. The simplest approach is to use the ProcessTag field of the SDFA token like an Activation Name and replicate the MDFM throttle. This would require a compiler or a rewrite of the current SDFA programs. Alternatively, the allocation of new RecTagStack values could be controlled explicitly. In both cases, a throttle unit would need to be incorporated into the system and information on global workload extracted. The latter could come from the lengths of the token queues, as it does in the MDFM. This technique would doubtless be as effective on SDFA as on the MDFM; however, the basic mechanism is centralized and contrary to the distributed nature of SDFA.

Further research into throttling mechanisms, both existing and wholly new, would constitute a major part of continuing work on SDFA.

8.3.4. Effective Caching in the Matching Stores

As noted in Section 8.2.5. above, the two level matching store is not effective at exploiting temporal locality. Assuming that adequate locality exists, alternative mechanisms for exploiting it need to be sought. It is only through the ability to use cheaper memory technology, for the larger components of the matching stores, that a cost effective SDFA system could be developed. Several possibilities exist.

The primary weakness of the current scheme is the fact that the two levels of the matching store must be searched repeatedly and redundantly. The number of searches could be reduced by maintaining a short-list of forwarded tokens in Level 1. This would reduce the number of searches in Level 1 to a minimum and improve the scheme generally. The approach does, however, increase the complexity of the matching store substantially.

The simplest solution would be to increase the number of banks in the matching store significantly, and use some form of cost effective, pseudo associative memory in each bank. If the SSMS cannot satisfy enough matches, sequential bottle-necks are likely to be a problem, due to the long pipeline of such a matching store design. Also, although the tag hashing scheme works well with the number of banks studied herein, it remains to be determined if the scheme could generalize to support both a large number of banks and the hashing scheme internal to the individual pseudo associative memories.

Other approaches could also be considered and developed. In any case, the existence of an effective multi-level matching store is only the first step. Although there is some indication that locality exists naturally at all stages of a computation, its effective exploitation may continue to be difficult. If an efficient, multi-level matching store can be developed, SDFA is in no worse position than conventional distributed memory architectures, in which compilers and programmers apply extensive effort to organize programs such that they can exploit memory hierarchies.

---

[84] An aggressive, multi-level approach to controlling a large parallel system is reported in [Alve92a].

8.3.5. Alternative Network Designs

The network design used in this study is a simple, relatively high performance, packet switch. Further research into SDFA could build on the techniques explored by other distributed memory system designers. The basic difference in these approaches is that the communication time between processors is not uniform. These networks can exploit multiple levels of spatial locality, since processors have near neighbours and distant neighbours. Routing of messages is either performed by the main processors or by separate communication processors. In these approaches, there are usually several input and output ports on each processor, but with lower band-widths and longer latencies. Within this class of network, there are two basic approaches taken to routing messages, *worm hole* routing (a survey of these approaches can be found in [Ni93a]) and *packet switch* routing (an example of this approach can be found in [Pitt88a]).

8.3.6. Instruction-Set Enhancements

The fraction of total workload, resulting from the simplicity of the current instruction-set (see Chapter 5.1.), indicates that a few simple changes in the instruction-set architecture could produce substantial reductions in the number of tokens created in SDFA programs. A redesign of the instruction-set, in light of what has been learned, could provide both a leaner and more powerful instruction-set and, hence, system. The possibilities fall into three basic approaches, increasing the complexity of instructions, the addition of proliferate instructions, and the addition of new simple instructions.

The instruction-set complexity can be increased by combining the orthogonal aspects of the instruction-set into a collection on independent actions, which can be performed on different parts of a token. There are three basic actions performed on tokens which could be performed, more or less, independently: data filed computation, simple re-tagging, and conditional flow of control. This approach is partially emulated by the CIS facility used in Chapter 5.1. An integrated approach would produce better results than the ad hoc CIS facility. Not only could code be targeted to the facility, but other combinations of instructions could be included, e.g. simple recursive calls could follow other operations. (This is not possible using the CIS facility). None of these enhancements effect the basic restrictions placed on the design of the initial instruction-set, in particular they do not increase the number of result tokens beyond two.

Proliferate instructions, however, can increase the number of tokens produced as the result of a single instruction. The function of a proliferate instruction is to replicate a one value and tag it with different index fields. This is needed commonly in scientific computation to replicate scalars for computation with arrays. In the MDFM, this facility is provided by an instruction which executes for several cycles, creating one token per cycle [Böhm90a]. This approach is more appropriate to the SDFA system than other approaches.[85] However, the proliferate instruction ties up what ever function unit is used to execute it. This can create problems in controlling parallelism [Böhm90a]. Assuming that, in SDFA, proliferate instructions are performed by their own functions unit, the long running nature of proliferate instructions could block the instruction fetch, preventing execution of other instructions until the proliferate finished. One solution is to put a relatively long input queue on the proliferate function unit. Alternatively, the proliferate instruction could be interruptable.[86]

---

85 Alternatives to this approach exist. In the CSIRAC architecture, the matching store performs this function. A special matching function is provided that, when a scalar matches with a vector, the scalar is replicated to match the elements in the vector.

86 A mechanism similar to the following is envisioned. A proliferate, in SDFA, would require the value to be replicated and an Index Set Descriptor (ISD) describing the indices of the resulting array. Should the input queue to the proliferate function unit approach a full state, the currently executing proliferate instruction could stop, and reissue both input tokens to itself, with an appropriately

New basic instructions could also be investigated. The most likely area for further research would be the development of instructions which modify the index field of the tag. Bit reverse, multiply-add index arithmetic, and circular shift are obvious possibilities. Others could be developed which would provide support for higher dimensionality arrays and sparse matrix operations.

Although not strictly an instruction-set enhancement, the possibility of investigating explicit instruction scheduling should be noted. Since Data-Flow instructions are not necessarily planted in a linear sequence, part of the instruction could be planted as part of the address, e.g.the op-code, without reducing the effective size of the destination. Data-Flow computation usually has several executable instructions available for execution on each cycle. Using the encoded information, dynamic scheduling of instructions is possible at little extra cost and with more flexibility than conventional dynamic scheduling schemes, such as [Henn92a].

8.3.7. Limited Support for State

At some point it may be desirable to include some notion of explicit state in SDFA. There are three reasons why this might be desirable: 1) to provide support for run-time system services, such as ProcessTag management; 2) to provide support for non-deterministic computation; and 3) to provide storage for large constant data structures. The disadvantages of a global, shared state have been made clear; however, these cases might nonetheless prove useful. All three can be provided either by storage nodes supporting copy and exchange matching functions, or by separate, specialized "processors" attached to the network.[87] With respect to reasons 1 and 2 above, the size and frequency of access to these data structures would typically be small, and M-Structures would be investigated initially as a programming model [Bart91a]. In the third case, the data structures resemble persistent I-Structures [Arvi80a], which are created once, read frequently, and not destroyed until the end of the program. Access to these data structures would be more frequent, but read only, and therefore, parts of them could be copied to where they were needed using mechanisms similar to caching.

8.3.8. VLSI Associative Memory

This section began, at the highest level of abstraction in the SDFA system, with a discussion of the need for compilation software. At the bottom of the abstraction hierarchy lies the need for fast and cost effective associative memory. Further research into SDFA would require a careful look at the design of VLSI associative memory. Unlike conventional RAM, associative memory is either highly specialized (as in the facilities needed for SDFA) or highly complex (as that used in associative array processors [Hwan89a]). This means that an effective associative memory for SDFA is unlikely to be useful in other applications. Likewise, although commodity associative memory could perform the necessary functions, it is unlikely to be dense enough to be effective.[88] It, therefore, must be custom designed. Design and simulation of fast, dense associative memories would be an essential part of any further research into SDFA.

---

reduced ISD. Thus, the input queue would be partially emptied, and the executing proliferate would be again subject to the throttle mechanisms of the system. There are many details of this approach which can be explored.

87 In either case, data structures would need to be stored colourlessly.

88 Commodity parts could be used in SDFA for the main matching store but clearly not for the SSMS within the processing element.

8.4. Closing Comments

The Stateless Data-Flow Architecture, like all other architectures, is a compromise between design philosophy and implementation restrictions. SDFA is most successful in its ability to scale and exploit temporal and spatial locality. The use of contemporary, high performance, RISC processor techniques has meant that this success is not at a cost in raw performance. The promise in SDFA may lie in further development of the stateless approach or in attempts to improve existing approaches, such as distributing the cost of synchronization in multi-threaded systems.

As with many aspects of parallel processing, the problems encountered and lessons learned have first been encountered in Data-Flow systems.

Epilogue

SDFA took the leap of faith and followed the SDFA way. Whilst upon his journey, he endured many trials, but he did not go to hell. Yet, he must do much more to become a high priest.

Appendix A: Architectural Details

This appendix contains details of the SDFA architecture and explanations of some of the design decisions. It is divided into three main sections. The first contains details of the execution model, the second details on the physical architecture, and last, a listing of the definitive architecture configuration file used with the simulator.

A.1. Execution Model

A.1.1. Horizontal Instruction Word Formats

The SDFA instruction-set is designed to support the possibility of combining related operations into a single, horizontal instruction. In a single instruction, for example, the result of a floating multiply could be compared to a constant and routed to one, of two, distinct destinations.

Wide instruction word models have been explored in traditional computer architectures. The most extreme example is the Multi-Flow system.[89] Depending on configuration, this machine supports from 7 to 28 instruction fields. In order to achieve efficient, function unit utilization, the compiler performs speculative computation and plants additional code to "un-do" some computation, when the need arises. This compilation technique is called trace scheduling [Fish79a], and a description of the architecture can be found in [Fish88a]. Other examples of this approach are the Tera and Horizon architectures [Smit90a, Smit88a], with three instruction fields, and the Cyber+, with 4 instruction fields which control 10 function units. The same level of parallelism can be exploited by allowing the orthogonal aspects of the instruction-set to execute independently. This has been done by the author, and others, to improve the performance of sequential computations on the HEP [Kowa85a]. In particular, a separate program counter is assigned to the loading, computation, and storage functions of the code. An extension of this technique is exemplified in the ACRI architecture [ACRI92a]. In this system, the compiler plants code independently for the orthogonal domains of the instruction-set. These communicate through fixed length queues, rather than registers (HEP) or a matching store (Data-Flow).

A.1.2. Token Tagging and Hashing

Since the TagType (the tag field which indicates the format of other fields), may vary during the execution of a program, the technique for computing the TagType (based on the sizes of the other fields) should ensure that two tokens with the same tag values always have the same TagType. This is necessary to perform matching and distribution functions. The same should hold for how bits from each field are used as routing information in the network. Mathematically, the process for determining the TagType must be a function and not a more general relation.

The size of Destination field can vary depending on the size destination address. If the code for computationally intensive routines is planted in low memory, more tag space is

[89] The company has ceased operations.

available for other fields during the main body of computation, when, for example, tag space might be needed for array indexing. In fact the current SDFA assembler does this implicitly.

There are two hashing functions used in SDFA, one for determining to which bank of the matching store (MS), or the Short Stay Matching Store (SSMS), a token is to be sent, and one for determining on which processor an instruction is to execute. The MS and SSMS bank hashing functions are based on the index and destination fields of the token. This function distributes well in the interpreter, where the matching store has 256 banks.[90] The maximum number of banks, used in the simulator, is 16.

The processor hashing function is based on the index and RecTagStack fields. Whereas the bank hashing functions use all the bits of the respective fields, the processor hashing function can ignore the low order bits of each field and shift up the high order bits. These two actions, which are under the control of the ProcessTag, serve to increase locality and to distribute small computations more effectively. The effectiveness of these two actions is studied in Chapter 6.

A.1.3. Recursive Function Call Management

In this section, the algorithms for recursive call management are formalized, using the following definitions.

A *simple function call* is any call to a non-recursive function or to a primitive instruction within the system.

The *scope* of a function is the usual sense of scope, with the extension that all simple functions are transparent. That is, they are fully in-lined into their calling function.

An *initial function call* is any call to a recursive function from outside its scope.

A *recursive function call* is any call to a recursive function from within its own scope.

A *singly recursive function call* is a recursive function call which is unique within its own scope.

A *multiply recursive function call* is a recursive function call which is <u>not</u> unique within its own scope.

A *tail recursive function call* is a singly recursive function call such that the results of the deepest recursive call may be passed directly to the place where the function was initially called. This is the classical definition of tail recursion and is frequently called "iterative".

A *linear recursive function call* is a singly recursive function call, such that the results of a recursive call must be combined with other values before the function can return a result. This is the classical definition of linear recursion.

90 The large number of banks improve the performance of the interpreter itself.

A *function return* is the point in a function where the results of the function are returned to the point where the function was called. For simple functions, the results are sent to the unique point where the function was called. Function returns from recursive functions, however, may send their results to one, two, or more distinct places, depending on where the particular call was made.

A.1.3.1. Recursive Function Calls

i) The initial value of the RecTag is zero, and the initial value of the RecTagStack is "empty".

ii) At any initial function call, the current value of the RecTag is stacked onto the RecTagStack and the RecTag is set to zero.

iii) At any singly recursive function call, the RecTag is incremented.

iv) At any multiply recursive function call, the current value of the RecTag is stacked onto the RecTagStack and the RecTag is set to 1, 2, 3, etc. depending on which of the multiple calls is being made.

v) At any singly, linear recursive function return, if the current RecTag is zero, the previous RecTag is un-stacked, from the RecTagStack, and the token sent to the destination indicated by the initial call. Otherwise, the RecTag is decremented and the result sent to the destination, within the function body.

vi) At any tail recursive function return, the previous RecTag is un-stacked and the token sent to the destination indicated by the initial call.

vii) At any multiply recursive function return, if the current RecTag is zero, the previous RecTag is un-stacked and the token sent to the destination indicated by the initial call. Otherwise, it is sent to the destination, within the function body, determined by the current RecTag and the RecTag is un-stacked.

The process of stacking the RecTag can be done in several ways. The simplest way is to multiply the RecTagStack by an integer greater than the largest expected RecTag and add in the current RecTag. For multiply recursive functions, this multiplier is one greater than the number of recursive calls within the scope of the function. For singly recursive functions, it is any value larger than the maximum expected depth of recursive calls. In both cases, this multiplier is a literal value on one of the input ports to the call and return instructions. This approach wastes tag bits and is available as an option in the software..

A more complicated strategy involves a function unit which maintains a data structure for the actual RecTagStack and supplies unique values for the RecTagStack field in the tag. This requires careful management of work load distribution and should not create a bottle-neck within the system. That is, given several instances of the same current tag, as in the various arguments to a function, the same new tag must be generated for each, even when these tokens are arriving on different processors. This is the approach used in SDFA.

Figures A.1, A.2, and A.3 provide examples of tail recursion, linear recursion and multiple recursion.

```
function facl (n: integer returns integer)
if (n = 0) then
   1
else
   n * facl (n-1)
end if
end function
```

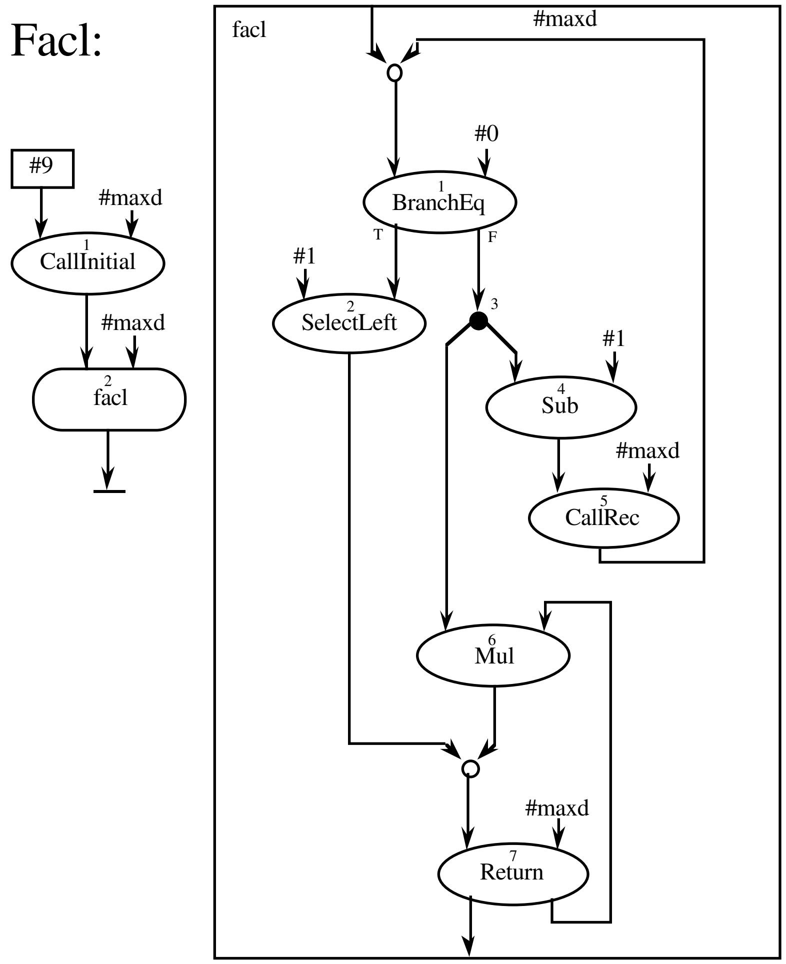


Figure A.1: Linear recursive factorial in SISAL and SDFA Data-Flow Graph.

```
function fact (n, acc: integer returns integer)
if (n = 0) then
   acc
else
   fact (n-1, acc*n)
end if
end function
```

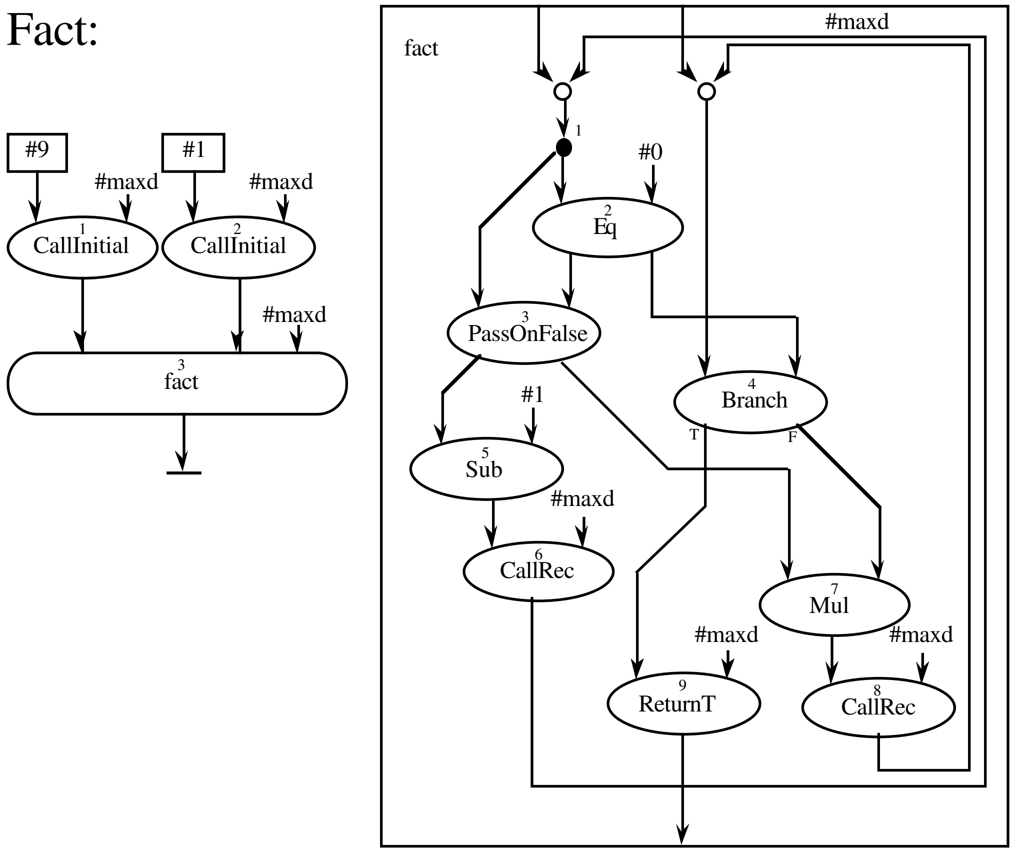


Figure A.2: Tail recursive factorial in SISAL and SDFA Data-Flow Graph.

```
function fib (n: integer returns integer)
if (n <= 1) then
   1
else
   fib (n-1) + fib (n-2)
end if
end function
```

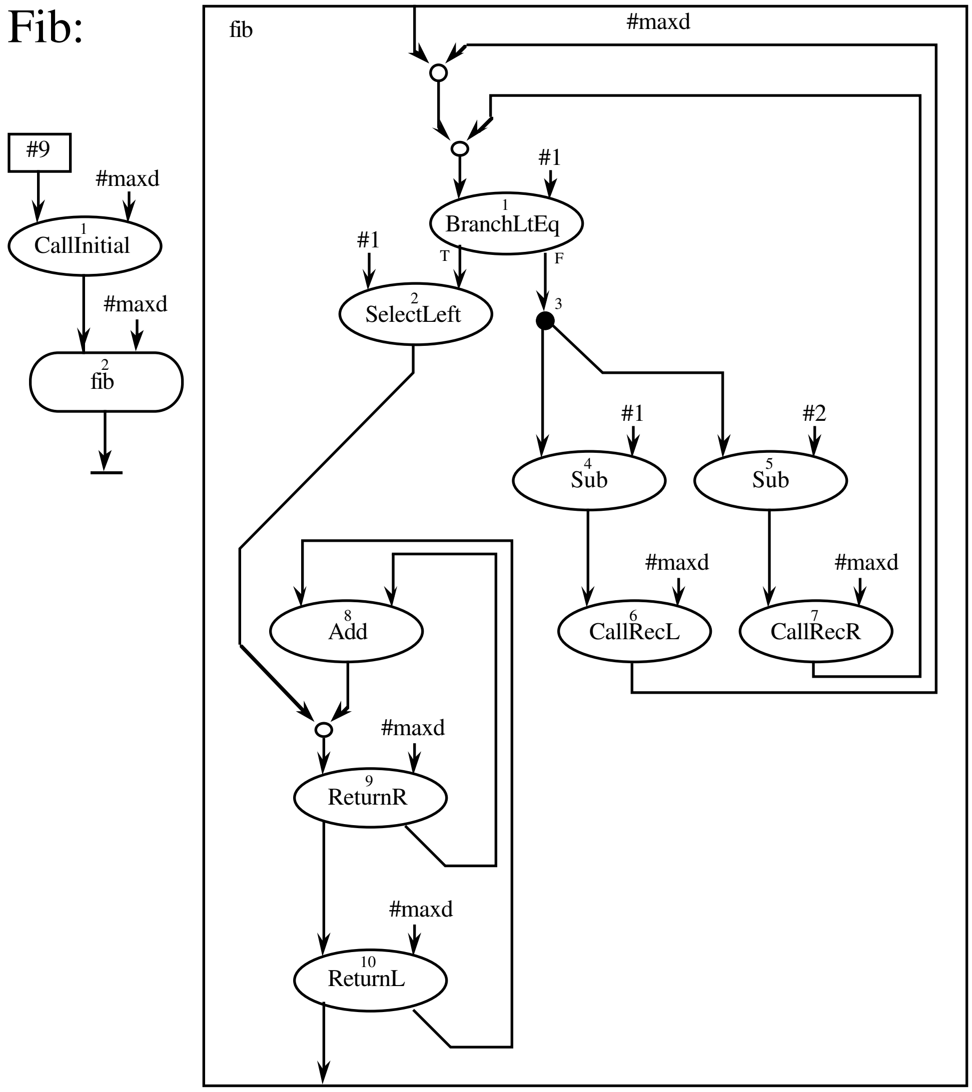


Figure A.3: Multiply recursive Fibonacci in SISAL and SDFA Data-Flow Graph.

A.1.4. The Instruction-set

The format of an instruction and a full definition of the instruction-set are included in this section.

A.1.4.1. Instruction Format

Instruction ::

Op-code	See below	The operation to be performed.
InL-token	Boolean	Input to left port is a token.
InR-token	Boolean	Input to right port is a token.
OutL_Match	Boolean	Indicates need to match at destination.
OutL_Port	(Left, Right)	Input port for token at destination node.
OutL_Dest	Destination	Address of destination node.
OutR_Match	Boolean	Indicates need to match at destination.
OutR_Port	(Left, Right)	Input port for token at destination node.
OutR_Dest	Destination	Address of destination node.
Literal	$[0..2^{64} -1]$	Real, integer, or boolean data value.

A.1.4.2. The Basic Instruction-set

The input values, vl and vr, may be tokens, literals, either tokens or literals, or null, as indicated below by a T, L, E, or N under the headings for vl and vr. The destination values, indicated dl and dr, either contain a destination or an indication that no token is to be sent from that port. Some instructions conditionally send a value out the left or right port . In the table below, $\perp$ is used to indicate that no value is sent on the given port. The tag function notation is as follows: a pair of angle brackets surrounds a list of changes to the various fields of the tag. Only those fields which are changed by the instruction are listed, and the abbreviations should be self-explanatory. The error types used below are N - no error, O - overflow, U - under-flow, and DZ - divide by zero.

Integer and Unsigned Arithmetic:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
Add	E	E	(vl+vr) mod 2^64	<>	dl,dr	N
Carry	E	E	(vl+vr) div 2^64	<>	dl,dr	N
Sub	E	E	(vl-vr) mod 2^64	<>	dl,dr	N
Mul	E	E	(vl*vr) mod 2^64	<>	dl,dr	N
MulHigh	E	E	(vl*vr) div 2^64	<>	dl,dr	N
Div	E	E	vl div vr	<>	dl,dr	DZ
Mod	E	E	vl mod vr	<>	dl,dr	DZ
Abs	T	N	$\lvert vl \rvert$	<>	dl,dr	N
Float	T	N	Float(vl)	<>	dl,dr	N

Floating Point Arithmetic:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
Fadd	E	E	vl+vr	<>	dl,dr	O
Fsub	E	E	vl-vr	<>	dl,dr	O
Fmul	E	E	vl*vr	<>	dl,dr	O,U
Fdiv	E	E	vl/vr	<>	dl,dr	O,U,DZ
Fmod	E	E	vl mod vr	<>	dl,dr	DZ
Fabs	T	N	$\lvert vl \rvert$	<>	dl,dr	N
Floor	T	N	$\lfloor vl \rfloor$ as integer	<>	dl,dr	O
Ceiling	T	N	$\lceil vl \rceil$ as integer	<>	dl,dr	O

Floating Point Transcendental Functions:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
Fsqrt	T	N	$\sqrt{vl}$	<>	dl,dr	O
Fexp	T	N	e^{vl}	<>	dl,dr	O
Fln	T	N	ln(vl)	<>	dl,dr	O
Fsin	T	N	sin(vl)	<>	dl,dr	N
Fcos	T	N	cos(vl)	<>	dl,dr	N
Fatn	T	N	$\tan^{-1}(vl)$	<>	dl,dr	O
Fpwr	E	E	vl^{vr}	<>	dl,dr	O

Integer and Floating Point Relational Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
Eq	E	E	vl = vr	<>	dl,dr	N
Neq	E	E	vl ≠vr	<>	dl,dr	N
Lt	E	E	vl < vr	<>	dl,dr	N
LtEq	E	E	vl ≤vr	<>	dl,dr	N
Gt	E	E	vl > vr	<>	dl,dr	N
GtEq	E	E	vl ≥ vr	<>	dl,dr	N

Unsigned Relational Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
UEq	E	E	vl = vr	<>	dl,dr	N
UNeq	E	E	vl ≠vr	<>	dl,dr	N
ULt	E	E	vl < vr	<>	dl,dr	N
ULtEq	E	E	vl ≤vr	<>	dl,dr	N
UGt	E	E	vl > vr	<>	dl,dr	N
UGtEq	E	E	vl ≥ vr	<>	dl,dr	N

Boolean Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
Or	E	E	vl ∨ vr	<>	dl,dr	N
And	E	E	vl ∧ vr	<>	dl,dr	N
Xor	E	E	vl xor vr	<>	dl,dr	N
Equiv	E	E	vl equiv vr	<>	dl,dr	N
Not	T	N	~vl	<>	dl,dr	N

Bit Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
ShiftL	E	E	vl sft L vr	<>	dl,dr	N
ShiftR	E	E	vl sft R vr	<>	dl,dr	N
ShiftC	E	E	vl sft R circ. vr	<>	dl,dr	N
ShiftA	E	E	vl sft R vr sign ext.	<>	dl,dr	N
Pop	T	N	No. of 1 bits in vl	<>	dl,dr	N

Special Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
Dup	T	N	vl	<>	dl,dr	N
Kill	T	N	–	–	⊥,⊥	N
Error	T	L	–	–	⊥,⊥	N
Output	T	E	vl out on channel vr	–	⊥,⊥	N
NoOp	T	N	vl	<>	dl,⊥	N
SelectLeft	E	E	vl	<>	dl,dr	N
SelectRight	E	E	vr	<>	dl,dr	N
SetPT	E	E	vl	<pt := vr>	dl,dr	O
GetPT	T	N	pt	<>	dl,dr	N

Control Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
PassOnTrue	E	E	vl	<>	if vr then dl,dr else ⊥,⊥	N
PassOnFalse	E	E	vl	<>	if ~vr then dl,dr else ⊥,⊥	N
Branch	E	E	vl	<>	if vr then dl,⊥ else ⊥,dr	N
BranchEq	E	E	vl	<>	if vl=vr then dl,⊥ else ⊥,dr	N
BranchNeq	E	E	vl	<>	if vl≠vr then dl,⊥ else ⊥,dr	N
BranchLt	E	E	vl	<>	if vl<vr then dl,⊥ else ⊥,dr	N
BranchLtEq	E	E	vl	<>	if vl≤vr then dl,⊥ else ⊥,dr	N
BranchGt	E	E	vl	<>	if vl>vr then dl,⊥ else ⊥,dr	N
BranchGtEq	E	E	vl	<>	if vl≥vr then dl,⊥ else ⊥,dr	N
BranchUEq	E	E	vl	<>	if vl=vr then dl,⊥ else ⊥,dr	N
BranchUNeq	E	E	vl	<>	if vl≠vr then dl,⊥ else ⊥,dr	N
BranchULt	E	E	vl	<>	if vl<vr then dl,⊥ else ⊥,dr	N
BranchULtEq	E	E	vl	<>	if vl≤vr then dl,⊥ else ⊥,dr	N
BranchUGt	E	E	vl	<>	if vl>vr then dl,⊥ else ⊥,dr	N
BranchUGtEq	E	E	vl	<>	if vl≥vr then dl,⊥ else ⊥,dr	N

Function Call and Return Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
CallInitial	T	L	vl	<rts:=Stack(rts,rt,vr), rt := 0>	dl,⊥	O
CallRec	T	L	vl	<rt := rt+1>	dl,⊥	O
CallRecL	T	L	vl	<rts:=Stack(rts,rt,vr), rt := 1>	dl,⊥	O
CallRecR	T	L	vl	<rts:=Stack(rts,rt,vr), rt := 2>	dl,⊥	O
CallRecC	T	L	vl	<rts:=Stack(rts,rt,vr), rt := 3>	dl,⊥	O
CallRecλ	T	L	vl	<rts:=Stack(rts,rt,vr), rt := λ>	dl,⊥	O

Return	T	L	vl	if rt=0 then
				<rt := Getrt (rts,vr),
				rts := Getrts (rts,vr)>
				else
				<rt := rt-1>
					if rt=0 then dl,⊥
					else ⊥,dr
						N

ReturnT	T	L	vl	<rt := Getrt (rts,vr), rts := Getrts (rts,vr)>	dl,⊥	N
ReturnL	T	L	vl	<rt := Getrt (rts,vr), rts := Getrts (rts,vr)>	if rt=1 then ⊥,dr else dl,⊥	N

ReturnR	T	L	vl	if rt=2 then
				<rt := Getrt (rts,vr),
				rts := Getrts (rts,vr)>
				else
				<>
					if rt=2 then ⊥,dr
					else dl,⊥
						N

ReturnC	T	L	vl	if rt=3 then
				<rt := Getrt (rts,vr),
				rts := Getrts (rts,vr)>
				else
				<>
					if rt=3 then ⊥,dr
					else dl,⊥
						N

Returnλ	T	L	vl	if rt=λ then
				<rt := Getrt (rts,vr),
				rts := Getrts (rts,vr)>
				else
				<>
					if rt=λ then ⊥,dr
					else dl,⊥
						N

where λ = 1, 2, 3, ... to some implementation limit;
For a simple arithmetic implementation of the RecTagStack let
Stack(rts,rt,vr) = rts*vr + rt; Getrts (rts,vr) = rts div vr; and Getrt (rts,vr) = rts mod vr.
The stacking and un-stacking functions may be other mechanisms as long as they behave functionally.

Index Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
Setφ	E	E	vl	<iφ := vr>	dl,dr	O
Getφ	T	N	iφ	<>	dl,dr	N
Incφ	E	E	vl	<iφ := iφ+vr>	dl,dr	O
Decφ	E	E	vl	<iφ := iφ-vr>	dl,dr	O
SplitφEq	E	E	vl	<>	if iφ=vr then dl,⊥ else ⊥,dr	N
SplitφNeq	E	E	vl	<>	if iφ≠vr then dl,⊥ else ⊥,dr	N
SplitφLt	E	E	vl	<>	if iφ<vr then dl,⊥ else ⊥,dr	N
SplitφLtEq	E	E	vl	<>	if iφ≤vr then dl,⊥ else ⊥,dr	N
SplitφGt	E	E	vl	<>	if iφ>vr then dl,⊥ else ⊥,dr	N
SplitφGtEq	E	E	vl	<>	if iφ≥vr then dl,⊥ else ⊥,dr	N
Swapxy	T	N	vl	<ix := iy, iy := ix>	dl,dr	N
Swapxz	T	N	vl	<ix := iz, iz:= ix>	dl,dr	N
Swapxt	T	N	vl	<ix := it, it := ix>	dl,dr	N
Swapxs	T	N	vl	<ix := is, is := ix>	dl,dr	N
Swapyz	T	N	vl	<iy := iz, iz := iy>	dl,dr	N
Swapyt	T	N	vl	<iy := it, it := iy>	dl,dr	N
Swapys	T	N	vl	<iy := is, is := iy>	dl,dr	N
Swapzt	T	N	vl	<iz := it, it := iz>	dl,dr	N
Swapzs	T	N	vl	<iz := is, is := iz>	dl,dr	N
Swapts	T	N	vl	<it := is, is := it>	dl,dr	N

where φ = [x, y, z, t, s].

Index Set Descriptor Operations:

Mnemonic	vl	vr	Value Function	Tag Function	Destination	Error
SetHigh	E	E	X(S(vl),B(vl),L(vl),vr)	<>	dl,dr	O
SetLow	E	E	X(S(vl),B(vl),vr,H(vl))	<>	dl,dr	O
SetBlock	E	E	X(S(vl),vr,L(vl),H(vl))	<>	dl,dr	O
SetStride	E	E	X(vr,B(vl),L(vl),H(vl))	<>	dl,dr	O
GetHigh	T	N	H(vl)	<>	dl,dr	N
GetLow	T	N	L(vl)	<>	dl,dr	N
GetBlock	T	N	B(vl)	<>	dl,dr	N
GetStride	T	N	S(vl)	<>	dl,dr	N
GetLength	T	N	max(0,H(vl)-L(vl)+1)	<>	dl,dr	N

where H(v) = v mod 2^20, L(v) = (v div 2^20) mod 2^20, B(v) = max(1, (v div 2^40) mod 2^12), S(v) = max(1, v div 2^52), and X(s,b,l,h) = h + 2^20*(l + 2^20*(b + 2^12*s)). Note that the sizes of the ISD fields are implementation dependent.

The following instructions are only used in Chapter 7.5. and represent an enhancement of the basic SDFA model.

Enhanced Instruction-set Operations:

Mnemonic	vl	vr	Value Functions	Tag Function	Destination	Error
Proliferateφ	T	L	v1	<iφ := iφ - 1> <iφ := iφ>	if iφ = rv then ⊥,dr else dl,dr	N
ShiftφR	T	E	v1	<if iφ < H(vr) then iφ := iφ + 1 else iφ := L(vr)>	dl,dr	N
ShiftφL	T	E	v1	<if iφ > L(vr) then iφ := iφ - 1 else iφ := H(vr)>	dl,dr	N

where φ = [x, y, z, t, s],
H(v) = v mod 2^20, L(v) = (v div 2^20) mod 2^20, B(v) = max(1, (v div 2^40) mod 2^12),
S(v) = max(1, v div 2^52), and X(s,b,l,h) = h + 2^20*(l + 2^20*(b + 2^12*s)). Note that the sizes of the ISD fields are implementation dependent, and may even depend on the ProcessTag.

A.1.4.3. Index Set Descriptor

For each dimension of an array there is an index set descriptor (ISD), which contains fields for *low*, *high*, *stride*, and *block*. Where *low* and *high* are the first and last indices of the array, and *block* is the number of consecutive indices occurring at intervals of *stride*. The following invarient must hold for all ISDs:

$$(block <= stride \ \& \ low <= high \ \& \ block <= high\text{-}low+1) \mid (low = 1 \ \& \ high = 0).$$

The final pair of conditions is used to indicate the null array dimension. If *stride* equals *block*, the array is a dense, that is all indices are present. For simplicity of coding, if block or stride is equal to zero, it is the same as if block or stride were equal to one. The field widths for this implementation are: stride - 12 bits, block - 12 bits, low - 20 bits, and high - 20 bits. These parameters could also be a function of the ProcessTag. This would, for example, allow an individual user to set block and stride to much smaller widths and, hence, use very long, dense arrays.

A.2. Physical Architecture

The following sections contain details of the physical architecture which were not included in Chapter 3.

A.2.1. Simplification of the Packet Switch Model

The arbitrator in the packet switch allows only one token to be routed to a given output port during one beat of the output pipeline. This is a simplification of a true packet switch as it does not account for routing conflicts inside the switch itself. The number of possible conflicts per beat modelled is (N-1), where N is the number of processors. A packet switch constructed from 2x2 butterfly switches would create N * Log_2(N)/2 potential conflicts per beat.[91] For the sizes of system which can currently be simulated,

[91] The full formula is n(m-1)Log_m(n)/m potential conflicts per beat, where n is the number of processors and m is the complexity of the switching element.

the difference is quite small. In a 16 processor system using 2x2 switches, this approach accurately models 47% of the conflicts. In a 256 processor system using 4x4 switches, 33% of the conflicts are accurately modelled. These values are acceptable in light of the token traffic results in Chapter 7. However, in a large system, a more detailed model would be required, since in a system with 32K processors using 8x8 switches, only 23% of the potential conflicts modelled accurately.

A.2.2. Processing Element Details

The following sections contain details pertaining to the processing element that were omitted from Chapter 3, but which are, nonetheless, relevant.

A.2.2.1. Instruction Class Assignment

Table 1 contains the assignment of instruction classes to function units within the processing element.

Instruction Class	Function Unit
Integer and Unsigned Arithmetic	Integer
Integer Div and Mod	Divide-Sqrt
Floating Point Arithmetic	Floating Point
Floating Point FDiv, FMod, Fsqrt	Divide-Sqrt
Floating Point Transcendental Functions	Transcendental[92]
All Relational Operations	Integer
Boolean and Bit Operations	Integer
Special Operations	Integer
Control Operations	Integer
Call and Return Operations	Call-Return
Index Operations	Integer
ISD Operations	Integer
Enhanced Instruction-set Operations	Integer

Table 1: Assignments of instruction-set classes to function units.

[92] The transcendental function unit is provided in the simulator only to initialize some of the test programs. The impact of these instructions on the overall results are negligible. If a hardware implementation allowed its inclusion, the current design parameters would be reasonable.

A.2.2.2. Call-Return Function Unit Details

Before providing a justification of the design parameters for this function unit, a workable call-return mechanism is required. The following are some ways that the RecTagStack can be maintained.

- The simplest approach is the multiplication approach described earlier. This approach, although very simple, would not be workable, since the technique is very wasteful of tag field bits. For example, in a binary recursive program, the number of values used by this mechanism is 3^l, where l is the depth of recursion. If the call tree is dense,[93] then the number of values actually needed is 2^l, representing a waste of $3^l - 2^l$.

All the remaining schemes require a memory in which to store data representing the relationships between values of the RecTagStack field (the "Call-tree"). There are a variety of techniques which all share a basic theme. When the first token, requiring new RecTagStack value, is encountered by the system, a free slot in a table is used to store the old values of the RecTag and the RecTagStack, and the address of the slot is used as the new RecTagStack field. For all further tokens, this table is searched and the slot address is used as before. Since tokens, having the same RecTag and RecTagStack value, can exist in different processors, the Call-tree must be propagated throughout the system. The mechanism must, also, ensure that each new RecTagStack value is unique. Several strategies for this follow:

- A global "tag bus" is attached to all processors in the system. This bus is used to propagate a new Call-tree table entry to all the processors in the system. Locking of the bus would ensure that race conditions did not arise when creating new table entries. Since the goal of this study is to build an extensible system, this approach is rejected.

Rather than using a separate bus to perform the distribution, the system network can be used for synchronization and propagation. This is the case with the following two approaches, but each uses a different technique for avoiding race conditions.

- All calls, not satisfied locally, are sent to a "master" processor. The new RecTagStack value is generated and then propagated throughout the system, via the network.

- The approach adopted is like the one above, but organized to distribute the "master" role, based on the current value of the RecTagStack.

The following scheme is implemented in the simulator and is the default used throughout this study. This scheme requires a memory on each processor which contains a local copy of the Call-tree. If an entry does not exist for a particular RecTag and RecTagStack combination, either a new entry is generated for it on the current processor or the token is re-routed to the processor which will *own* the particular RecTagStack value. *Ownership* is based on the value of the current RecTagStack field and the processor number. In any case, the owner processor creates the RecTagStack entry and propagates the new entry through the system. This scheme is outlined in the following:

---

93 E.g. all possible branches are taken, a feature somewhat rare in most programs of this sort.

If a call instruction fails to find an entry in the Call-tree table

Then

> If the current processor is to be the owner of the new RecTagStack value
>
> Then
>
> > The current processor creates a new entry in the RecTagStack table containing the old RecTag and RecTagStack value and the new RecTagStack value.[94]
> >
> > The left result of this call instruction is the normal result of the call. The right result, only present in this case, is a new value containing the new RecTagStack value and an offset field, whose value is half the number of processors. This result is sent to an SetRTS instruction, planted at a fixed location in memory on the processor whose number is one more that the current processor's modulo the number of processors.
>
> Else
>
> > The token is re-routed to the same call instruction in the processor which will own the new RecTagStack value.

Else

> The call instruction takes place on the current processor using the Call-tree table entry.

The SetRTS instruction creates an entry in the RecTagStack table of the executing processor, based on the tag and value of the incoming token. If the value of the offset field is greater than one, a new value is created which contains the same RecTagStack value but half the offset. This value is then sent to the same SetRTS instruction on the next processor and the processor whose number is "offset" more than the current processor's number (modulo the number of processors). In this way the information is propagated in $Log_2(N)$ steps (N = the number of processors).[95]

Return instructions simply look-up the old values of the RecTag and RecTagStack[96].

Brief consideration of this scheme raises the question of the size of the local Call-tree table, which is an expensive, associative memory. A few points should be noted. First, the existence of data structures in a program does not, in any way, effect the number of entries in this table.[97] Second, this mechanism is used only when a recursive function is called from within another recursive function and for multiple

---

94 Note that the new RecTagStack value must be unique and therefore must be functionally derived from the current processor number. Also, the new RecTagStack value should ensure that future RecTagStack values are created in other processors.

95 Note that tokens carrying this information must share system resources with computational tokens. Most importantly, the time spent by these tokens in the token queue could result in a large number of tokens being routed unnecessarily to their respective "master" processors. However, since all SetRTS instructions are planted at a specific location, the token queue gives priority to these tokens. This approach could be applied to a section of the program address space and, thereby, provide a general mechanism for low latency execution of system service functions.

96 There is a possibility that a return instruction could arrive at a processor which had not yet received an entry for the relevant slot in its table. Such programs do not occur in practice and are rather contrived. The simulation system assumes that they don't happen, but, if they do, a warning error is issued. The solution to this problem would be to send such tokens back to the token queue, similarly to defer tokens in the Manchester Data-Flow Machine's matching store [Gurd87a].

97 In the Manchester Data-Flow machine, tag space is used to identify each instance an array which is one of the elements in another array, e.g. the columns of a matrix each have a tag associated with them.

recursion. Third, similarly structured call trees in different parts of the program can use the same entries in the table. Fourth, similarly structured call trees belonging to different users can use the same entries in the table. Last, in many cases table entries can be shared by differently structured call trees. For example, a multiply recursive call tree has entries in common with a call tree composed of nested calls to recursive functions.

A bonus of this table sharing is that, once the tables have been partially filled (by the execution of a variety of programs), programs run later will frequently find their calls satisfied locally. It is even possible to envision a short suite of small programs which would set up a system "tuned" to the local job mix.

There is the possibility that a particularly complex, multiply recursive program can fill up the table with rarely used entries. Clearing up after this kind of situation might require a re-initialization of the system, unless some special mechanism is provided.

One such mechanism is to place jobs into classes and include an entry in the table indicating which classes use a given entry. Jobs with complex behaviour can be run with a ProcessTag indicating this. The operating system, after ensuring that no jobs in this class are running, clears those entries which are used only by programs from the given class. By controlling which job classes are running at a given time, the tables can be cleared regularly. Notice that sharing of entries is still possible between jobs in different classes.

Another mechanism, for limiting the number of entries, is to generate sequential code for inner, and possibly outer, levels of a program. Also, by introducing a ROM based section of the table containing "frequently used" entries, the size of the dynamic part of the table can be reduced.

A.2.2.3. Short Stay Matching Store Note

There is one situation in which the Short Stay Matching Store (SSMS) does not perform ideally. In some short, strict, re-entrant code fragments, it is possible for a token to match with a token from another instance of the code fragment. In order for this problem to arise, such a code fragment would have to be fully strict, rely on this strictness for re-entrancy, and not use the call-return tagging mechanism. None of the codes used in this study rely on strictness to ensure re-entrancy. Should such a code be implemented, some mechanism would be required to prevent tokens from one iteration matching with those from another. There are several straightforward approaches: 1) Set the ProcessTag of the program, and ProcessTag control structure in the processing element, to indicate that copies of forwarded tokens with this ProcessTag are not to be kept in the SSMS when forwarded to the main matching store; and 2) add a new instruction which clears the forwarded section of the SSMS and which could be executed inside the implied barrier of a strictly coded function. This could be either at the start or the end of the function.

A.2.3. Matching Store Details

The details of the matching and migrating functions of the main matching store are presented in the following sections.

A.2.3.1. Matching

When an incoming token arrives at a main matching store bank, the level one store is searched first and, if a match is found, the token pair is sent to the token queue. If there is no matching token in the first level store the token must be forwarded to level two and the matching process repeated there. If the level one store is not full, a copy of the incoming token is stored in level one. Thus, if no match is found in the second level, this copy of the token can then be discarded. However, should a match be found in level two, the token stored in level one must be removed. This is accomplished by creating a clearance token and sending it to the level one store, in addition to sending the matched pair to the token queue. This technique exploits temporal locality at the main matching store as does the related technique in the SSMS.

Table 2 contains all possible actions that an incoming token can take and the circumstances surrounding it. The actions are defined as follows:

Store - No match is found and the token is stored at the given level. The exception to this is that, if the token is marked with a clearance action, it will not be stored in level one, since its partner is in level two.

Extract - A match is found, extracted from the store, and the pair is sent to the token queue or, if either token was marked with a clearance action, both tokens are destroyed.

Clear - A pair has been extracted at level two. The pair is marked to indicate that a copy of one of the tokens must be sent to level one as a clearance token.

Discard - No match is found and the token is destroyed.

Forward - No match is found in level one and the token is sent to level two.

Level-1 Full	Matched in Level-1	Matched in Level-2	Stored in Level-1	Action in Level-1	Action in Level-2
False	True	-	-	Extract	-
False	False	True	True	Store & Forward	Extract & Clear
False	False	True	False	Forward	Extract
False	False	False	True	Store & Forward	Discard
False	False	False	False	Forward	Store
True	True	-	-	Extract	-
True	False	-	-	Forward	Store
True	False	True	-	Forward	Extract
True	False	False	-	Forward	Store

Table 2: Truth Table of Matching Actions in Matching Store Banks.

A.2.3.1 Migrating

The migration process acts in a similar fashion to caches in conventional systems. Like the SSMS, the design of the migration function is motivated by the assumption that there is temporal locality in the arrival of matching pairs of tokens.

Under this assumption it would be desirable to allow "old" tokens to migrate up to the second level and "young" tokens to remain in the first level. The problem is that a newly arrived token, which fails to find a match in the first level, must be sent to the second level, thus putting it in the wrong place. It may have been possible to store it in level one, as noted above, but, if level one is already full, the "young" token will be put into level two. The migration action attempts to ensure that there is always room in level one for incoming tokens. The solution adopted in this implementation is to migrate a token, chosen at random, up to level two when the occupancy of level one exceeds some threshold.[98] The migration process is carried out when the matching store bank is otherwise idle.

A.2.4. Pin Counts

Recall, that each of the main components of the processor is packaged on a single chip. The number of pins, attached to the chip, is a serious design limitation. The design decision, that a token can be sent from one component to another in a single cycle, forces the transfer to be done in parallel, thus increasing the pin count. The token queue and the matching store each require parallel data paths for three full tokens plus two additional values. The processing element requires only three full tokens plus one additional value, but also needs access to a program store. Since the size of a data value is fixed at 64 bits, the pin count is driven by the size of the tag. The exact size of the tag part of this study, but it is expected that a minimum of 64 bits would be required. The corresponding pin count is 512 for data only. Additional pins would be required for power, external control, and debug. This is within current packaging technology, but not commonplace. The DEC Alpha has 431 pins [DEC92a], and the T9000 has 208 pins [Inmo91a]. If many more bits are required in the tag, the intra-chip connections would need to be multiplexed.

---

98 An enhancement of this scheme allows migration in both directions. Should the occupancy of the first level fall below some threshold, a token is migrated from the second level down to the first level.

A.3. The Architecture Configuration File.

```
Cycle time of this system E in nanoseconds                 : 6.0

Topology of the network, [Packet, Torus, Hypercube]        :
Packet
Dimension of the network (0 for Packet switch)             : 0
Number of processors in the network                        : 1
The sizes of the various dimensions (e.g. 4 4 4)           : na
Length of the inter-processor pipelines                    : 1
Beat of the inter-processor pipelines                      : 2

Length of the processor input queues                       : 4
Width of the processor input queues                        : 1
Length of the processor output queues                      : 16
Width of the processor output queues                       : 1

Type of matching store, [Simple, Forwarding]               :
Forwarding
Number of banks in a matching store                        : 8
Number of matching store input pipelines                   : 2
Length of the matching store input pipelines               : 1
Beat of the matching store input pipelines                 : 1
Length of the matching store output queue                  : 4
Width of the matching store output queue                   : 1

Maximum size of level-1 store in a matching store bank     : 65535
Percent of level-1 maximum at which forwarding begins      : 99
Length of the level-1 input queue                          : 4
Width of the level-1 input queue                           : 1
Length of the clear queue                                  : 20
Width of the clear queue                                   : 1
Length of the level-1 pipeline                             : 2
Beat of the level-1 pipeline                               : 2
Length of the forwarding queue                             : 8
Width of the forwarding queue                              : 1
Length of the return queue                                 : 16
Width of the return queue                                  : 1
Length of the level-1 output queue                         : 16
Width of the level-1 output queue                          : 1
Length of the output pipeline                              : 2
Beat of the output pipeline                                : 1

Length of the multiplexed level-1/2 transfer pipeline      : 1
Beat of the multiplexed level-1/2 transfer pipeline        : 2

Maximum size of level-2 store in a matching store bank     : 65535
Length of the level-2 input queue                          : 4
Width of the level-2 input queue                           : 1
Length of the level-2 pipeline                             : 5
Beat of the level-2 pipeline                               : 4
Length of the level-2 output queue                         : 2
Width of the level-2 output queue                          : 1
```

```
Type of token queue, [Queueing, Stacking, Sorting]          :
Queueing
Maximum length of Token Queue                               : 65535
Length of the token queue input pipelines                   : 1
Beat of the token queue input pipelines                     : 1
Length of the token queue output queue                      : 4
Width of the token queue output queue                       : 2

Length of the processing element input pipeline             : 1
Beat of the processing element input pipeline               : 1
Number of processing element output pipelines               : 2
Length of the processing element output pipelines           : 1
Beat of the processing element output pipelines             : 1
Length of the processing element input queue                : 16
Width of the processing element input queue                 : 2
Length of the processing element output queue               : 4
Width of the processing element output queue                : 2

Instruction fetch arbitration [Fair, Local, Match]          : Local
Complex Instruction-set facility enabled                    : False
Number of active instruction fetch pipelines                : 2
Length of the instruction fetch pipeline(s)                 : 1
Beat of the instruction fetch pipeline(s)                   : 1

Length of the Call-Return input queue                       : 4
Width of the Call-Return input queue                        : 2
Length of the Call-Return pipeline                          : 2
Beat of the Call-Return pipeline                            : 1
Length of the Call-Return output queue                      : 4
Width of the Call-Return output queue                       : 2

Length of the Transcendental input queue                    : 4
Width of the Transcendental input queue                     : 2
Length of the Transcendental pipeline                       : 1
Beat of the Transcendental pipeline                         : 15
Length of the Transcendental output queue                   : 4
Width of the Transcendental output queue                    : 2

Length of the Div-Sqrt input queue                          : 4
Width of the Div-Sqrt input queue                           : 2
Length of the Div-Sqrt pipeline                             : 1
Beat of the Div-Sqrt pipeline                               : 10
Length of the Div-Sqrt output queue                         : 4
Width of the Div-Sqrt output queue                          : 2

Length of the Floating Point input queue                    : 4
Width of the Floating Point input queue                     : 2
Length of the Floating Point pipeline                       : 5
Beat of the Floating Point pipeline                         : 1
Length of the Floating Point output queue                   : 4
Width of the Floating Point output queue                    : 2

Number of Integer function units, [1,2]                     : 2
Length of the Integer input queue(s)                        : 4
Width of the Integer input queue(s)                         : 1
Length of the Integer pipeline(s)                           : 2
Beat of the Integer pipeline(s)                             : 1
Length of the Integer output queue(s)                       : 4
Width of the Integer output queue(s)                        : 2

Local by-pass enabled, [True, False]                        : True
Length of by-pass queue                                     : 32
Width of by-pass queue                                      : 2
```

```
Short stay matching store, [None, Simple, Rectifying]      :
Rectifying
Number of cycles to wait in the short stay matching store  : 64
Size of the short stay matching store                      : 512
Length of short stay matching store input queues           : 4
Width of short stay matching store input queues            : 2
Length of the short stay matching store pipelines          : 1
Beat of the short stay matching store pipelines            : 1
Length of the short stay matching store success queue      : 4
Width of the short stay matching store success queue       : 2
Length of the short stay matching store fail queue         : 4
Width of the short stay matching store fail queue          : 2
```

Appendix B: Details of the Experimental Environment

B.1. SDAL

B.1.1. BNF Syntax

The variant of BNF notation used here is the British Standards Institution's (BSI) syntactic metalanguage defined by BS 6154, 1981. All terminal components of the SDAL are in bold and non-terminal structure names are always in italics.

Level 1: ? White space is required between two non-terminals. ?

assembly_unit	=	'(', { *constant_def* | *function_def* | *token_def* }, ')' .
constant_def	=	'(', '**def_con** ', *con_identifier*, *constant* , ')' .
function_def	=	'(', '**def_fun** ', *identifier*, *argument_list*, *node_list*, ')' .
token_def	=	'(', '**def_tok** ', *constant_ref*, '**:**', *destination*, ')' .
argument_list	=	'(', *arg_spec*, { *arg_spec* }, ')' .
node_list	=	'(', { *node* }, ')' .
arg_spec	=	(*arg_type*, '**:**', *destination*) | *con_identifier* .
arg_type	=	'**s**' | '**a**' | '**d**' .
node	=	'(', *node_number*, *identifier*, *input_list*, *destination_list*, ')' .
input_list	=	'(', *input_spec*, { *input_spec* }, ')' .
input_spec	=	*arg_type* | *constant_ref* .
destination_list	=	'(', *destination*, { *destination* }, ')' .

Level 2: ? No white space allowed. ?

destination	=	*node_number*, *port_indicator* .
node_number	=	*digit* , { *digit* } .
port_indicator	=	'**L**' | '**R**' | '**N**' | '**a**' .. '**z**' .
constant_ref	=	*con_identifier* | *constant* .

con_identifier	=	'**#**', *identifier* .
constant	=	'**#**', (*decimal_constant* | *hex_constant*) .
decimal_constant	=	['**+**' | '**-**'], *digit* ,{ *digit* }, ['**.**', { *digit* }, [('**e**' | '**E**'), ['**+**' | '**-**'], *digit* , { *digit* }]] .
hex_constant	=	'**0x**', *hex_digit*, { *hex_digit* } .
identifier	=	*letter*, { *letter* | *digit* | *symbol* } .
symbol	=	'**_**' | '**?**' .
letter	=	'**a**' .. '**z**' | '**A**' .. '**Z**' .
digit	=	'**0**' | '**1**' | '**2**' | '**3**' | '**4**' | '**5**' | '**6**' | '**7**' | '**8**' | '**9**' .
hex_digit	=	*digit* | '**a**' .. '**f**' | '**A**' .. '**F**' .

Notes:

1- L and R indicate the left and right ports of simple nodes. N indicates that no result token is to be sent on that port. This may be omitted for right port indicators.

2- The letters 'a'..'z', when used as port indicators, refer to the argument number (or result number) of functions.

3- Within a function definition, nodes start numbering at 1, and a zero node destination indicates that the arc is a result arc of the function.

4- Identifiers may be names of constants, functions, or the mnemonics of the primitive nodes.

5- A ';' indicates that the remainder of the line is a comment.

6- New-lines and tabs are treated as white space.

7- In the program as a whole, the function called 'main' is node number 1 of the 0th function, and all initial tokens (those defined in 'def_tok' statements) must have destinations 1a, 1b, etc. The results of the program are the result tokens from the function 'main'.

8- The argument types 'a' (array) and 'd' (array descriptor) should appear only in function input and argument lists. All primitive nodes have only scalar ('s') and constant ('#') inputs.

9- In the argument list of a function, each array argument must be preceded by the ISD's describing it, the 'X' dimension first then the others in order. If there is a missing dimension, with respect to a given function, its descriptor should be omitted.

10- A constant name, appearing in a function input list, creates a local name for a constant, defined where the function is invoked.

11- Function input lists and argument lists must type match for scalars, arrays, array descriptors, and constants.

B.1.2. Examples

The following examples illustrate the major features of SDAL used in the creation of benchmark programs for SDFA. Where omitted, the Data-Flow graphs for these programs can be found in Appendix A.1.3.

B.1.2.1. Linear Recursion: Facl

```
function facl (n: integer returns integer)
if (n = 0) then
   1
else
   n * facl (n-1)
end if
end function

function main (returns integer)
let
   n := 9
in
   facl (n)
end let
end function


; -- Linear recursive version of factorial.
((def_con #maxd #1024)
 (def_tok #9:1a)

 (def_fun facl (s:1L #maxd)
   ((1 BranchEq    (s #0)     (2R 3L))
    (2 SelectLeft  (#1 s)     (7L))
    (3 Dup         (s)        (6L 4L))
    (4 Sub         (s #1)     (5L))
    (5 CallRec     (s #maxd)  (1L))
    (6 Mul         (s s)      (7L))
    (7 Return      (s #maxd)  (0a 6R))))

 (def_fun main (s:1L)
   ((1 CallInitial (s #maxd)  (2a))
    (2 facl        (s #maxd)  (0a)))))
```

B.1.2.2. Multiple Recursion: Fib

```
function fib (n: integer returns integer)
if (n <= 1) then
   1
else
   fib (n-1) + fib (n-2)
end if
end function
```

```
; -- Multiply recursive Fibonacci sequence.
((def_con #maxd #3)
 (def_tok #9:1a)

 (def_fun fib  (s:1L #maxd)
   ((1  BranchLtEq  (s #1)     (2R 3L))
    (2  SelectLeft  (#1 s)     (9L))
    (3  Dup         (s)        (4L 5L))
    (4  Sub         (s #1)     (6L))
    (5  Sub         (s #2)     (7L))
    (6  CallRecL    (s #maxd)  (1L))
    (7  CallRecR    (s #maxd)  (1L))
    (8  Add         (s s)      (9L))
    (9  ReturnR     (s #maxd)  (10L 8R))
    (10 ReturnL     (s #maxd)  (0a 8L))))

 (def_fun main (s:1L)
   ((1  CallInitial (s #maxd)  (2a))
    (2  fib         (s #maxd)  (0a)))))
```

B.1.2.3. Tail Recursion and Arrays: Smooth

```
type vector = array [real]

function smooth (v: vector; times: integer returns vector)

if (times = 0) then
   v
else
   let
      newv := for i in 0,249
              returns array of
                 if ((i = 0) | (i = 249)) then
                    v[i]
                 else
                    (v[i-1] + v[i] + v[i+1])/3.0
                 end if
              end for
   in
      smooth (newv, times - 1)
   end let
end if
end function

function main(returns array [real])
let
   times := 10;
   span  := 10.0;
   v     := for i in 0,249
            returns array of
               if (mod(i,2) = 0) then 1.0 else span end if
            end for
in
   smooth (v, times)
end let
end function
```

Smooth:

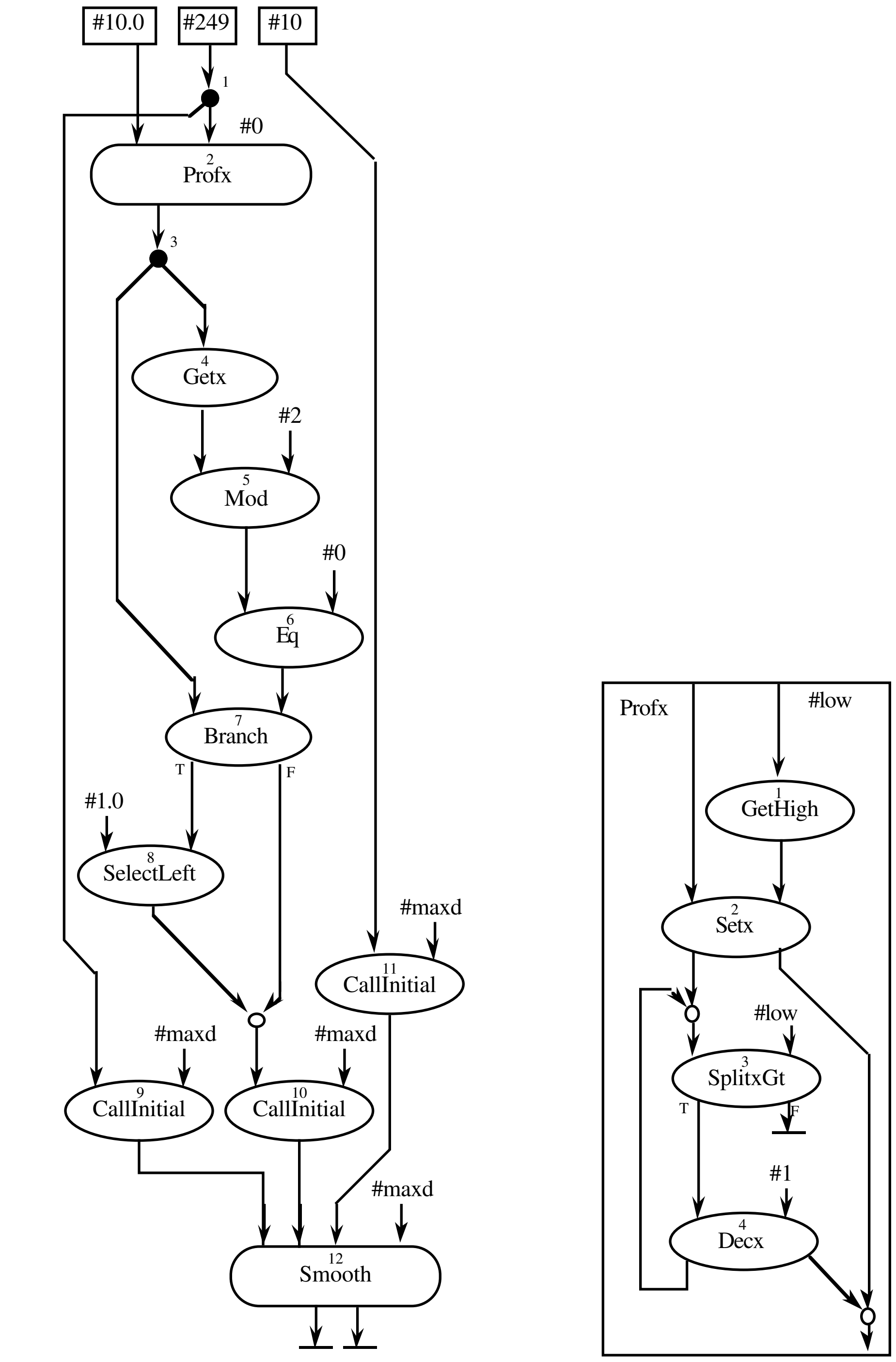


Figure B.1: Data-Flow Graph for Smooth Main Program and Profx Function.

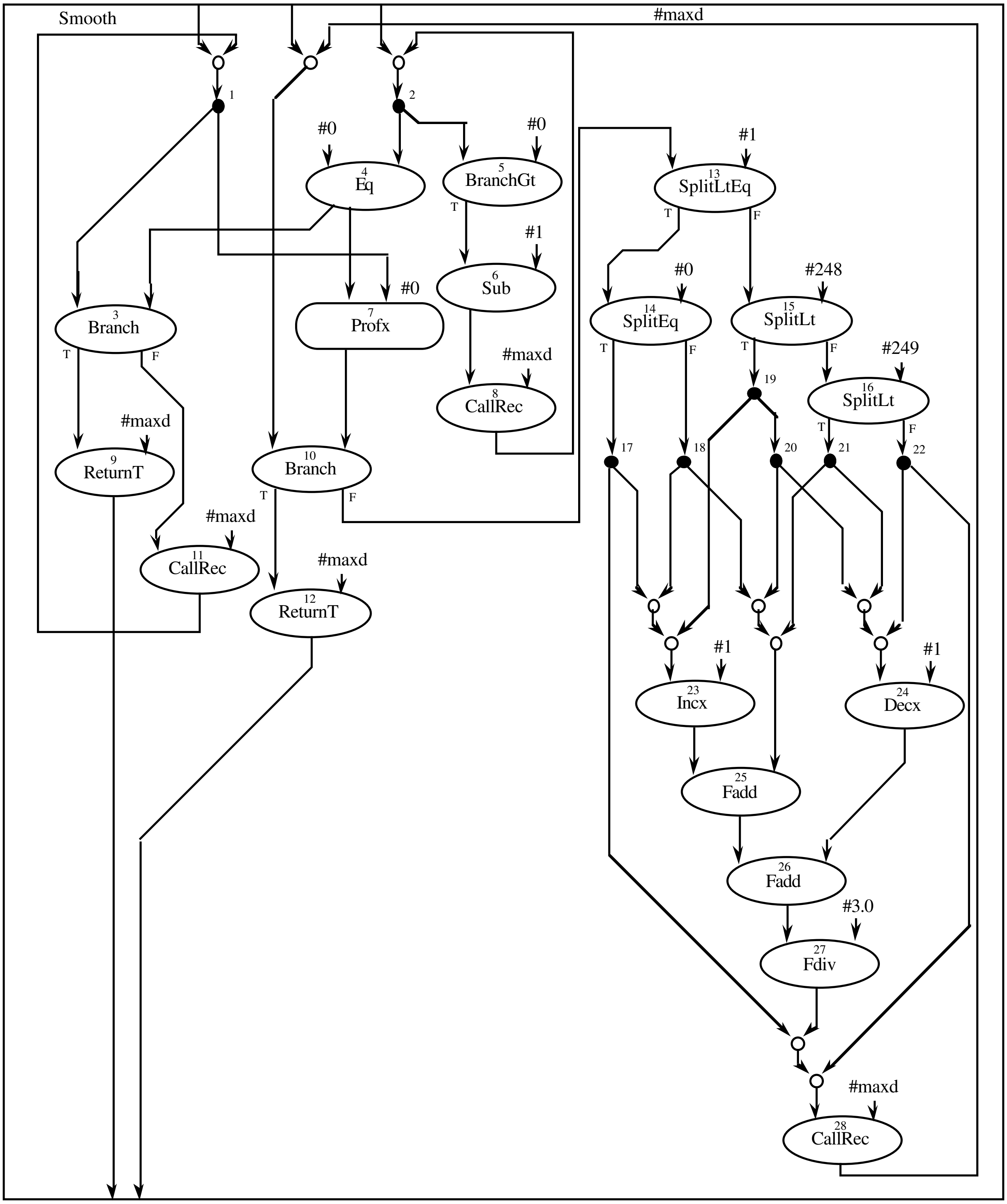


Figure B.2: Data-Flow Graph for Function Smooth

```
; -- Smooth
((def_con #maxd #1000)
 (def_tok #10.0:1a)
 (def_tok  #249:1b)
 (def_tok   #10:1c)

 ;-- Profx: Input a is proliferated with x indices [0..b]
 (def_fun Profx (s:2L s:1L #low)
   ((1 GetHigh  (s)       (2R))
    (2 Setx     (s s)     (3L 0a))
    (3 SplitxGt (s #low)  (4L))
    (4 Decx     (s #1)    (3L 0a))))

 ; -- Smooth
 (def_fun Smooth (d:1L a:10L s:2L #maxd)
   ((1  Dup         (s)        (3L  7b))
    (2  Dup         (s)        (4R  5L))
    (3  Branch      (s s)      (9L  11L))
    (4  Eq          (#0 s)     (3R  7a))
    (5  BranchGt    (s #0)     (6L))
    (6  Sub         (s #1)     (8L))
    (7  Profx       (s s #0)   (10R))
    (8  CallRec     (s #maxd)  (2L))
    (9  ReturnT     (s #maxd)  (0a))
    (10 Branch      (s s)      (12L 13L))
    (11 CallRec     (s #maxd)  (1L))
    (12 ReturnT     (s #maxd)  (0b))
    (13 SplitxLtEq  (s #1)     (14L 15L))
    (14 SplitxEq    (s #0)     (17L 18L))
    (15 SplitxLt    (s #248)   (19L 16L))
    (16 SplitxLt    (s #249)   (21L 22L))
    (17 Dup         (s)        (28L 23L))
    (18 Dup         (s)        (23L 25R))
    (19 Dup         (s)        (23L 20L))
    (20 Dup         (s)        (25R 24L))
    (21 Dup         (s)        (25R 24L))
    (22 Dup         (s)        (24L 28L))
    (23 Incx        (s #1)     (25L))
    (24 Decx        (s #1)     (26R))
    (25 Fadd        (s s)      (26L))
    (26 Fadd        (s s)      (27L))
    (27 Fdiv        (s #3.0)   (28L))
    (28 CallRec     (s #maxd)  (10L))))

 ; -- Main program for Smooth.
 (def_fun main (s:2a s:1L s:11L)
   ((1  Dup         (s)            (9L  2b))
    (2  Profx       (s s #0)       (3L))
    (3  Dup         (s)            (7L 4L))
    (4  Getx        (s)            (5L))
    (5  Mod         (s #2)         (6L))
    (6  Eq          (s #0)         (7R))
    (7  Branch      (s s)          (8R  10L))
    (8  SelectLeft  (#1.0 s)       (10L))
    (9  CallInitial (s #maxd)      (12a))
    (10 CallInitial (s #maxd)      (12b))
    (11 CallInitial (s #maxd)      (12c))
    (12 Smooth      (d a s #maxd)  (0a  0b)))))
```

B.2. Simulation Software

This section contains details of the interpreter, Sdi, and the simulator Sds.

B.2.1. The Interpreter

The following are the modules in the interpreter along with a brief description of each. The first three are unique to the interpreter; the rest are used by the simulator as well.

Sdi
: The main control program.

MatchingStorei
: Performs the matching function in the interpreter.

Processori
: Manages the flow of tokens within a processor and between processors.

Loader
: Loads the SDAL source code into the program memory and produces the initial token buffer.

Stats
: Implements the abstract data type for the statistics (events and magnitudes).

Token
: Implements the abstract data type for tokens.

Value
: Implements the abstract data type for the data field tokens.

Buffer
: Implements the abstract data type for a token buffer.

ProgramMemory
: Implements the abstract data type for storing the Instructions in a SDAL program.

PE
: Performs instructions on matched and by-passed tokens.

SdSystem
: Contains system-wide data types, constants, and functions.

SdError
: Contains system error routines.

The following pseudo code is a description of the basic flow of control in the interpreter.

```
BEGIN Sdi

   Get run time parameters.
   Create input buffers.
   Load SDAL program and fill input buffer with initial tokens.

   Initialize the statistics package.
   Initialize the instruction execution module (PE).
   Initialize the processor module.
   Initialize the matching store module.

   Create Processors and output buffers.

   REPEAT

      FOR all processors
         RunProcessor (see below).
      END FOR

      IF (this is a report step)    THEN
         Print interval report.
      END IF

      Increment the cycle counter.

      FOR all output buffers
         Append output buffer to input buffer.
      END FOR

   UNTIL (there are no active tokens).

   Print one last interval report.
   Print final statics.

   Clean up.

END Sdi.

BEGIN RunProcessor

   WHILE (the input buffer is not empty) DO

      Remove a token from the input buffer.
      Try to match.

      IF (match was successful or the token was a by-pass token)
THEN
         Execute the instruction.
         Add result tokens to the destination processor's buffer.
      END IF

   END WHILE

END RunProcessor.
```

B.2.2. The Simulator

The following are those modules used by the simulator that are not shared with the interpreter. Together, the simulator and interpreter are 13000 lines of Modula-2, 8700 lines of which are shared.

Sds
: The main control program.

Network
: Simulates the switch network and then the network of processors.

Processor
: Simulates an individual processor.

MatchingStore
: Simulates an individual matching store within a processor.

TokenQueue
: Simulates an individual token queue within a processor.

ProcessingElement
: Simulates an individual processing element within a processor.

ShortStayMS
: Simulates an individual Short Stay Matching Store within a processing element.

Queue
: Simulates a queue timing and storage primitive.

Pipe
: Simulates a pipe timing and storage primitive.

Architecture
: Loads the architecture configuration file and exports the configuration data to the other modules of the system.

The following pseudo code is a description of the basic flow of control in the simulator.

```
BEGIN Sds
   Get run time parameters.
   Initialize the statistics package.
   Initialize the instruction execution module (PE).
   Load architecture configuration file.

   Initialize Processing Element.
   Initialize Short Stay Matching Store.
   Initialize Matching Store.
   Initialize Token Queue.
   Initialize Processor.
   Initialize Network.

   Create network (this creates the whole system hierarchically).
   Load SDAL program and fill input queue with initial tokens.

   REPEAT

      SimulateNetwork (see below).
      Advance all queues.
      Advance all pipes.

      IF (this is a report step) THEN
         Print interval report.
      END IF

      Increment the cycle counter.

   UNTIL (there are no active tokens);

   Print one last interval report.
   Print final statics.
   Clean up.
END Sds.

BEGIN SimulateNetwork

   Collect tokens from the host & send them to destination
processor.
   Do all inter-processor transfer of tokens.

   FOR all processors
      SimulateProcessor (see below).
   END FOR
END SimulateNetwork.

BEGIN SimulateProcessor

   Distribute tokens from processing element and network to
           the matching store and token queue.
   Simulate the matching store.
   Move output tokens from matching store to the token queue.
   Simulate the token queue.
   Move output tokens from the token queue to the processing
element.
   Simulate the processing element.
END SimulateProcessor.
```

B.3. The Benchmark Suite

The following SISAL programs are the bench-mark used in this study.

B.3.1. Adaptive Quadrature

Sisal Code:

```
%      Program for recursive Adaptive Quadrature
%
define     Main
% --- The function to be integrated.
function func (x: real returns real)
   x*x*x
end function

% --- The test area for good result.
function good (old_area, new_area, delta_x: real returns boolean)
let
   tol := 0.0001
in
   (abs(old_area - new_area) < (delta_x * tol))
end let
end function

% --- The Adaptive Quadrature function
function aq (l, lv, r, rv: real returns real)

let
   mid := 0.5 * (l + r);
   midv := func(mid);
   old_area := 0.5 * (r - l) * (lv + rv);
   new_area := 0.5 * ((r - mid) * (rv + midv) +
                      (mid - l)  * (midv + lv))
in
   if good (old_area, new_area, r-l) then
      new_area
   else
      aq (l, lv, mid, midv) + aq (mid, midv, r, rv)
   end if
end let
end function

function   Main   (returns real)
let
   low   := 0.0;
   high  := 5.0;
   lowv  := func (low);
   highv := func (high)
in
   aq (low, lowv, high, highv)
end let
end function
```

B.3.2. Linear Recursive Factorial

Sisal Code:

```
function facl (n: integer returns integer)
if (n = 0) then
   1
else
   n * facl (n-1)
end if
end function

function main (returns integer)
let
   n := 9
in
   facl (n)
end let
end function
```

B.3.3. Tail Recursive Factorial

Sisal Code:

```
function fact (n, acc: integer returns integer)
if (n = 0) then
   acc
else
   fact (n-1, acc*n)
end if
end function

function main (returns integer)
let
   n := 9;
   a := 1
in
   fact (n, a)
end let
end function
```

B.3.4. Gaussian Elimination

Sisal Code:

```
function gauss (a: array [array [real]]
         returns   array [array [real]])
let
   k := array_limh (a)
in
   if (k = 0) then
      a
   else
      let
         rowk   := array [0: a[k]];
         aprime := for i in 0,k-1 cross j in 0,array_limh(a[0])
                   returns array of
                      a[i,j] - a[k,j] * a[i,k] / a[k,k]
                   end for
      in
         gauss (aprime) || rowk
      end let
   end if
end let
end function

function main (returns  array [real])

let
   n := 10
in
   let
      a  := for i in 0,n cross j in 0,n
            returns array of 1.0 - 1.0 / real(i+j+2)
            end for;
      ag := gauss (a)
   in
      for i in 0,n dot j in 0,n
      returns array of ag[i,j]
      end for
   end let
end let
end function
```

B.3.5. Matrix Multiply

```
Sisal Code:

type matrix = array[ array [real]]

function sdot (a, b: array [real] returns real)
   for x in a dot y in b
   returns value of sum (x * y)
   end for
end function

% A matrix multiplication program which does not
% do a transpose of the second matrix.

function matmult(a, b : matrix returns matrix)
for row in a cross col in b
returns array of
   sdot (row, col)
end for
end function

function main (returns array[array [real]])

let
   n := 9
in
   let
      a := array_fill (0, n, array_fill (0, n, 1.0));
      b := array_fill (0, n, array_fill (0, n, 2.0))
   in
      matmult(a, b)
   end let
end let
end function
```

B.3.6. Queens

Sisal Code:

```
type board = array [integer]

function safe (dim, queen: integer; brd: board returns boolean)
let
   rank := queen/dim;
   file := mod (queen, dim)
in
   for q in brd
      r := q/dim;
      f := mod (q,dim)
   returns value of product
      (rank ~= r) & (file ~= f) & (abs (rank-r) ~= abs (file-f))
   end for
end let
end function

function find (dim, queen: integer; brd: board returns integer)
let
   rank := queen/dim;
   file := mod (queen, dim)
in
   if rank=dim then
      1
   else
      if safe (dim, queen, brd) then
         find (dim, (rank+1)*dim, array_addh (brd, queen))
      else
         0
      end if
      +
      if (file+1) = dim then
         0
      else
         find (dim, queen+1, brd)
      end if
   end if
end let
end function

function main (returns integer)
let
   dim   := 8;
   brd   := array board [];
   queen := 0
in
   find (dim, queen, brd)
end let
end function
```

B.3.7. Saxpy

<u>Sisal Code:</u>

```
type matrix = array[ array [real]]

function sxpy (x, y : matrix; s : real returns matrix)
for xrow in x dot yrow in y
returns array of
   for xelem in xrow dot yelem in y
   returns array of
      s * xelem + yelem
   end for
end for
end function

function main (returns array[array [real]])

let
   n := 9
in
   let
      a := array_fill (0, n, array_fill (0, n, 1.0));
      b := array_fill (0, n, array_fill (0, n, 2.0));
      s : 3.1415926535
   in
      sxpy (a, b, s)
   end let
end let
end function
```

B.3.8. Shallow

Sisal Code:

```
%                       Program Shallow
%
%     Shallow is a simple weather prediction program which is
%     representative of the kind of computation involved in
weather
%     prediction models, although the data access pattern is
vastly
%     simplified.
%
%     The source for the model is: "The Dynamics of Finite-
Difference
%     Models of the Shallow Water Equations," Sadourny, R.,
Journal
%     of Atomospheric Sciences, Volume 32, Number 4, April 1975.
%
%     The original version of this code can be found in "Aspects
%     of Using Multiprocessors in Meteorological Modeling,"
Hoffmann
%     G-R., Swarztrauber, P. N., and Sweet, R. A. in
Multiprocessing
%     in Meteorological Models, Hoffmann, G-R. and Snellng, D. F.
eds.
%     (Springer-Verlag: Berlin) 1988.
%
%     Original SISAL code by David F. Snelling
%
%     This version adapted by G.K. Egan RMIT (Australia)
%
define    Main,                   % Main entry point to program.
          Initialize,             % Define initial condition.
          Fluxes,                 % Compute the mass fluxes.
          Height,                 % Compute the field height.
          Potential,              % Compute the total potential
velocity.
          TimeStep                % Compute next time step.
%
% convention used for cell neighbour indices is:
%
%                        j-1 is jn (north)
%     i-1 is iw (west)        i,j          i+1 is ie (east)
%                        j+1 is js (south)
%

% ----- Type Specifications -----
type     GridVariables     = array [array [real]];

% ----- Imported functions -----

global     DF_SIN (x: real returns real)
global     DF_COS (x: real returns real)
```

```
% ----- Initialize -----
function     Initialize  (M:          integer;
                          delta:      real
             returns      GridVariables, GridVariables,
GridVariables)

 let
     P           : GridVariables;
     Psi         : GridVariables;
     U           : GridVariables;
     V           : GridVariables;

     Amp         : real := 1.0E6; % Amplitude of initial waves.
     pie         : real := 3.14159265359;
     el          : real := real(M+1)*delta;
     pcf         : real := pie * pie * Amp * Amp / (el * el);
     delta_i     : real := 2.0 * pie / real(M+1);
     delta_j     : real := 2.0 * pie / real(M+1);

     Psi := for j in 0,M cross i in 0,M
            returns array of
               Amp * DF_SIN (delta_i*(real(i+1)-0.5))
                   * DF_SIN (delta_j*(real(j+1)-0.5))
            end for
in
     for j in 0,M cross i in 0,M
     returns array of
          pcf * (DF_COS (2.0*delta_i*real(i))
              +  DF_COS (2.0*delta_j*real(j))) + 50000.0
     end for,

     for j in 0,M
     returns array of
          let js := if j=M then 0 else j+1 end if in
          for i in 0,M
          returns array of
               -(Psi[js,i]-Psi[j,i]) / delta
          end for
          end let
     end for,

     for j in 0,M
     returns array of
          for i in 0,M
          returns array of
          let ie := if i=M then 0 else i+1 end if in
               (Psi[j,ie]-Psi[j,i]) / delta
          end let
          end for
     end for
end let
end function
```

```
% ----- Fluxes -----
function     Fluxes  (P, U, V:    GridVariables
             returns  GridVariables, GridVariables)

let M := array_limh(P) in
for j in 0,M
returns array of
     for i in 0,M
     returns array of
     let iw := if i=0 then M else i-1 end if in
          0.5 * (P[j,i] + P[j,iw]) * U[j,i]
     end let
     end for
end for,

for j in 0,M
returns array of
     let jn := if j=0 then M else j-1 end if in
     for i in 0,M
     returns array of
          0.5 * (P[j,i] + P[jn,i]) * V[j,i]
     end for
     end let
end for
end let
end function


% ----- Height -----
function     Height (P, U, V:     GridVariables
             returns GridVariables)

let M := array_limh(P) in
for j in 0,M
returns array of
     let js := if j=M then 0 else j+1 end if in
     for i in 0,M
     returns array of
          let ie := if i=M then 0 else i+1 end if in
               P[j,i] + 0.25*(U[j,ie]*U[j,ie] + U[j,i]*U[j,i]
               + V[js,i]*V[js,i] + V[j,i]*V[j,i])
          end let
     end for
     end let
end for
end let
end function
```

```
% ----- Potential -----
function     Potential (P, U, V:     GridVariables;
                      delta:       real
             returns    GridVariables)

let
     fsdx : real    := 4.0 / delta;
     fsdy : real    := 4.0 / delta;
     M    : integer := array_limh(P);
in
     for j in 0,M
     returns array of
          let jn := if j=0 then M else j-1 end if in
          for i in 0,M
          returns array of
               let iw := if i=0 then M else i-1 end if in
                    (fsdx*(V[j,i]-V[j,iw])-fsdy*(U[j,i]-U[jn,i]))/
                    (P[jn,iw]+P[jn,i]+P[j,iw]+P[j,i])
               end let
          end for
          end let
     end for
end let

end function

% ----- Time Step -----
function     TimeStep (deltat, delta:                real;
                     Psmooth, Usmooth, Vsmooth:
GridVariables;
                     Uflux, Vflux, H, PotVel:      GridVariables
             returns   GridVariables, GridVariables,
GridVariables)

let
     deltat_8 : real    := deltat / 8.0;
     deltat_d : real    := deltat / delta;
     M        : integer := array_limh(Psmooth);

in
     for j in 0,M
     returns array of
          let js := if j=M then 0 else j+1 end if in
          for i in 0,M
          returns array of
               let ie:= if i=M then 0 else i+1 end if in
                    Psmooth[j,i]-deltat_d*(Uflux[j,ie]-Uflux[j,i])
                    -deltat_d*(Vflux[js,i]-Vflux[j,i])
               end let
          end for
          end let
     end for,
```

```
      for j in 0,M
      returns array of
           let jn := if j=0 then M else j-1 end if;
               js := if j=M then 0 else j+1 end if
           in
           for i in 0,M
           returns array of
                let iw := if i=0 then M else i-1 end if;
                    ie := if i=M then 0 else i+1 end if
                in
                     Usmooth[j,i]+
                     deltat_8*(PotVel[js,i]+PotVel[j,i])
                     *(Vflux[js,i] +Vflux[js,iw]
                     +Vflux[j,iw]+Vflux[j,i])
                     -deltat_d*(H[j,i]-H[j,iw])
                end let
           end for
           end let
      end for,

      for j in 0,M
      returns array of
           let jn := if j=0 then M else j-1 end if;
               js := if j=M then 0 else j+1 end if
           in
           for i in 0,M
           returns array of
                let iw := if i=0 then M else i-1 end if;
                    ie := if i=M then 0 else i+1 end if
                in
                     Vsmooth[j,i]-
                     deltat_8*(PotVel[j,ie]+PotVel[j,i])
                     *(Uflux[j,ie]+Uflux[j,i]
                     +Uflux[jn,i]+Uflux[jn,ie])
                     -deltat_d*(H[j,i]-H[jn,i])
                end let
           end for
           end let
      end for
end let

end function

% ----- Smooth -----
function     Smooth (alpha:             real;
                     X, Xsmooth, Xnext: GridVariables
             returns GridVariables)

let M := array_limh(X) in
   for j in 0,M cross i in 0,M
   returns array of
        X[j,i] + alpha * (Xnext[j,i]-2.0*X[j,i]+Xsmooth[j,i])
   end for
end let
end function
```

```
% ----- Shallow -----
function Shallow (iter:                              integer;
                  P, Psmooth, U, Usmooth, V, Vsmooth:
GridVariables;
                  alpha, deltat:                     real;
                  alphap, deltatp, delta:            real;
         returns
GridVariables)

if (iter = 0) then
     P
else
     let
          newP, newPsmooth, newU, newUsmooth, newV, newVsmooth :=
          let
               Uflux  : GridVariables;    % Mass flux in E/W.
               Vflux  : GridVariables;    % Mass flux in N/S.
               H      : GridVariables;    % The height of fluid.
               PotVel : GridVariables;    % Potential velocity.
               Pnext  : GridVariables;    % Intermediate results.
               Unext  : GridVariables;    % Intermediate results.
               Vnext  : GridVariables;    % Intermediate results.

               Uflux, Vflux := Fluxes     (P, U, V);
               H            := Height     (P, U, V);
               PotVel       := Potential  (P, U, V, delta);
               Pnext, Unext, Vnext
                            := TimeStep   (deltat, delta,
                                           Psmooth, Usmooth,
Vsmooth,
                                           Uflux, Vflux, H,
PotVel)
          in
               Pnext,
               Smooth (alpha, P, Psmooth, Pnext),
               Unext,
               Smooth (alpha, U, Usmooth, Unext),
               Vnext,
               Smooth (alpha, V, Vsmooth, Vnext)
          end let;
     in
          Shallow (iter-1,
                  newP, newPsmooth,
                  newU, newUsmooth,
                  newV, newVsmooth,
                  alphap, deltatp, alphap, deltatp, delta)
     end let
end if
end function
```

```
% ----- Main Program -----
function     Main   (lastiter:      integer
             returns array [real])

let
     M := 31;
     p :=
     let
          % Constants:
          delta   : real := 1.0E5;        % Grid spacing.
          deltatp : real := 180.0;        % Time step size in
seconds
                                          % after first iteration.
          deltat  : real := deltatp/2.0;  % Time step size in
seconds
                                          % for 1st iteration.
          alphap  : real := 0.001;        % Time filtering
parameter
                                          % after first iteration.
          alpha   : real := 0.0;          % Time filtering
parameter
                                          % for first iteration.

          % Initial Values:
          P, U, V := Initialize (M, delta);
          Psmooth := P;
          Usmooth := U;
          Vsmooth := V
     in
          Shallow (lastiter, P, Psmooth, U, Usmooth, V, Vsmooth,
                   alpha, deltat, alphap, deltatp, delta)
     end let
in
    for i in 0,M                           % Diag elements of P.
    returns array of P[i,i]
    end for
end let
end function
```

B.3.9. Ulam

<u>Sisal Code:</u>

```
function   ulam   (n, max, cnt: integer returns integer, integer)

if (n = 1) then
   max, cnt
else
   let
      newn := if (mod(n,2) = 0) then
                 n / 2
              else
                 n * 3 + 1
              end if
   in
      ulam (newn, max(newn,max), cnt+1)
   end let
end if
end function

function   Main   (returns integer, integer)

let
   n   := 27;
   max := 1;
   cnt := 1
in
   ulam (n, max, cnt)
end let
end function
```

Appendix C: Details of Experiments

This appendix contains the details of experiments reported in Chapters 6 and 7. The section headings are the same as those used in Chapters 6 and 7.

C.1. Effectiveness of Local By-pass Queue

C.1.1. Method

The three most diverse codes in the benchmark suite (Mm, Queens, and Shallow) are used in this experiment. For Queens, a 6x6 board is used; for Mm, a 32x32 pair of matrices are used; and for Shallow, the grid size is 16x16 and there are 6 iterations. These problem sizes are large enough to saturate the single processor on which they are run. The Short Stay Matching Store is disabled for this experiment in a effort to isolate the effect of the by-pass queue.

Three tests are run for each code. The first is a base case, in which the local by-pass is completely disabled. In the later cases, the local by-pass is set to process one and two tokens per cycle.

It should be noted that the instruction fetch strategy used takes locally by-passed tokens in preference to tokens from outside the processing element.

C.1.2. Measurements

Two measurements are taken in this experiment, the execution time of each program and the percentage of all by-pass tokens that take the local by-pass. Note that all by-pass tokens could be processed locally if the queue length were infinite and could accept every token that was generated in any single cycle.

The length of the local by-pass queue is set at 64.

C.1.3. Assumptions

A saturated processor, in a multiprocessor, will behave similarly to a saturated uni-processor. Therefore, design decisions made in this experiment will apply to larger systems.

The effect of matching store performance is minimal. The parameterization of the matching store is set to be four times what is required. This is done to isolate the processing element from the rest of the processor.

These programs are large enough to saturate a single processor, and therefore, the results generalize to larger problems on a single processor.

C.1.4. Additional Results

The local by-pass queue length is not a factor. Only in one case did it grow beyond 4 tokens, and then only to 10 tokens.

C.2. Design of the Short Stay Matching Store

C.2.1. Method

The three primary test programs are used (Mm: 32x32, Queens 6x6, and Shallow:16x16-6). Since there are three on chip memories required for the SDFA architecture,[99] and modern chip designs allow about 16 KBytes [DEC92a, Inmo91a]. The largest the SSMS could be is about 8 KBytes, which represents approximately 512 tokens.[100] Also, it is clear that the larger this memory is the more effective it will be. This is tested using the rectifying SSMS with a fixed delay of 64 and varying sizes for the SSMS.

To determine an effective delay, the delay in the SSMS is set at 16, 32, 64, and 128 cycles, and the test programs run for both types of SSMS.[101] The final data from Chapter 6.1 serves as a base case for comparisons. The size of the SSMS for these experiments is 512.

The instruction fetch strategy takes locally matched tokens first, followed by locally by-passed tokens, followed by tokens from outside the processing element.

C.2.2. Measurements

The data needed, in all cases, are the execution time, the number of local extract operations, the number of local extract/clear operations, the number of tokens in the SSMS, the length of the SSMS input queues, and the length of the local by-pass queue.

C.2.3. Assumptions

A saturated processor, in a multiprocessor, will behave similarly to a saturated uni-processor. Therefore, design decisions made in this experiment will apply to larger systems.

The effect of matching store performance is minimal. The parameterization of the matching store is set to be four times what is required. This is done to isolate the processing element from the rest of the processor.

These programs are large enough to saturate a single processor, and therefore, the results generalize to larger problems on a single processor.

---

99 These are the SSMS, the Recursive Tag Stack memory, and an instruction cache.

100 This assumes a 64 bit data field and 64 bit tag field. The later is probably optimistic.

101 The total pipeline length of a single processor is approximately 23 to 32 stages depending on the design of the main matching store. These delay values span that range and to wait much longer could be wasteful in sequential computations.

C.2.4. Additional Results

Even in the presence of the SSMS, the local by-pass queue does not fill up significantly. The averages range from about 1.2 to 1.5 tokens depending on program and SSMS parameters. The maximum length for all experiments was 22 tokens. In light of this, its length has been set to 32.

The following table contains the data obtained by varying the size of the SSMS, with the delay fixed at 64, and using the rectifying strategy. The data represents the percentage of all matches which were performed locally and the execution time in milliseconds.

	Percent Local			Execution Time		
Size	Mm	Queens	Shallow	Mm	Queens	Shallow
64	1.64%	46.66%	9.90%	0.874	0.946	0.914
128	1.77%	49.24%	11.16%	0.876	0.940	0.938
256	1.87%	50.23%	12.54%	0.870	0.941	0.917
512	2.41%	52.88%	13.64%	0.872	0.937	0.915

The following table contains the execution times for each program, SSMS type, and delay time. Times are in milliseconds.

	Simple			Rectifying		
Delay	Mm	Queens	Shallow	Mm	Queens	Shallow
16	0.877	0.938	0.927	0.868	0.934	0.912
32	0.878	0.941	0.930	0.875	0.934	0.929
64	0.873	0.946	0.911	0.872	0.937	0.915
128	0.893	0.960	0.916	0.873	0.945	0.919

Recall that the times for the same programs without SSMS were:
Mm = 0.881 ms, Queens = 0.953 ms, and Shallow = 0.940 ms.

C.3. Tuning the Function Unit Queues

C.3.1. Method

The three primary test programs are used, (Mm: 32x32, Queens 6x6, and Shallow:16x16-6). Using parameters within the simulator, all the queue lengths can be set to large values and histograms of the distribution in queue length produced. The maximum size used is 128 slots.

Once an optimum value is chosen, it is then entered into the architectural configuration file and the performance compared to the version with long queues. This provides an indication of how close the chosen value, for queue length, is to the best possible. All tests are run on a uni-processor.

C.3.2. Measurements

In the first part of this experiment, the number of cycles, that a given queue length was observed, is recorded and histograms produced. The maximum, average, and standard deviation of each value are also computed, providing a numerical equivalent to the histogram. This data is provided directly by the simulator. In the second part of this experiment, the execution time is used.

C.3.3. Assumptions

A saturated processor, in a multiprocessor, will behave similarly to a saturated uni-processor. Therefore, design decisions made in this experiment will apply to larger systems.

The effect of matching store performance is minimal. The parameterization of the matching store is set to be four times what is required. This is done to isolate the processing element from the rest of the processor.

These programs are large enough to saturate a single processor, and therefore, the results generalize to larger problems on a single processor.

C.3.4. Additional Results

The Following tables are extracted from the simulator output and contain the numerical form of the data from this experiment. The histograms themselves provided little, or no, additional information and have, therefore, been omitted.

Results for Mm		Maximum	Average	Std. Dev.
Length of Call/Return input	:	0	0.000	0.00
Length of Div/Sqrt input	:	0	0.000	0.00
Length of FPU input queue	:	30	2.025	4.04
Length of IPU input queues	:	2	1.480	0.70
Length of Call/Return output	:	0	0.000	0.00
Length of Div/Sqrt output	:	0	0.000	0.00
Length of FPU output queue	:	44	0.614	2.31
Length of IPU output queues	:	19	2.001	1.30

Results for Queens		Maximum	Average	Std. Dev.
Length of Call/Return input	:	2	0.035	0.18
Length of Div/Sqrt input	:	15	0.870	1.52
Length of FPU input queue	:	0	0.000	0.00
Length of IPU input queues	:	256	232.410	60.12
Length of Call/Return output	:	2	0.035	0.18
Length of Div/Sqrt output	:	3	0.173	0.49
Length of FPU output queue	:	0	0.000	0.00
Length of IPU output queues	:	256	238.378	45.56

Results for Shallow		Maximum	Average	Std. Dev.
Length of Call/Return input	:	3	0.077	0.28
Length of Div/Sqrt input	:	4	0.068	0.28
Length of FPU input queue	:	24	2.116	2.92
Length of IPU input queues	:	2	1.189	0.76
Length of Call/Return output	:	2	0.073	0.26
Length of Div/Sqrt output	:	2	0.010	0.10
Length of FPU output queue	:	4	0.708	0.49
Length of IPU output queues	:	142	2.351	5.64

C.4. Super Scalar and CIS Study

C.4.1. Method

The three primary test programs (Mm: 32x32, Queens 6x6, and Shallow:16x16-6) are executed using all four possible combinations of instruction-set complexity and multiple issue. These tests are all run on a single processor. In order to establish an optimal combination, both execution time and efficiency of execution need to be considered.

C.4.2. Measurements

There are four measurements taken for each test: the time to execute; the average number of instructions executed, in a single cycle; the average number of operations performed, in a single cycle; and the number of operations that can be executed, on average, as part of a single complex instruction. An "operation" here refers to a primitive instruction of the basic SDFA instruction-set. Therefore, for the simple instruction tests, each instruction involves one operation, and, in the complex instruction tests, an instruction can perform 1, 2, or 3 operations.

The average number of instructions issued per cycle is used as a measure of the efficiency of the processor.[102]

C.4.3. Assumptions

A saturated processor, in a multiprocessor, will behave similarly to a saturated uniprocessor. Therefore, design decisions made in this experiment will apply to larger systems.

The effect of matching store performance is minimal. The parameterization of the matching store is set to be four times what is required. This is done to isolate the processing element from the rest of the processor.

These programs are large enough to saturate a single processor, and therefore, the results generalize to larger problems on a single processor.

The complex instruction-set emulation is based on dynamic examination of instructions as they are processed by the processing element and is, therefore, not fully integrated. This is likely to produce less efficient execution than if code were generated specifically for a complex instruction-set.

---

102 Efficiency in parallel systems is difficult to measure. This technique assumes that the instruction issue rate has the greatest impact on utilization. Further discussion of the pitfalls of measuring efficiency can be found in [Snel91a, Hock92a].

C.4.4. Additional Results

Figure C.1. shows the number of operations performed, on average, in a single cycle.

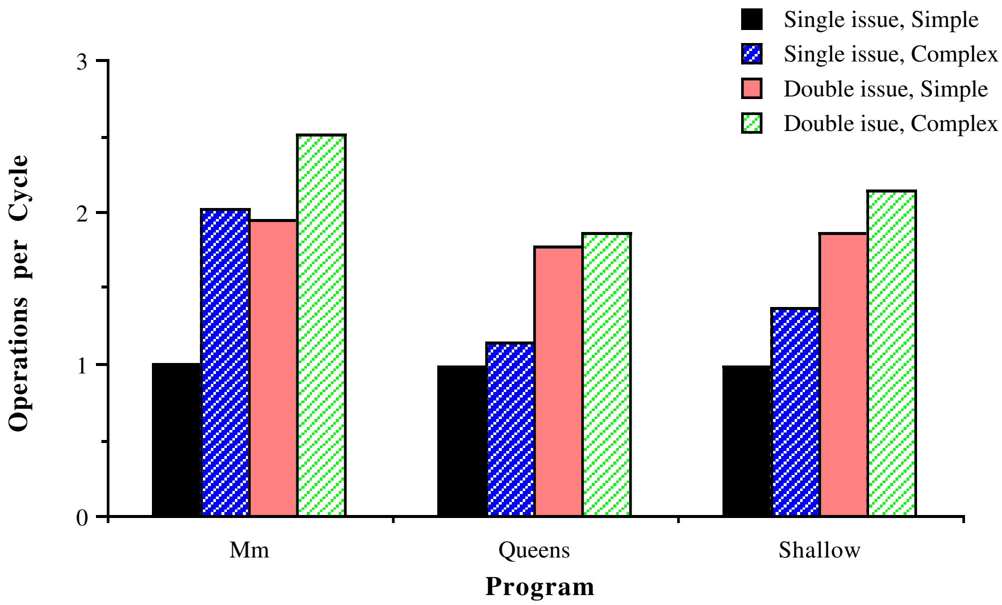


Figure C.1: Operations per cycle for various instruction fetch strategies.

Figure C.2. indicates the efficiency of the various approaches, measured in average instructions per cycle.

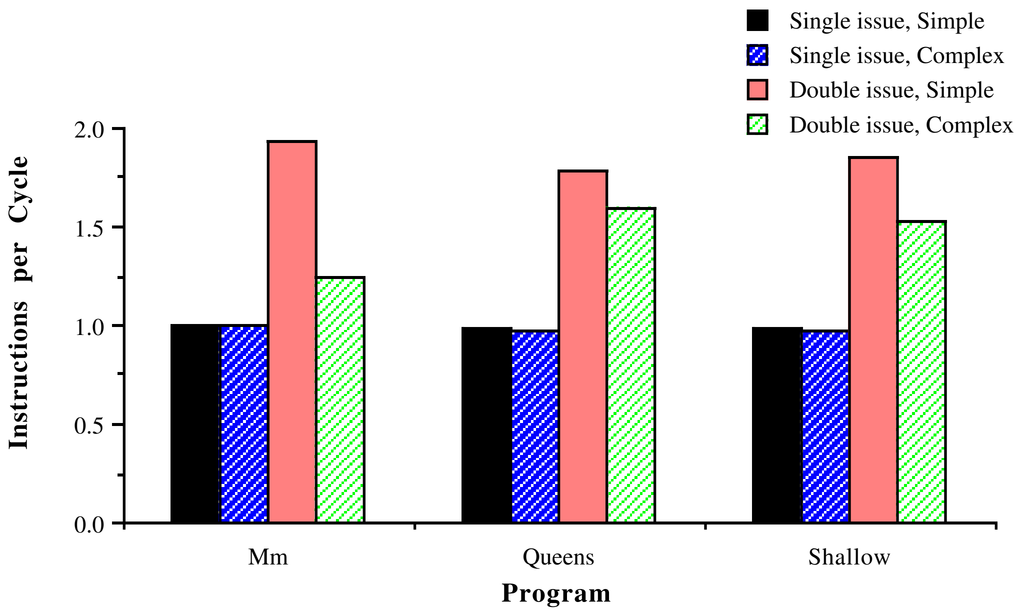


Figure C.2: Instructions per cycle for various instruction fetch strategies.

The following table indicates the success of the complex instruction-set emulation used in these experiments.

	Instructions per Cycle		
Program	Maximum	Average	Std. Dev.
Mm	3.0	2.021	0.71
Queens	3.0	1.172	0.38
Shallow	3.0	1.398	0.71

C.5. Sequential Bottle-neck Study

C.5.1. Method

Three test programs are used which exhibit sequential behaviour: linearly recursive factorial (Facl), tail recursive factorial (Fact), and an implementation of a test for Ulam's conjecture (Ulam). The *ideal parallelism*[103] of these three programs is 1.0 for Facl, 1.6 for Fact, and 2.1 for Ulam.

These programs are run on three different configurations of uni-processor SDFA systems. First, both the local by-pass queue and the SSMS are disabled. Then, only the local by-pass queue is enabled. And last, both the local by-pass and the SSMS are enabled.

C.5.2. Measurements

Execution time is measured, and the percentage improvement, over the first case, is computed. This normalizes the effect of the locality mechanism over the different execution times of the various programs.

C.5.3. Assumptions

The effect of matching store performance is minimal. The parameterization of the matching store is set to be four times what is required. This is done to isolate the processing element from the rest of the processor.

Since parallelism is low, the frequency of multiple issues in the instruction fetch will be small. Thus, the super scalar aspect of the processing element should not effect the results.

C.6. The Need for a Throttle Mechanism

C.6.1. Method

The usual test programs (Shallow, Mm, and Queens) are run for a variety of problem sizes. The amount of memory required to run each of these programs sequentially is known, as a function of problem size. This represents a minimum space requirement. On a single processor SDFA machine, additional memory would be required to keep the pipelines full. Once the machine is saturated, the amount of extra memory required

---

103 This is measured in the interpreter and assumes an infinite number of processors, one cycle instructions, and zero time to transmit and match tokens. Its units are instructions per cycle.

should not increase more rapidly than this minimum requirement. Some increase, however, is to be expected. If this increase is "excessive", a store explosion has occurred.

C.6.2. Measurements

The two memories in the system used to store temporary results are the matching store and the token queue. There are two possible measurements of memory use, the maximum occupancy and the average occupancy. Both are measured every cycle. In the results below, both measurements and both system components are used to determine the extent of the store explosions.

The formulas for sequential memory requirement ($S_{application}$) are given below, where n is the problem size:[104]

$$S_{mm}(n) \approx 3 * n^2, \qquad S_{shallow}(n) \approx 16 * n^2, \qquad S_{queens}(n) \approx 2 * n^2.$$

C.6.3. Assumptions

The effect of matching store performance is minimal. The parameterization of the matching store is set to be four times what is required. This is done to isolate the processing element from the rest of the processor.

The saturation assumption from earlier experiments does not apply directly in this case. In a multiprocessor system, the same effect will be seen, but it will not occur until larger problems are run.

---

104 These ideals ignore any scalars used in control and partial calculations, as they remain constant with respect to n. In Mm and Shallow, this is not a problem, but the Queens ideal, because of the small problem sizes, is likely to be optimistic.

C.6.4. Additional Results

Figure C.3. shows the sum of token queue and matching store occupancy, and the memory required in an ideal sequential implementation, for Mm. Both maxima and averages are shown.

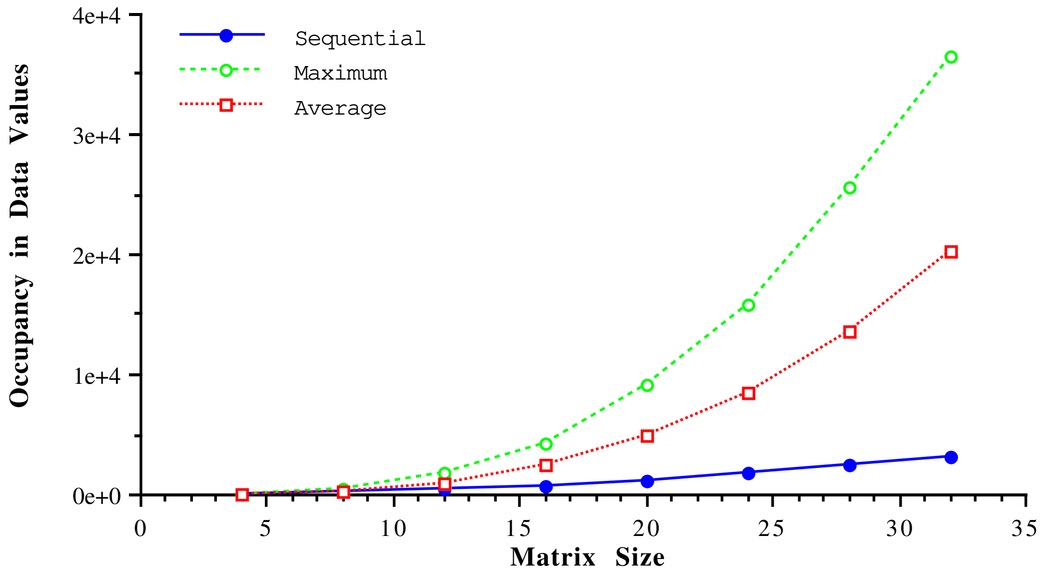


Figure C.3: Store explosion in Mm.

Figure C.4. shows the sum of token queue and matching store occupancy, and the memory required in an ideal sequential implementation, for Queens. Both maxima and averages are shown. Notice that Queens is presented on a logarithmic graph.

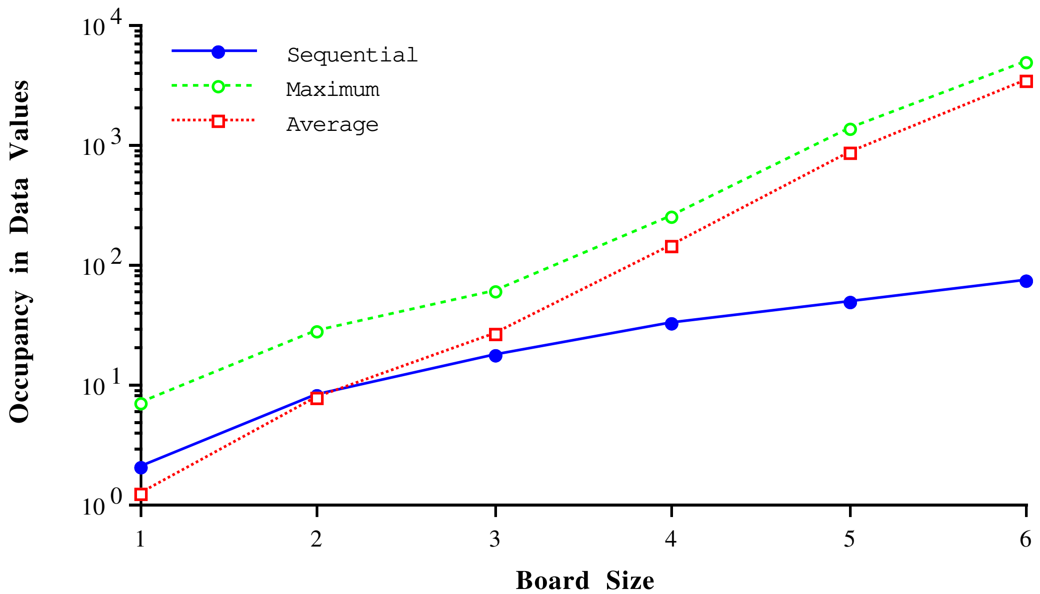


Figure C.4: Store explosion in Queens.

C.7. Software Throttling

C.7.1. Method

Shallow is run for 6 time steps on a 16x16 grid, and the occupancies of the system's bulk memories are measured.

C.7.2. Measurements

The execution time and maximum and average level of store use are measured for both the token queue and the matching store.

C.7.3. Assumptions

The effect of matching store performance is minimal. The parameterization of the matching store is set to be four times what is required. This is done to isolate the processing element from the rest of the processor.

This program is large enough to saturate a single processor, and therefore, the results generalize to larger problems on a single processor.

The multiprocessor saturation assumption, does not apply in this case, since these techniques can create temporary bottle-necks, and, therefore, the system does not run at saturation all the time.

C.7.4. Additional Results

This technique did not, however, significantly increase the number of tokens processed locally. It increased from 49.5% to 50.4% (K2-Bounded) and to 50.8% (strict).

C.8. Hardware Throttling

C.8.1. Method

The three primary test programs (Mm: 32x32, Queens 6x6, and Shallow:16x16-6) are run on a uni-processor system with and without the throttling token queue. The execution time in each case is compared with store usage and percent local computation.

C.8.2. Measurements

The execution time and maximum and average level of store use are measured for both the token queue and the matching store. The percent of tokens processed locally is also measured as an alternative indicator of efficiency of execution.

C.8.3. Assumptions

The effect of matching store performance is minimal. The parameterization of the matching store is set to be four times what is required. This is done to isolate the processing element from the rest of the processor.

The saturation assumption, from earlier experiments, does not apply in this case, since this technique can create temporary bottle-necks; therefore, the system does not run at saturation all the time.

The sorting function, as implemented, performs an exact sort. This is likely to be impractical in an implementation,. However, a small number of tokens could be sorted on the fly. If the token queue generally holds large numbers of tokens, this would not be the case, and a partial sort would be needed.

C.9. Matching Store Banking Study

C.9.1. Method

The three primary test programs (Mm: 32x32, Queens 6x6, and Shallow:16x16-6) are run for varying numbers of banks using three machine configurations. In the first case, the size of level one of the matching store is large enough for all problems that can reasonably be simulated; therefore, the first test assumes a full size SSMS and a full size level one matching store. This will dictate an absolute minimum number of banks. The second case assumes a full size level one store, but no SSMS. And the third case assumes only a level two store. In what follows, these cases are referred to as "Full", "Level 1 Only", and "Level 2 Only".

C.9.2. Measurements

The execution time in each case is plotted against the number of banks. Based on this data an optimum number of banks is chosen.

C.9.3. Assumptions

A saturated processor, in a multiprocessor, will behave similarly to a saturated uni-processor. Therefore, design decisions made in this experiment will apply to larger systems.

These programs are large enough to saturate a single processor, and therefore, the results generalize to larger problems on a single processor.

C.9.4. Additional Results

Figure C.5. shows the execution time of Mm for varying numbers of matching store banks in all three machine configurations.

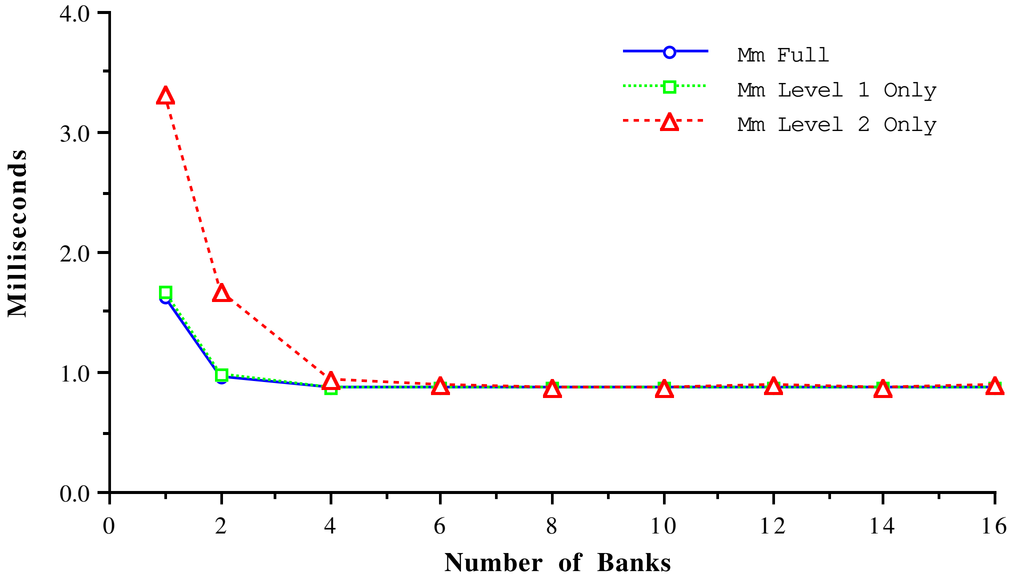


Figure C.5: The effect of varying the number of MS banks on Mm.

Figure C.6. shows the execution time of Queens for varying numbers of matching store banks in all three machine configurations.

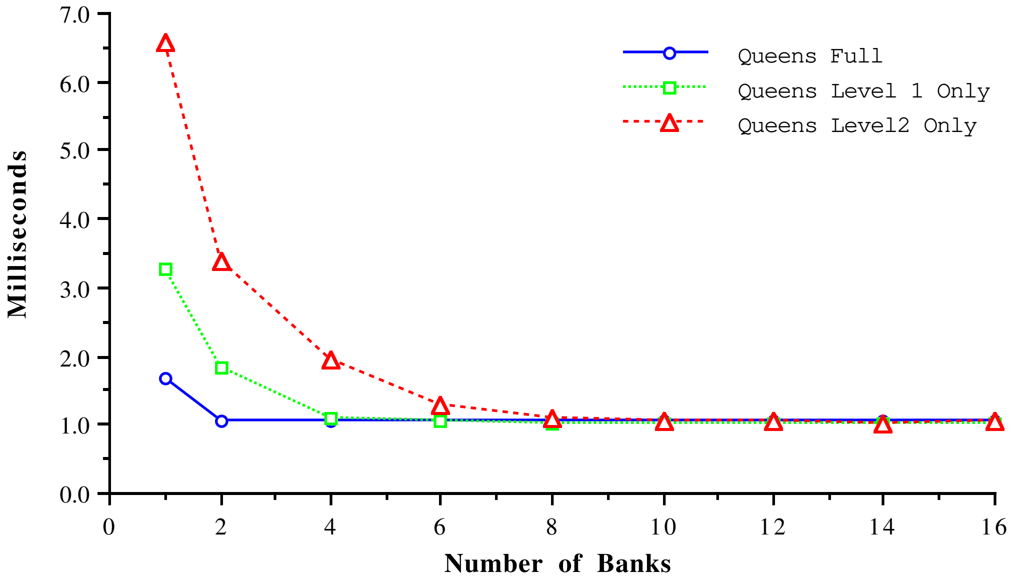


Figure C.6: The effect of varying the number of MS banks on Queens.

C.10. Matching Store Cache Study

C.10.1. Method

The three primary test programs are used (Mm: 32x32, Queens 6x6, and Shallow:16x16-6) in this experiment. The size of the Level 1 matching store bank is set at a value, different for each application, that will force the use of the Level 2 store. This figure is computed from the Level 1 average occupancy in the 8 bank, fully configured case from Chapter 6.9. Approximately half of the average bank occupancy is used as the size of a Level 1 bank. These values are given in the table below:

Program	Average Bank Occupancy	Level 1 Bank Capacity Used
Mm	1041	1024
Shallow	972	1024
Queens	180	128

The threshold parameter, which governs the level at which migration begins, is varied for each test. The threshold parameter is the percentage of maximum capacity at which tokens are migrated up to Level 2. In this experiment, the threshold parameter is set to values ranging from 100% to 80%.

C.10.2. Measurements

The time for each program is measured, however, this shows little variation, since the number of banks is set conservatively. In addition, the percentage of matches, occurring in Level 1, is also computed.[105] This provides a direct indication of the effectiveness of the caching mechanism.

C.10.3. Assumptions

A saturated processor, in a multiprocessor, will behave similarly to a saturated uniprocessor. Therefore, design decisions made in this experiment will apply to larger systems.

These problems are small and the artificial reduction in the size of Level 1 is likely to distort the results.

C.11. Spatial Locality in a Multiprocessor

C.11.1 Method

Since the extent of spatial locality, and not its effect on performance, is being studied, the interpreter is used in this experiment. The three main test programs are used. The sizes are 32x32 for Mm, 6x6 for Queens, and Shallow is enlarged to 32x32.and the number of time steps reduced to 4. The reason for this is as follows. The size of problem which can be simulated is limited, and the locality mechanism removes several of the bits which would normally be used to distribute work in a multiprocessor. Therefore, these tests are run on a 16 processor system using a *distribution factor* (see

105 Note that SSMS matches are not used in this calculation, and the percentage is of those matches occurring in the main matching store. Note that in all cases, the number of SSMS matches is basically constant.

Experiment 7.2.) which ensures near perfect load balancing for all values of the locality factor. This approach removes the need to consider distribution and spatial locality simultaneously.

C.11.2. Measurements

All that is required is the total number of tokens created and the number which are processed locally.

C.11.3. Assumptions

It is assumed that locality is independent of whatever distribution in workload exists. This is clearly not the case universally. In fact, as noted above, this experiment is contrived so that distribution remains nearly perfect for all cases. However, with smaller problems, or larger machine configurations, inclusion of an additional bit in the locality factor could halve the number of processors involved in the computation.

C.12. Workload Distribution in a Multiprocessor

C.12.1. Method

In this experiment, three simple programs are run on the interpreter for increasing problem size and, where appropriate, the *distribution factor* is varied.[106] The first problem is Queens and needs no elaboration. The second program is a simple matrix times scalar plus matrix computation (Saxpy). The tagging behaviour, with respect to the processor hashing function, of this problem is the same as Shallow and Mm. However, the parallelism is not adequate to saturate a multiprocessor, since the total amount of computation is too small. The third problem is Gaussian elimination (Gauss). Gauss, because the size of the active sub-matrix reduces as the computation proceeds, has poor load balancing characteristics. This truncation causes a progressively sequential bottle-neck,[107] which prevents it from saturating a multiprocessor of reasonable size.

Since the problem size is limited by the simulation environment and locality is high generally, the locality factor is zero in this experiment.

As with the previous experiment, information on workload distribution can be obtained from the interpreter.

C.12.2. Measurements

The measurement used, in this experiment, is the standard deviation in the number of tokens created (with respect to the number of processors), normalized by the total number of tokens created. The formula is given below:

---

106 The distribution factor determines the number of bit positions to shift some of the index fields prior to combining them and hashing for the processor number. The larger the distribution factor is, the more widely the computation can be distributed.

107 It is for this reason that Gauss has not been used more widely in this study. Only impracticably large problems would run effectively. However, these would still be small problems in the non-simulated world.

$$\text{Imbalance} = \frac{1}{T^*\sqrt{P}} \sqrt{\sum_{p=1}^{P} (T_p - T_{bar})^2}$$

where P is the number of processors; T_p is the number of tokens created on processor p; T is the total number of tokens created by the system; T_{bar} is the average number of tokens created by a single processor.

This formulation of workload imbalance is presented in [Snel89a]. It represents the expected variation, between any two processors, in the amount of work performed as a portion of the total work performed.

C.12.3. Assumptions

These results generalize to larger configurations, but with larger problems being required for the same level of balance.

It is assumed that the temporal distribution of workload is uniform throughout the execution of these programs. This appears to be the case in the interpreter runs. However, the timing behaviour in the simulator could be different. Later experiments, which examine the system as a whole, confirm the validity of this assumption for situations where static load balancing is good.

Figure C.7. shows the effect of deeper levels of multiple recursion on load imbalance using Queens. The graphs in figure C.8. show the effect that increasing problem size has on load imbalance in Saxpy, for several distribution factors. Figure C.9. contains the results for Gauss. The distribution factor was set at one bit for this study.

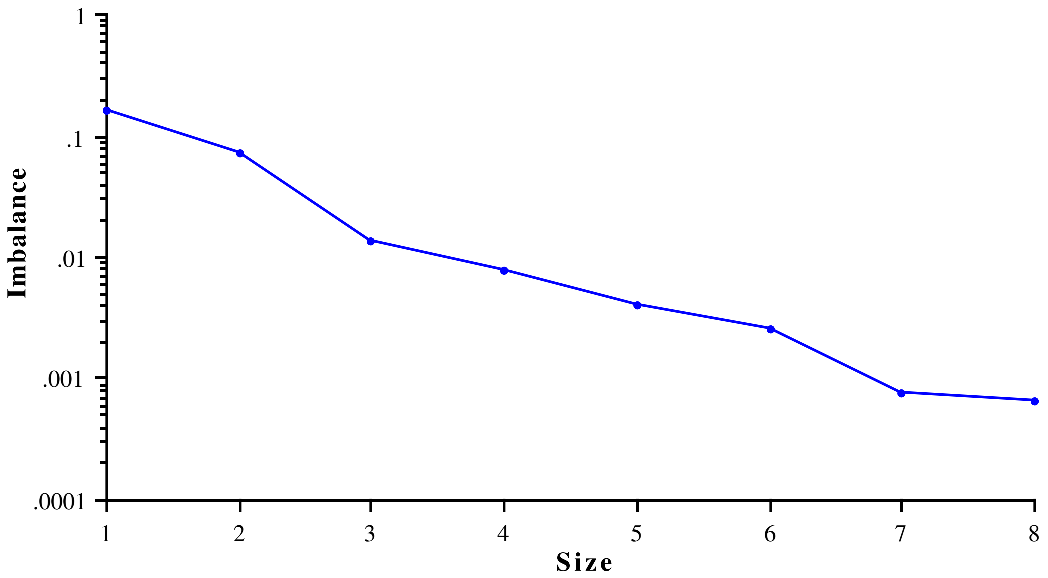


Figure C.7: Load imbalance in Queens.

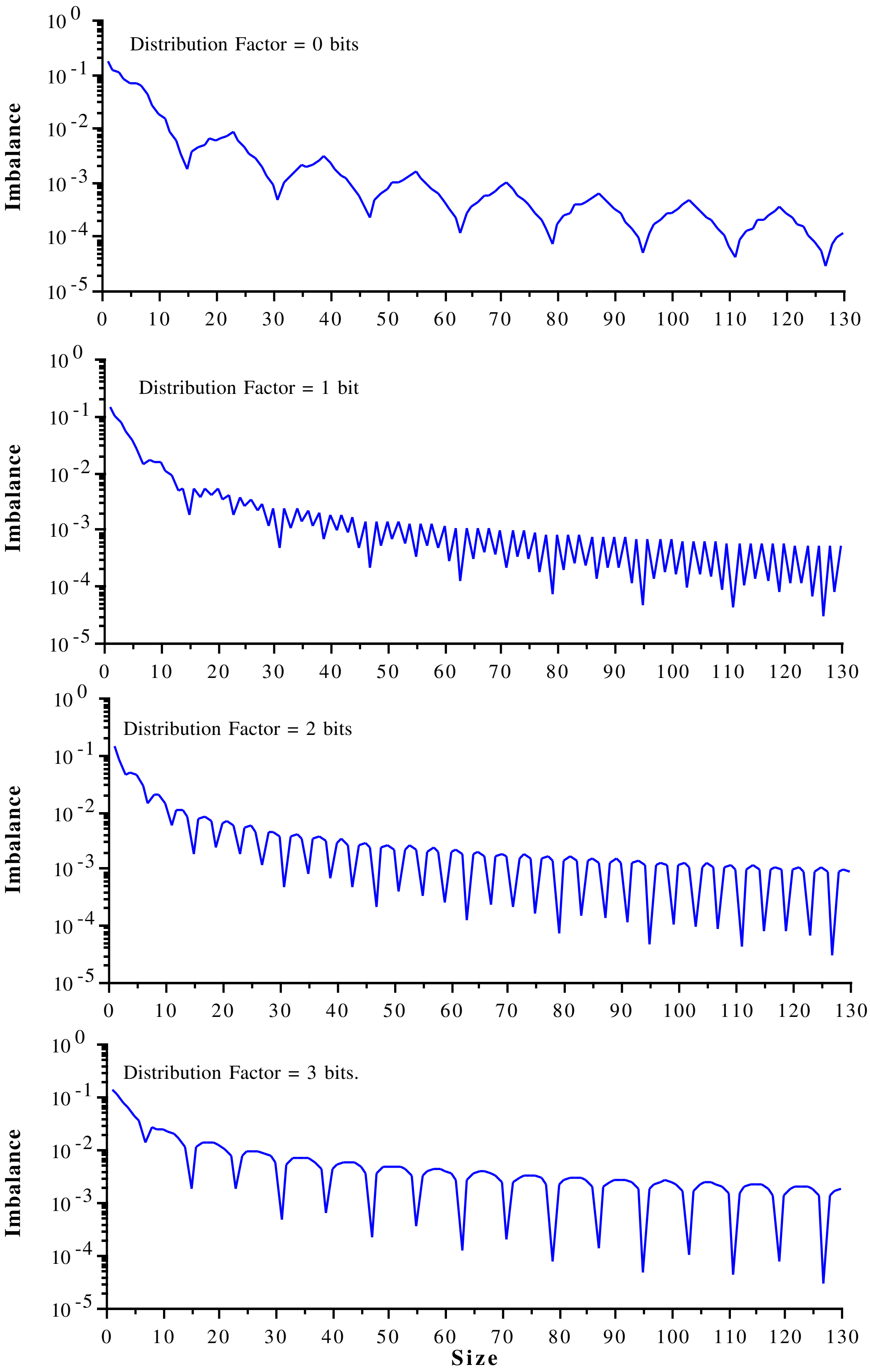


Figure C.8: Load imbalance in Saxpy.

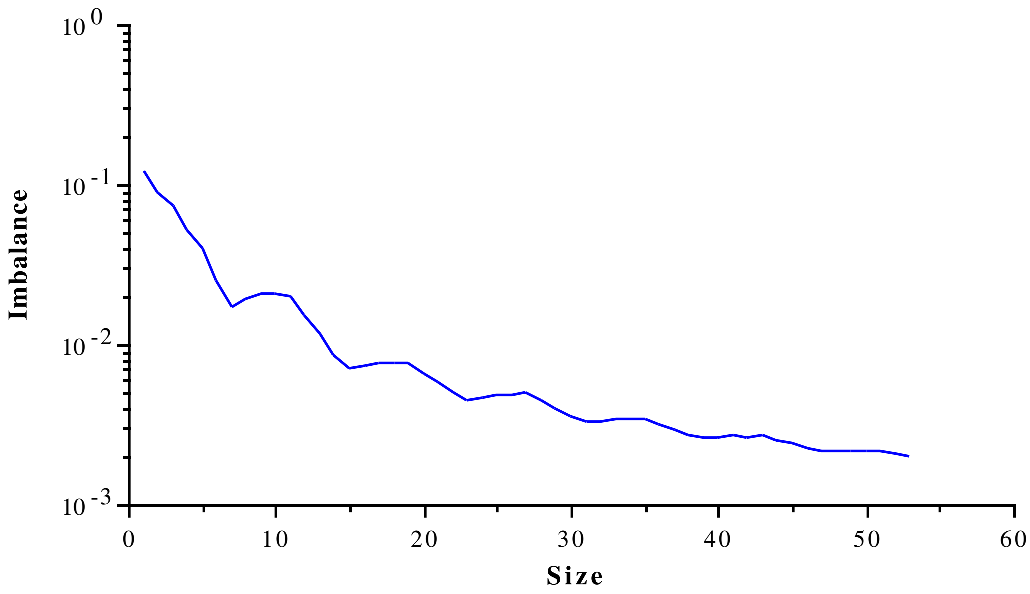


Figure C.9: Load imbalance in Gauss.

C.13. Multiprocessor Network Study

C.13.1. Method

The test programs, Queens (size 6), Mm (size 32), and Shallow (size 32 with 4 time steps), are run on several sized systems and for switching elements with various levels of performance. In this experiment, these switches are parameterized only in terms of the beat of the "pipes" simulating them.[108] Multiples of the switching element's beat vary the bandwidth and latency of the switch, but not the number of "hops" across the switch. The following table shows the bandwidth per processor (in tokens per cycle) and the latency of the network (in cycles) for those values of the switching element beat used in this experiment.

Beat	Per Processor Band Width	4 Proc. Latency	8 Proc. Latency	16 Proc. Latency
1	1.0000	2	3	4
2	0.5000	4	6	8
4	0.2500	8	12	16
8	0.1250	16	24	32
16	0.0625	32	48	64

In this experiment, a locality factor of two and a distribution factor of one are used.

C.13.2. Measurements

Execution time is the only measurement used.

108 Recall that the "pipes" abstraction is used to model the system's timing and that pipes are described in terms of their length and beat.

C.13.3. Assumptions

Only well balanced applications are used in this experiment.

It is assumed that the linking of bandwidth and latency of the switching elements, as described above, does not affect the results of this experiment in general. It is acknowledged that this approach does somewhat limit the design space. Likewise, the design space is limited by the choice of 2x2 switching elements.

C.13.4. Additional Results

Figure C.10. shows the execution time for Mm as a function of the beat parameter in the switch.

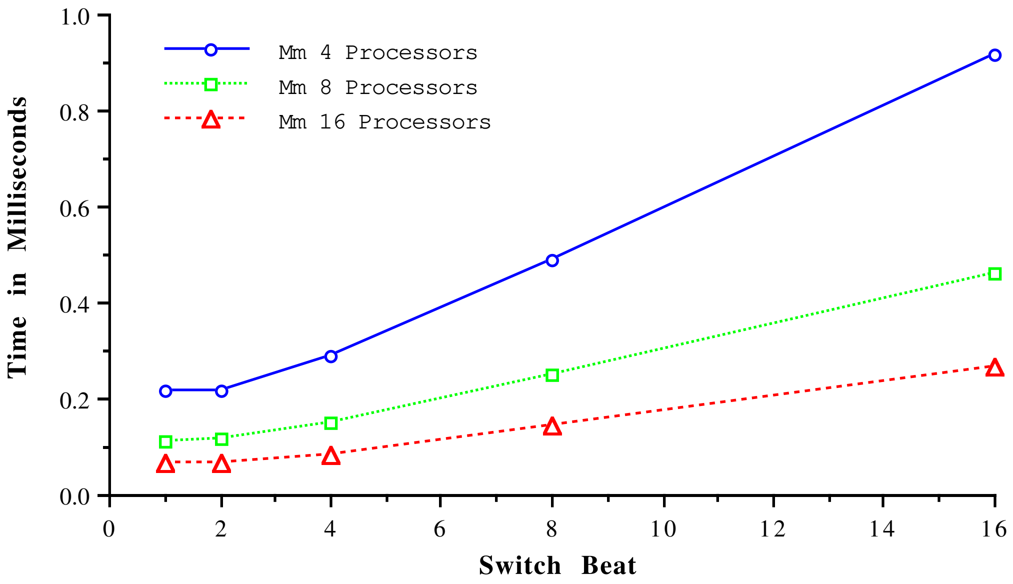


Figure C.10: Mm execution time as function of switch performance.

Figure C.11. shows the execution time for Queens as a function of the beat parameter in the switch.

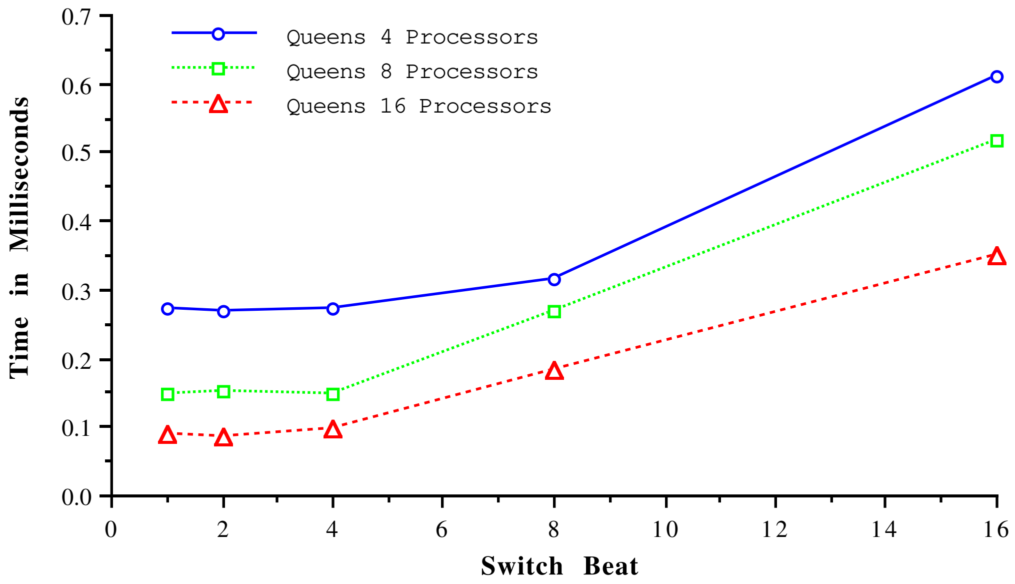


Figure C.11: Queens execution time as function of switch performance.

C.14. Scaling Study

C.14.1. Method

The three primary application programs are run, for several problem sizes, on systems ranging in size from a uni-processor to 64 processors. In order to eliminate simulation restrictions from the study, problem sizes and locality and distribution factors are selected which provide nearly uniform distribution on all systems tested. Recall that as problem size increases, the distribution tends toward uniformity, even for problem sizes which do not match the system, see Chapter 7.2. Therefore, there is no loss of generality in this approach.

Both the locality and distribution factors are set to two. This provides adequate locality in the largest system studied and still provides work for all processors. The problem sizes selected are Mm 16x16, 32x32, and 64x64; Queens 6x6, 7x7 and 8x8; and Shallow 16x16, 32x32, and 64x64 with 4 time steps.

C.14.2. Measurements

Time is measured, but reported as temporal performance (1.0/Time), as recommended by [Hock92a].

C.14.3. Assumptions

The assumptions, made in earlier experiments, have been necessary to allow one aspect of the system to be analyzed in isolation. In this experiment, the system is being examined as a whole; therefore, nothing is assumed, except that which is beyond the scope of this study.

Recall that the problem sizes have been selected to "fit" the systems being studied.

C.14.4. Additional Results

Figure C.12. contains the temporal performance results for Mm (16x16, 32x32, and 64x64) as a function of increasing number of processors.

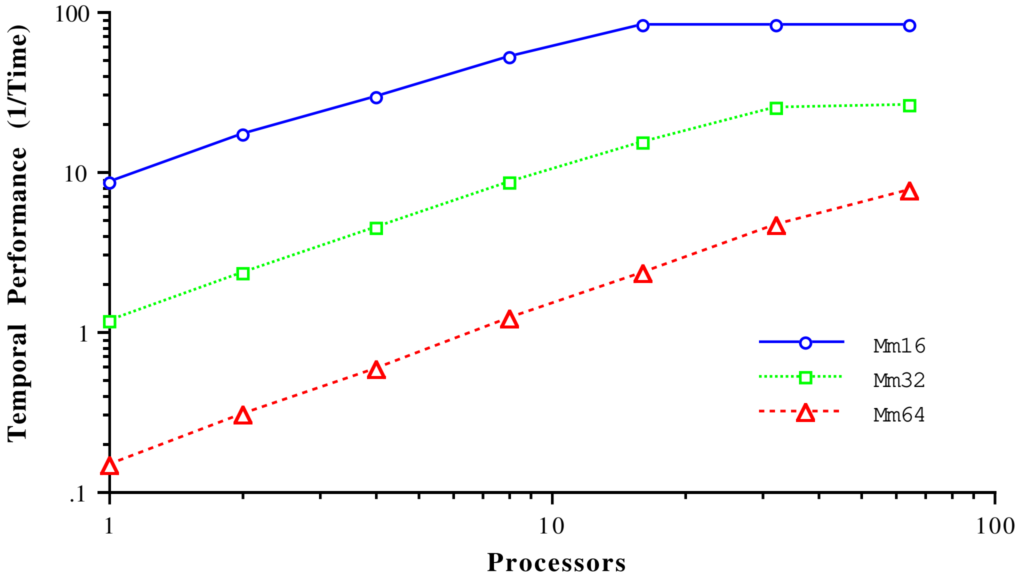


Figure C.12: Scaling performance of Mm.

Figure C.13. contains the temporal performance results for Queens (6, 7, and 8) as a function of increasing number of processors.

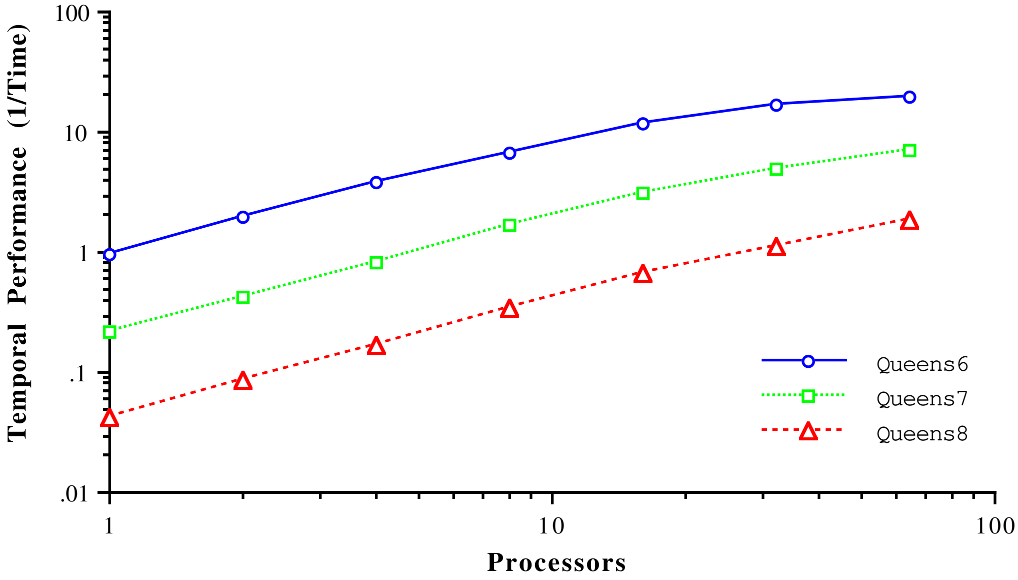


Figure C.13: Scaling performance of Queens.

C.15. Toward Peak Performance

C.15.1. Method

Three versions of each program are run. The first versions are the standard ones used throughout the preceding experiments. In the second group, the proliferating functions have been unrolled, and the last version is the standard version re-written to use two new instructions. The first is a *Proliferate* instruction which performs the function of the Split and Dec instructions in the proliferate function, see Chapter 5, figure 5.7. The second instruction, relevant only to Shallow, performs a circular shift of the index in a single instruction; the original approach requires two instructions. The two new instructions represent only a small increase in the complexity of the processing element.

Shallow is run for 16 iterations on a 32x32 grid, and Mm multiplies two 64x64 matrices. Shallow performs 1091584 Flop, and Mm performs 524288 Flop.

C.15.2. Measurements

Formulas have been created for the number of floating point operations, based on an inspection of the original SISAL code. All types of floating point operations are counted as a single operation, including divide, transcendental functions, and type conversion. These numbers matched the statistics obtained from the simulator.

C.15.3. Assumptions

As in the preceding experiment, the only assumption is the acknowledgement that the problem size has been carefully selected.

Experimental Supplement to The Design and Analysis of a Stateless Data-Flow Architecture

This supplement addresses some of the issues that were not covered in the thesis itself. In particular, the multi-level matching store is re-examined, and a short study of a SDFA system based on the T9000 processor and communications architecture is presented.

S.1. Migration in the Multi-Level Matching Store

The migration of tokens from Level 1 to Level 2, in the matching store, is an effective, cache-like mechanism for exploiting temporal locality.

Since scalability is dependent on the ability of the architecture to exploit a memory hierarchy, the effectiveness of the multi-level matching store must be examined. All the experimental applications, which can be reasonably simulated, fit within the default size of the Level 1 store. Therefore, in order to assess the effectiveness of the migrating mechanism, exceptional circumstances must be created within the system. This is accomplished by reducing the size of the Level 1 store to less than that required by the applications.

The goal of a cache is to provide a hardware facility whereby portions of applications with good temporal locality can run efficiently, even when embedded in programs which do not universally exhibit good temporal locality. The two applications used in this study are Mm and the strict version of Shallow. Both applications are embedded in a program with poor locality.

S.1.1. Method

In this experiment, each application is embedded in a program which initially fills up the Level 1 store, waits for the embedded application to finish, and then clears out this data. Execution of the initial application is delayed until the Level 1 store is full. The size of Level 1 has been reduced to 4K tokens so that Level 2 accesses are required. Also, the number of banks has been reduced to 2, ensuring that the timing effects of Level 2 can be detected in the overall execution time. These tests are all run on a uni-processor with the Short Stay Matching Store disabled. The program parameters are as follows:

In both tests, the data array is a 64x64 matrix, which occupies all of Level 1.

For Shallow, an 8x8 system is executed for 5 time steps.

For Mm, a 16x16 matrix multiply is used.

Note that, without the surrounding program, both Mm and Shallow would execute entirely within the 4K token space provided by Level 1.

Limit cases are established using a full size Level 1 store, in which the whole application executes within the Level 1 store and a zero size Level 1 store, in which the whole application must execute within the Level 2 store. In the remaining tests, the number of free slots maintained in Level 1 is varied between 0 and 8 in steps of 2. Two migration strategies are used: random migration of a token and migration of the oldest token in Level 1. This latter approach is essentially a "least recently used" strategy.[109]

Note that priority is given to migrating tokens over incoming tokens, except when the forwarding queue is full. This implies a change to Chapter 3.5.6. (page 65 of the thesis).[110]

S.1.2. Measurements

In addition to execution time, measurements of the number of extract (matching) actions occurring in Level 1 and 2 are taken at each report step. These measurements are used to compute the percentage of matches occurring in Level 1 and to produce time dependant profiles of matching activity.

S.1.3. Assumptions

A saturated processor, in a multiprocessor, will behave similarly to a saturated uni-processor. Therefore, design decisions made in this experiment will apply to larger systems.

S.1.4. Results

The following table contains a summary of the results obtained in the Mm experiment. The times are in milliseconds.

Size of	Number of	Random Migration		LRU Migration	
Level 1	Free Slots	Time	% L1 Matchs	Time	% L1 Matchs
0	NA	0.3677	0.0%	0.3677	0.0%
4K	0	0.3451	30.5%	0.3451	30.5%
4K	2	0.3255	49.4%	0.3296	47.5%
4K	4	0.3264	51.2%	0.3271	53.0%
4K	6	0.3256	53.2%	0.3233	55.1%
4K	8	0.3200	55.0%	0.3203	56.6%
64K	NA	0.2716	100.0%	0.2716	100.0%

109 Note that, unlike conventional systems, there is little or no extra effort required to maintain track of the oldest token. Since any token can be stored anywhere in the matching store memory, a simple circular buffer structure would ensure that the "oldest" token was always available for migration.

110 It was found during experimentation that giving priority to token migration resulted in a 35% increase in proportion of matches occurring in Level 1 and a 175% increase in the number of migrations. This is due, in part, to small number of banks used in this experiment, which were usually saturated by incoming tokens.

The following table contains a summary of the results obtained in the Shallow experiment. The times are in milliseconds.

Size of Level 1	Number of Free Slots	Random Migration Time	Random Migration % L1 Matchs	LRU Migration Time	LRU Migration % L1 Matchs
0	NA	0.6805	0.0%	0.6805	0.0%
4K	0	0.6454	16.1%	0.6454	16.1%
4K	2	0.4956	78.8%	0.4846	83.1%
4K	4	0.4833	82.9%	0.4781	86.2%
4K	6	0.4870	83.5%	0.4704	88.1%
4K	8	0.4810	84.2%	0.4701	87.8%
64K	NA	0.4349	100.0%	0.4349	100.0%

Figures 1a through 1g show the number of matches occurring in Level 1 and 2 as a function of execution time for Mm using the random migration approach.

Fig. 1a: Mm & No Level 1

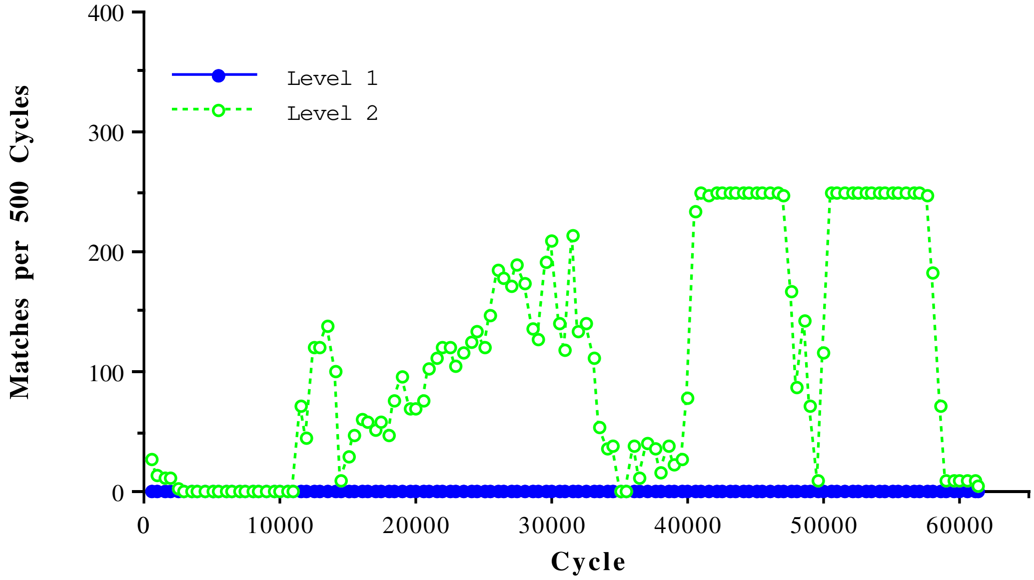


Fig. 1b: Mm, 4K Level 1, & No Free Slots

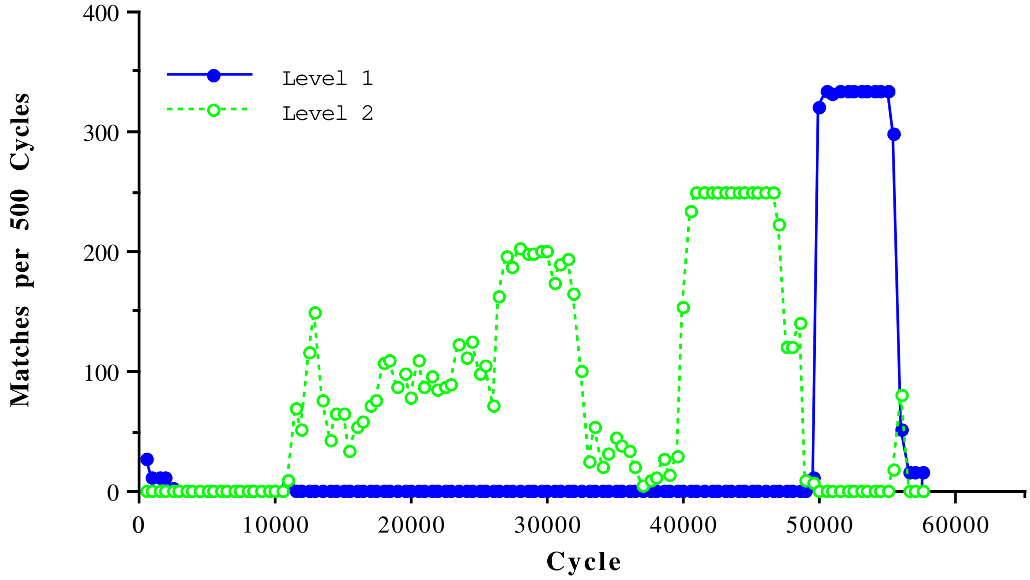


Fig. 1c: Mm, 4K Level 1, & 2 Free Slots

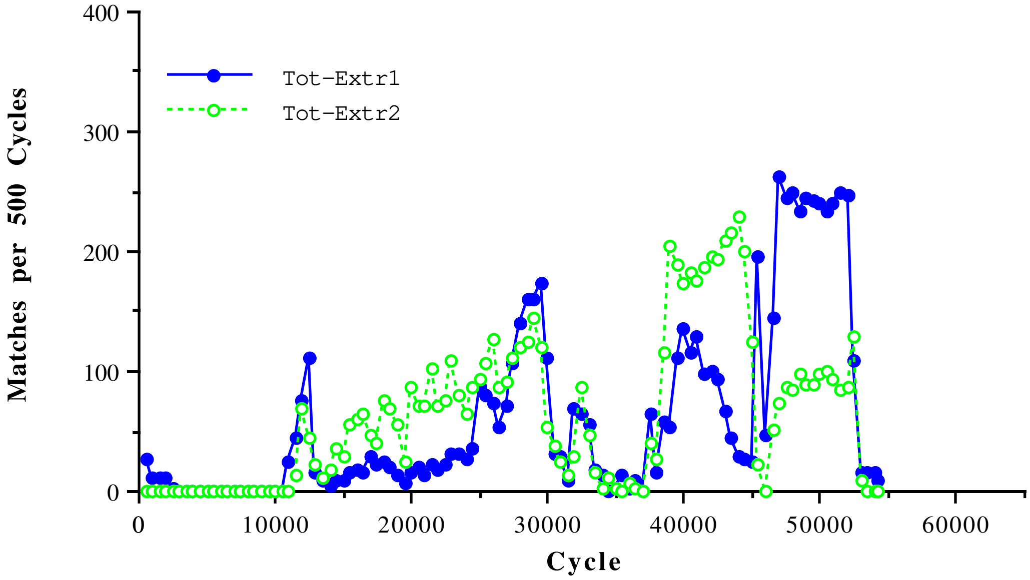


Fig. 1d: Mm, 4K Level 1, & 4 Free Slots

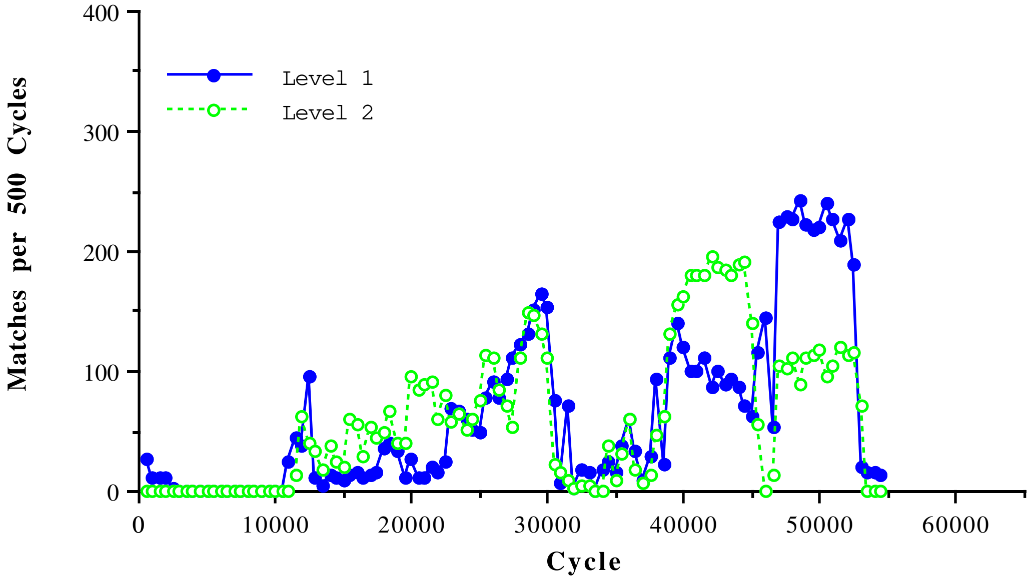


Fig. 1e: Mm, 4K Level 1, & 6 Free Slots

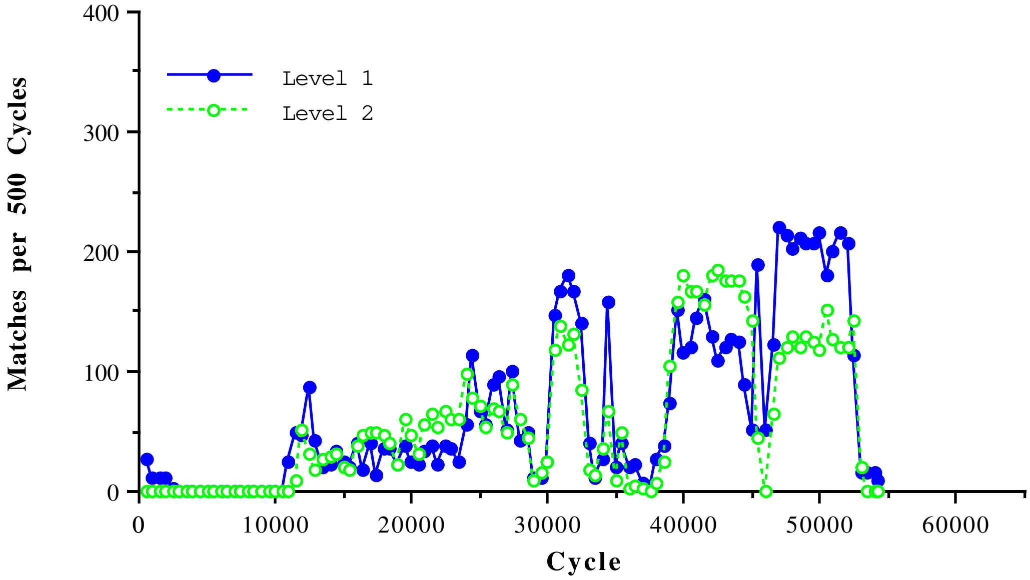


Fig. 1f: Mm, 4K Level 1, & 8 Free Slots

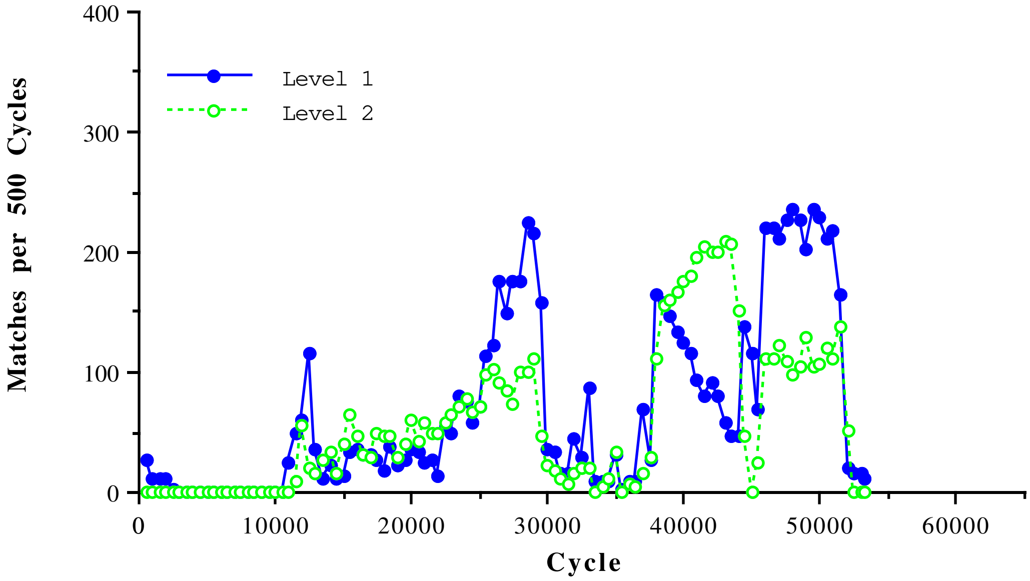
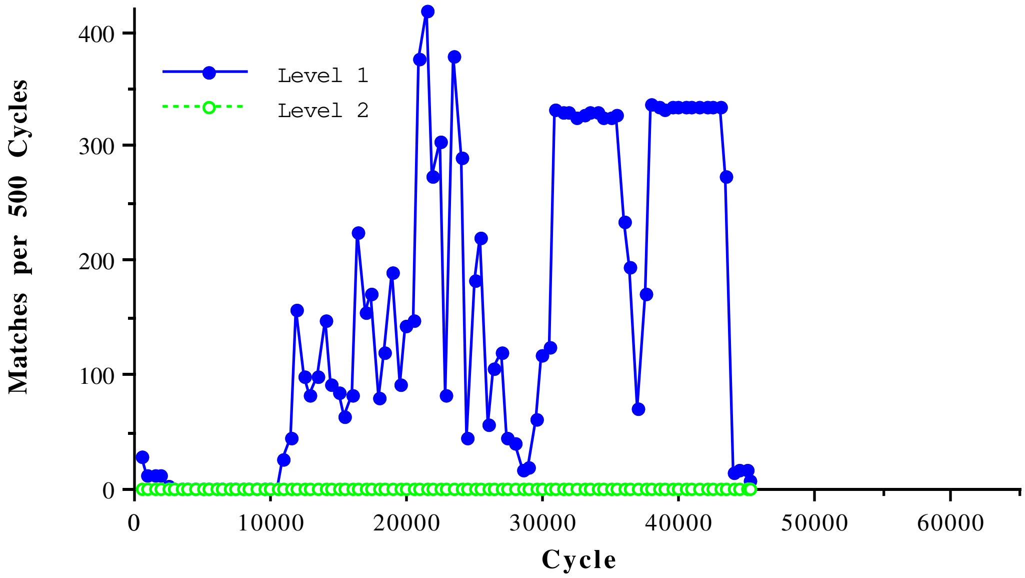


Fig. 1g: Mm & 64K Level 1



Figures 2a through 2f show the number of matches occurring in Level 1 and 2 as a function of execution time for Shallow using the random migration approach.

Fig. 2a: Shallow & No Level 1

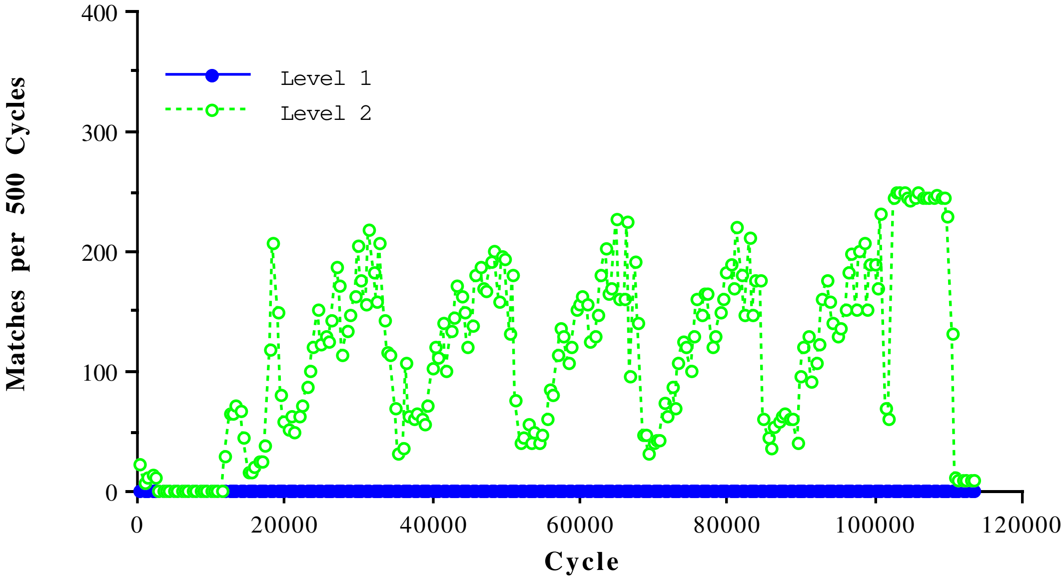


Fig. 2b: Shallow, 4K Level 1, & No Free Slots

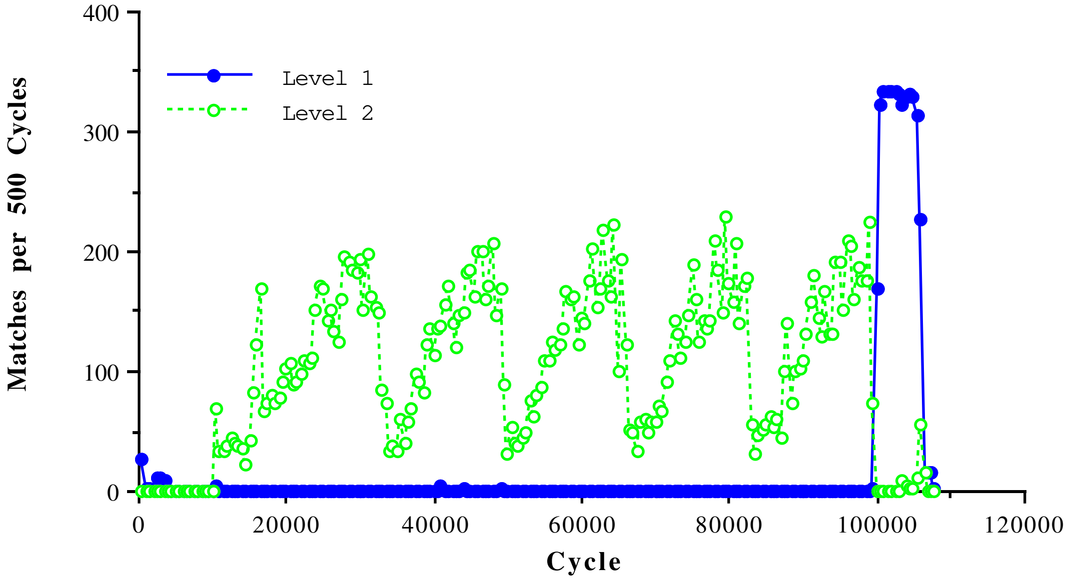


Fig. 2c: Shallow, 4K Level 1, & 2 Free Slots

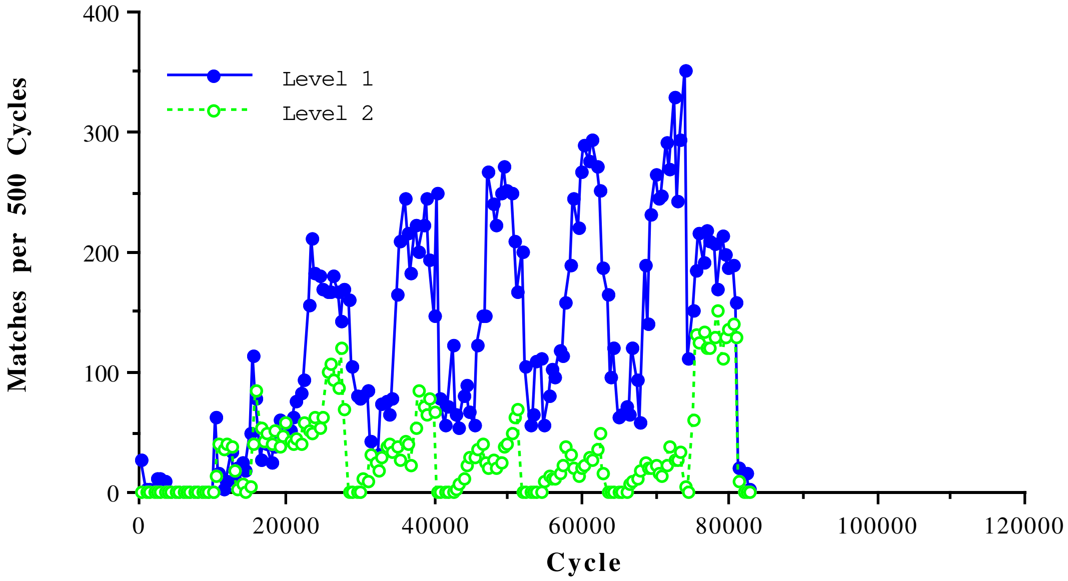


Fig. 2d: Shallow, 4K Level 1, & 4 Free Slots

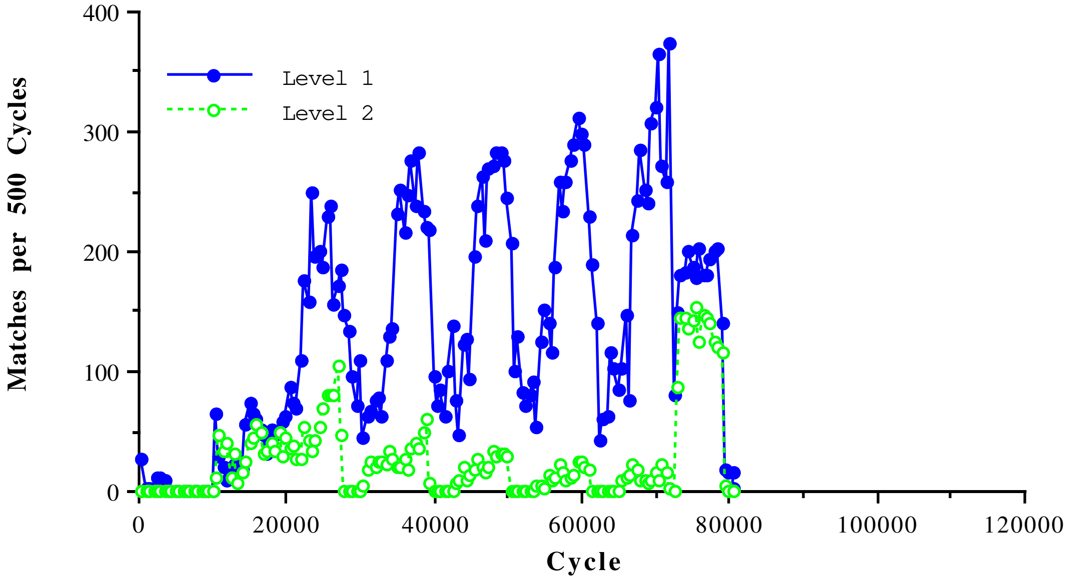


Fig. 2e: Shallow, 4K Level 1, & 6 Free Slots

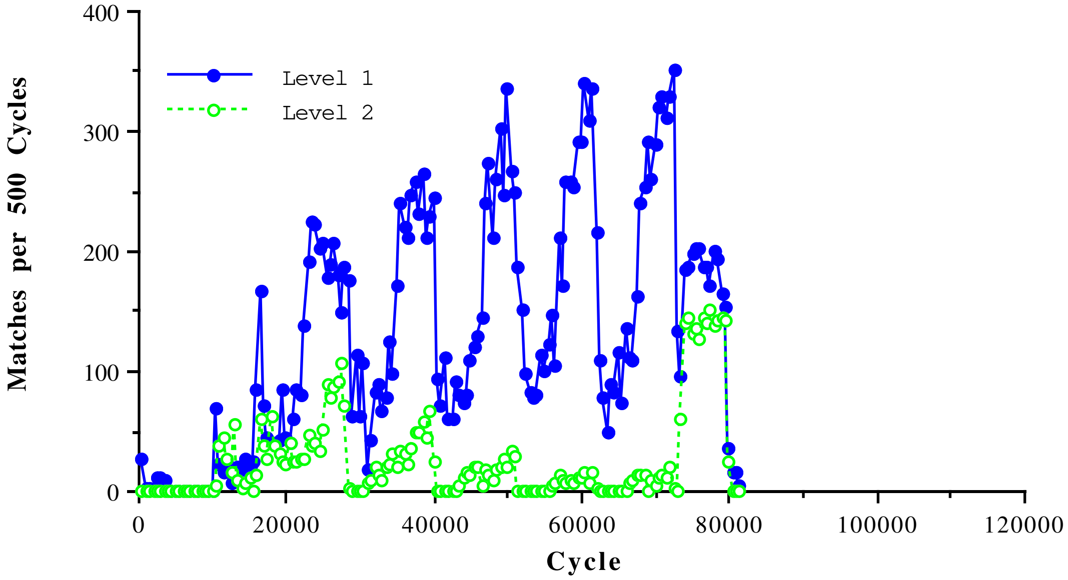


Fig. 2f: Shallow, 4K Level 1, & 8 Free Slots

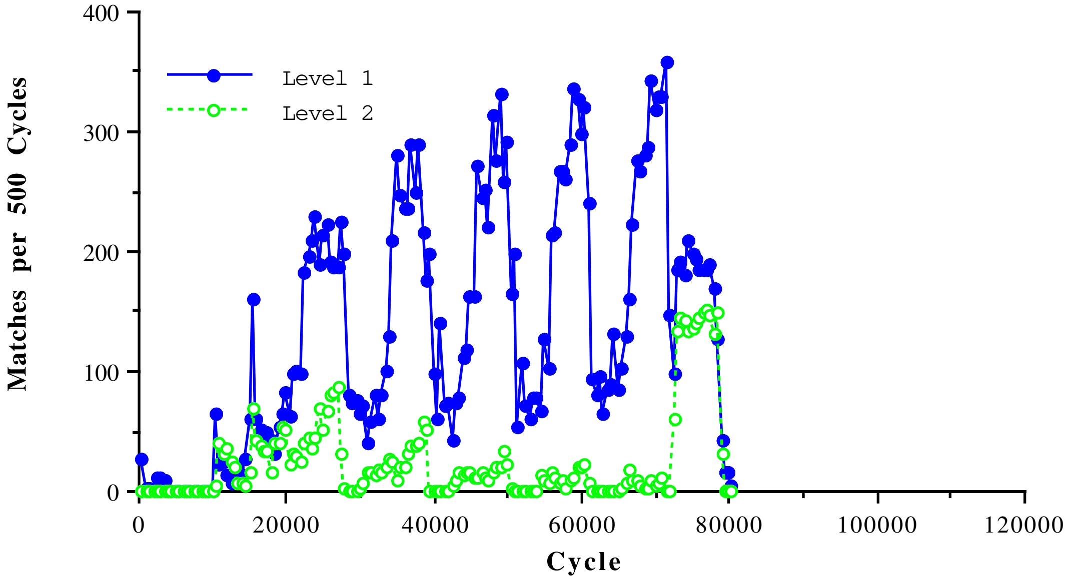


Fig. 2g: Shallow & 64K Level 1

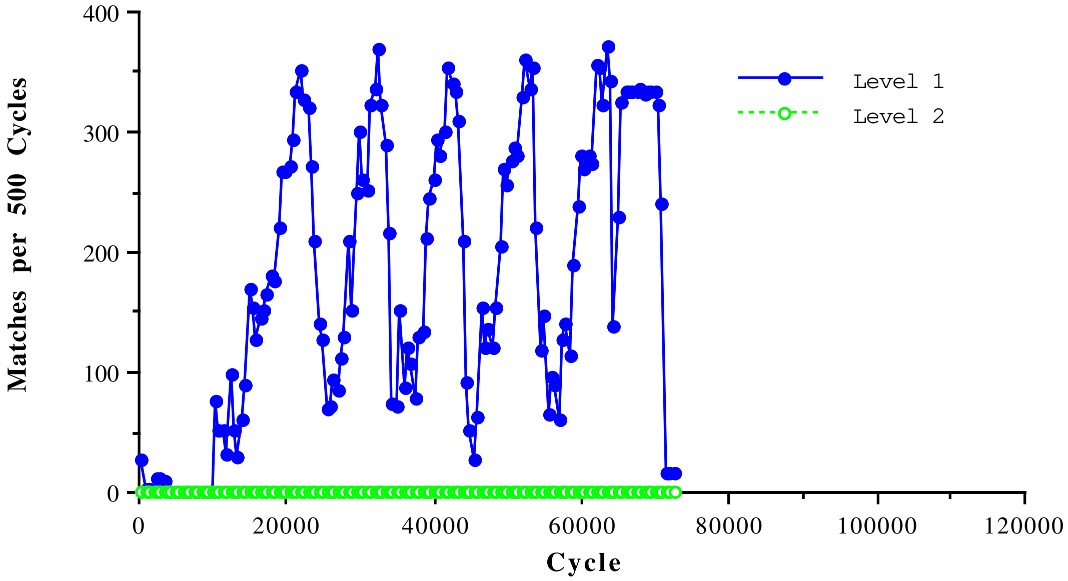


Fig. 3a: Mm, 4K Level 1, 8 Free Slots, & LRU

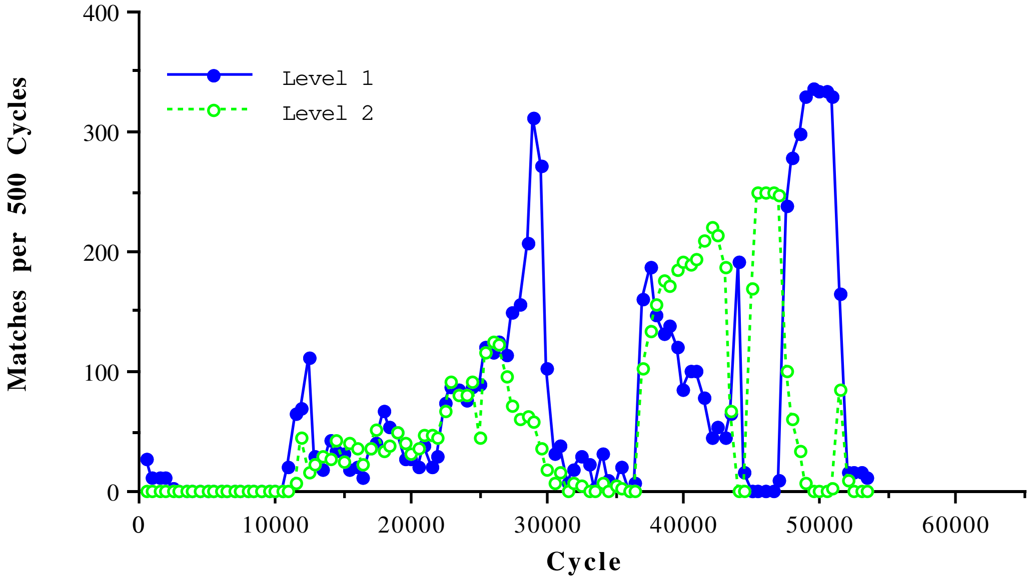
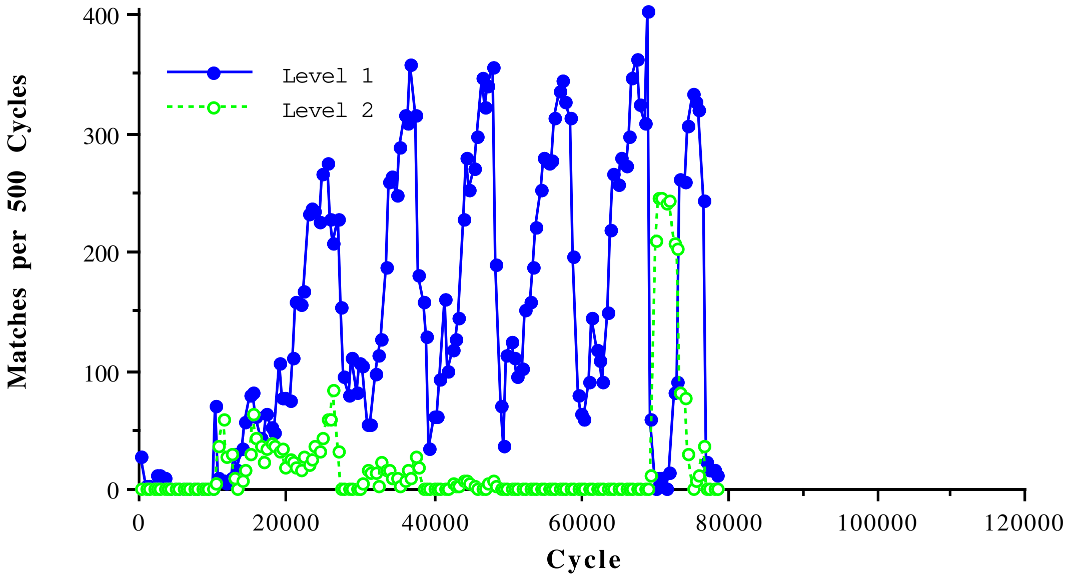


Fig. 3b: Shallow, 4K Level 1, 8 Free Slots, & LRU



Figures 3a and 3b are the same as figures 1f and 2f respectively, except that "least recently used" token migration is used.

S.1.5. Conclusions

The most important aspect of this, and all, caching mechanisms is that, unless applications exploit locality, caching cannot be effective. In this experiment, applications with varying degrees of locality are embedded into a framework which forces the caching mechanism to be used. Although the temporal performance of these applications does always not increase dramatically when the migration mechanism is enabled, the use of cache does. This implies that, for applications with good locality (possibly as a result of effective throttling techniques), the speed of the Level 2 store could be reduced further. In this case, the difference in execution time between the migration enabled and disabled cases would be more substantial.

The number of free slots, maintained by the migration mechanism, has an effect on the ability of the Level 1 cache to deal with sudden, large increases in the number of non-matching tokens. This formulation of matrix multiply begins by eagerly creating many operands required in advance of when they are required. This sudden increase in the number of tokens prevents the migration mechanism from clearing slots in Level 1. As a result, for smaller numbers of free slots, matrix multiply must use Level 2 to store some of these tokens, thus reducing the effectiveness of the caching mechanism. Likewise, the sudden increase in non matching tokens in Shallow results in some Level 2 use, but as the iterations of Shallow proceed, more elements of the original 64x64 matrix are removed and a full "working set" of slots develops in Level 1. This is true even for small numbers of free slots.

It is clear that, although not universal, the "least recently used" strategy has a performance benefit. This is as intuition would suggest. Since this approach is straight forward to implement, a "least recently used" strategy is assumed in the current default implementation of SDFA.

From the tables above, it is clear that, for these programs, four free slots are adequate for exploiting good locality and further increases are unlikely to improve the situation significantly. Therefore, four free slots are assumed in the current default implementation of SDFA.

S.2. Study of a SDFA/9000 System

A scalable version of SDFA can be designed based on the Inmos T9000 processor and communications architecture.

In honour of the external examiner of this thesis, this small experiment explores the scalability of a SDFA system based on the architecture of the T9000. The major differences between the SDFA/9000 and that of the default SDFA system are a reduction in the cycle time, the use of non-pipelined function units, and a relatively low bandwidth, long latency, nearest neighbour, 2D torus communication network. The Architecture Configuration file for the SDFA/9000 is included at the end of this document.

S.2.1. Method

The application programs Shallow and Queens are run on systems ranging in size from a uni-processor to 16 processors. In order to eliminate simulation restrictions from the study, problem sizes and locality and distribution factors are selected which provide nearly uniform distribution on all systems tested. Recall that as problem size increases, the distribution tends toward uniformity, even for problem sizes which do not match the system, see Chapter 7.2. Therefore, there is no loss of generality in this approach. The problem sizes selected are: Queens on a 6x6 board; and Shallow on a 32x32 grid for 2 time steps.

S.2.2. Measurements

Time is measured, but reported as temporal performance (1.0/Time).

S.2.3. Assumptions

Recall that the problem sizes have been selected to "fit" the systems being studied.

S.2.4. Results

Figure 4 shows the temporal performance of Queens and figure 5 shows that of Shallow.

Fig. 4: Queens 6 on SDFA/9000

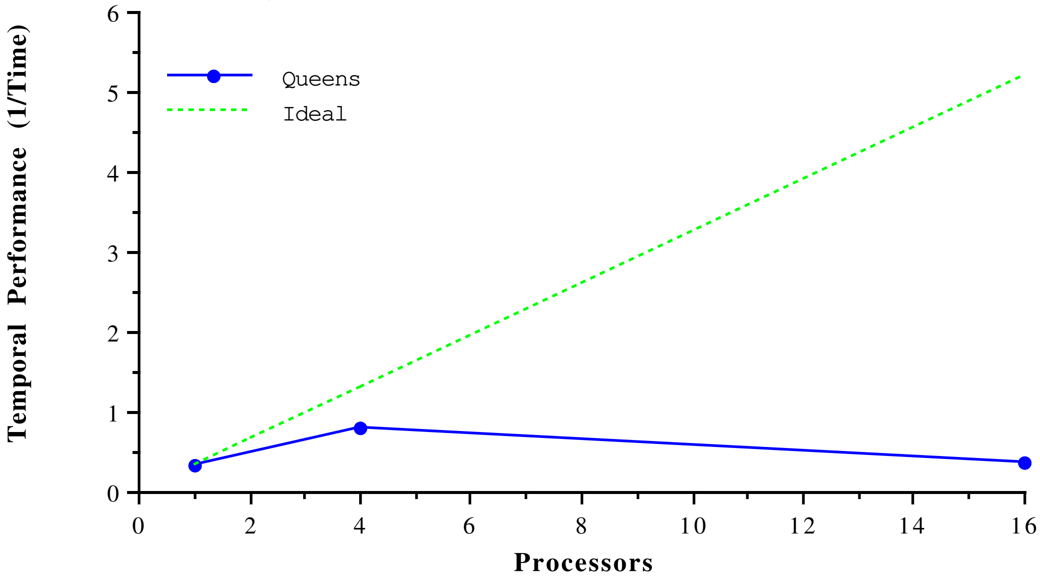


Fig. 5: Shallow 32x32, 2TS on SDFA/9000

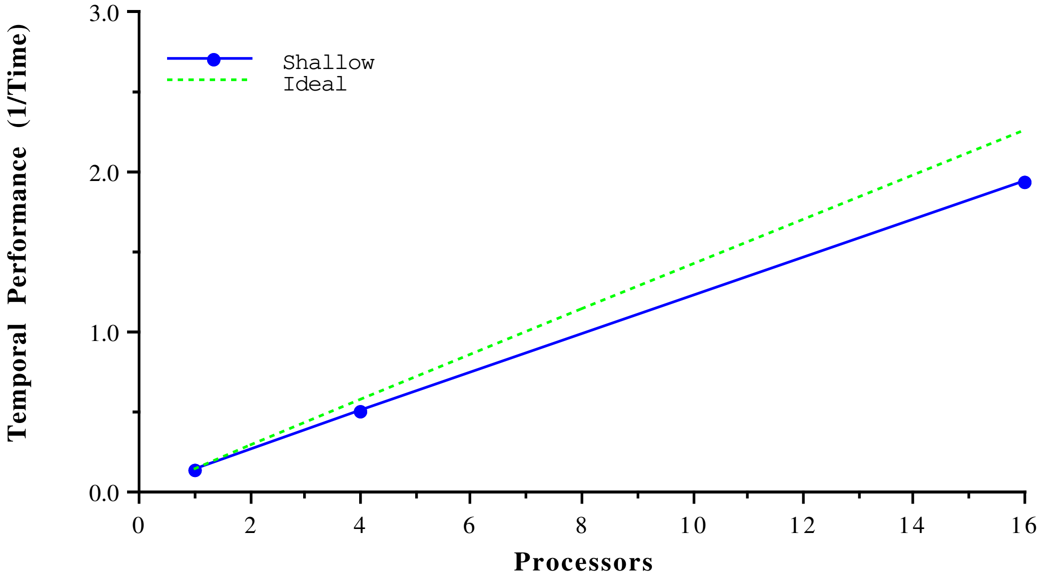
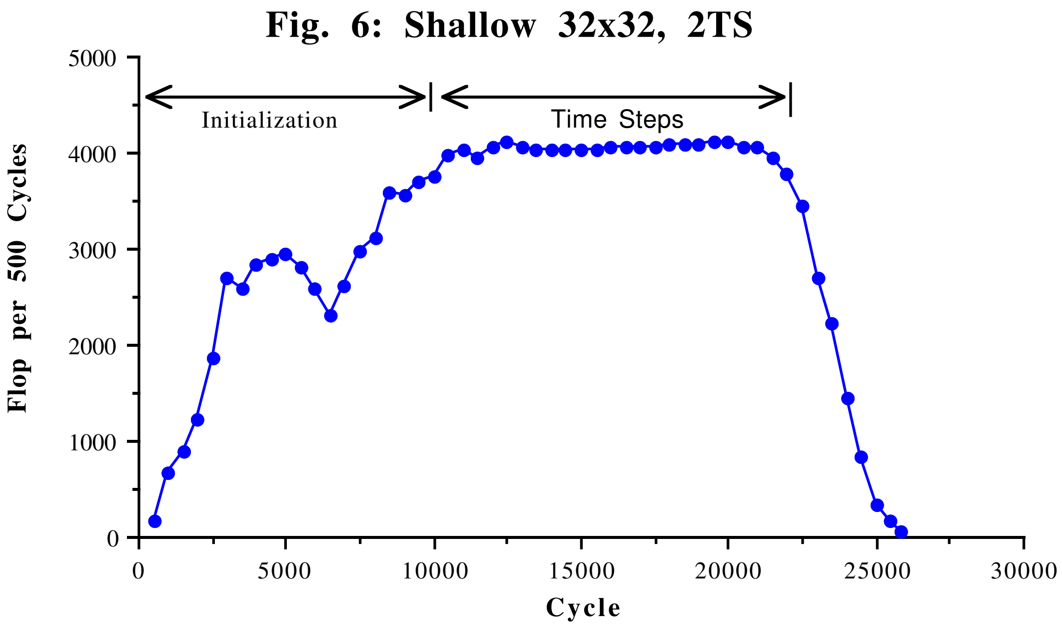


Figure 6 shows the effect that Amdahl's has on Shallow when run for only two time steps.



S.2.5. Conclusions

The absolute performance of Queens is limited by the speed of the divide function unit, which has a long latency (15 cycles) and is not pipelined. In the uni-processor case, the Div and Mod instructions account for 150600 of the 153747 cycles or 98% of the total execution time. Therefore, virtually all other operations are masked entirely by the Div and Mod operations.

The same is true of floating point operations in Shallow. In the uni-processor case, the peak performance of the processor is 28.3 MFlop/s and runs at a sustained 22.2 MFlop/s. This value would increase with more time steps, since the initialization phase requires calls to sin and cos, which are simulated as one Flop requiring 15 cycles (non-pipelined).

Scaling in Queens is not good. This is due to the fine grain nature of the parallelism resulting from the binary calling tree. Although, the data volume is not large, the long latency of the communication causes processors to idle frequently. Alternative algorithms could be devised which have higher degrees of locality.

Shallow, on the other hand, scales well. This is to be expected since the compute to communicate ratio is of O(n). In fact, in the 16 processor system, the average number of tokens transmitted per cycle is 0.54 and the available bandwidth is 3.2 tokens per cycle. Also, Amdahl's law is clearly having an effect on Shallow when running for only 2 time steps.

S.2.5. The SDFA/9000 Architecture Configuration File

```
Cycle time of this system E in nanoseconds               : 20.0

Topology of the network, [Packet, Torus, Hypercube]      : Torus
Dimension of the network (0 for Packet switch)           : 2
Number of processors in the network                      : 16
The sizes of the various dimensions (e.g. 4 4 4)         : 4 4
Length of the inter-processor pipelines                  : 1
Beat of the inter-processor pipelines                    : 40

Length of the processor input queues                     : 4
Width of the processor input queues                      : 1
Length of the processor output queues                    : 32
Width of the processor output queues                     : 1

Type of matching store, [Simple, Forwarding]             :
Forwarding
Number of banks in a matching store                      : 8
Number of matching store input pipelines                 : 2
Length of the matching store input pipelines             : 1
Beat of the matching store input pipelines               : 2
Length of the matching store output queue                : 8
Width of the matching store output queue                 : 1

Maximum size of level-1 store in a matching store bank   : 65535
Migration strategy [Random, LRU]                         :
Random
Target number of free slots to keep in level-1           : 1
Enable level-1 buffer [True, False]                      : False
Length of the level-1 input queue                        : 4
Width of the level-1 input queue                         : 1
Length of the clear queue                                : 20
Width of the clear queue                                 : 1
Length of the level-1 pipeline                           : 1
Beat of the level-1 pipeline                             : 2
Length of the forwarding queue                           : 8
Width of the forwarding queue                            : 1
Length of the return queue                               : 16
Width of the return queue                                : 1
Length of the level-1 output queue                       : 16
Width of the level-1 output queue                        : 1
Length of the output pipeline                            : 1
Beat of the output pipeline                              : 2

Length of the multiplexed level-1/2 transfer pipeline    : 1
Beat of the multiplexed level-1/2 transfer pipeline      : 2

Maximum size of level-2 store in a matching store bank   : 65535
Length of the level-2 input queue                        : 4
Width of the level-2 input queue                         : 1
Length of the level-2 pipeline                           : 2
Beat of the level-2 pipeline                             : 5
Length of the level-2 output queue                       : 2
Width of the level-2 output queue                        : 1
```

```
Type of token queue, [Queueing, Stacking, Sorting]       :
Queueing
Maximum length of Token Queue                            : 65535
Length of the token queue input pipelines                : 1
Beat of the token queue input pipelines                  : 2
Length of the token queue output queue                   : 4
Width of the token queue output queue                    : 2

Length of the processing element input pipeline          : 1
Beat of the processing element input pipeline            : 2
Number of processing element output pipelines            : 2
Length of the processing element output pipelines        : 1
Beat of the processing element output pipelines          : 2
Length of the processing element input queue             : 32
Width of the processing element input queue              : 2
Length of the processing element output queue            : 4
Width of the processing element output queue             : 2

Instruction fetch arbitration [Fair, Local, Match]       : Local
Complex Instruction-set facility enabled                 : False
Number of active instruction fetch pipelines             : 2
Length of the instruction fetch pipeline(s)              : 1
Beat of the instruction fetch pipeline(s)                : 1

Length of the Call-Return input queue                    : 8
Width of the Call-Return input queue                     : 2
Length of the Call-Return pipeline                       : 1
Beat of the Call-Return pipeline                         : 1
Length of the Call-Return output queue                   : 4
Width of the Call-Return output queue                    : 2

Vectorization enabled, [True, False]                     : False
Length of the Vector Unit input queue                    : 16
Width of the Vector Unit input queue                     : 2
Length of the Vector Unit pipeline                       : 1
Beat of the Vector Unit pipeline                         : 1
Length of the Vector Unit output queue                   : 16
Width of the Vector Unit output queue                    : 2

Length of the Transcendental input queue                 : 16
Width of the Transcendental input queue                  : 2
Length of the Transcendental pipeline                    : 1
Beat of the Transcendental pipeline                      : 15
Length of the Transcendental output queue                : 4
Width of the Transcendental output queue                 : 2

Length of the Div-Sqrt input queue                       : 16
Width of the Div-Sqrt input queue                        : 2
Length of the Div-Sqrt pipeline                          : 1
Beat of the Div-Sqrt pipeline                            : 15
Length of the Div-Sqrt output queue                      : 4
Width of the Div-Sqrt output queue                       : 2

Length of the Floating Point input queue                 : 16
Width of the Floating Point input queue                  : 2
Length of the Floating Point pipeline                    : 1
Beat of the Floating Point pipeline                      : 2
Length of the Floating Point output queue                : 4
Width of the Floating Point output queue                 : 2

Number of Integer function units, [1,2]                  : 2
Length of the Integer input queue(s)                     : 16
Width of the Integer input queue(s)                      : 1
```

```
Length of the Integer pipeline(s)                           : 1
Beat of the Integer pipeline(s)                             : 1
Length of the Integer output queue(s)                       : 4
Width of the Integer output queue(s)                        : 2

Local by-pass enabled, [True, False]                        : True
Length of by-pass queue                                     : 32
Width of by-pass queue                                      : 2

Short stay matching store, [None, Simple, Rectifying]       :
Rectifying
Number of cycles to wait in the short stay matching store  : 64
Size of the short stay matching store                       : 512
Length of short stay matching store input queues            : 4
Width of short stay matching store input queues             : 2
Length of the short stay matching store pipelines           : 1
Beat of the short stay matching store pipelines             : 1
Length of the short stay matching store success queue       : 4
Width of the short stay matching store success queue        : 2
Length of the short stay matching store fail queue          : 4
Width of the short stay matching store fail queue           : 2
```

Bibliography

Abra88a. D. Abramson and G. Egan. An Overview of the RMIT/SSIRO Parallel Systems Architecture Project. *Australian Computer Journal 20*, 3 (August 1988).

Abra90a. D. Abramson and G. Egan. The RMIT Data-Flow Computer: A Hybird Architecture. *The Computer Journal* (June 1990).

Abra91a. D. Abramson and G. Egan. Design of a High-Performance Data-Flow Multiprocessor. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 1, Englewood Cliffs, N.J. (1991) pp. 121-142.

ACRI92a. A. Rawsthorne, P. Bird, and C. Bertin, *User-Visible Instruction Set (Book 1)*, Advanced Computer Research Institute, ACRI1-AR-SPEC-0001 (June 1992).

Agar90a. A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz. A Processsor Architecture for Multiprocessing. In *Proceedings of ISCA-17* (May 1990) pp. 104-114.

Alve92a. G. Alverson, R. Alverson, and D. Callahan. Exploiting Heterogenous Parallelism on a Mul-ithreaded Multiprocessor. In *Proceedings of the ICS-92,* ACM, ACM Press (July 1992) pp. 188-197.

Amam91a. M. Amamiya. An Ultra-Multiprocessing Architecture for Functional Languages. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 1, Englewood Cliffs, N.J. (1991) pp. 95-120.

Arvi80a. Arvind and R.E. Thomas. I-structures: An Efficient Data Type for Functional Languages. Tech. Rept. LCS/TM-178 Massachusetts Institute of Technology, Cambridge, Mass. (1980).

Arvi86a. Arvind and D.E. Culler. Managing Resources in a Parallel Machine. In *Fifth Generation Computer Architectures*. Elsevier Science Publishers (1986) pp. 103-121.

Arvi87a. Arvind and R.A. Inaucci. Two Fundamental Issues in Multiprocessing. In *DFVLR Conference on Parallel Processing in Science and Engineering,* DFVLR, Bonn-Bad Godesberg, Germany (1987) pp. 61-88.

Arvi88a. Arvind and D.E. Culler. The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures. In *CONPAR 88,* C.R. Jesshope and K.D. Reinartz, British Computer Society, Cambridge University Press (1988) pp. 541-555.

Arvi88b. Arvind, D.E. Culler, and G.K. Maa. Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs. Tech. Rept. 279, Computational Structures Group Memo, Massachusetts Institute of Technology, Cambridge, Mass. (March 1988).

Arvi89a. Arvind and R.S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers* (1989), pp. 1-29.

Arvi91a. Arvind, L. Bic, and T. Ungerer. Evolution of Data-Flow Computers. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 1, Englewood Cliffs, N.J. (1991) pp. 3-34.

Babb84a. R.G.I. Babb. Parallel Processing with Large Grain Data Flow Techniques. *IEEE Computer* (July 1984), pp. 55-61.

Bako90a. H.B. Bakoglu, G.F. Grohoski, and R.K. Montoye. The IBM RISC System 6000 Processor: Hardware Overview. *IBM Journal of Research and Development 34*, 1 (January 1990) pp. 12-23.

Bart91a. P.S. Barth, R. Hikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. Tech. Rept. 327, Computational Structures Group Memo, Massachusetts Institute of Technology (March 1991).

Böhm89a. A.P.W. Böhm and Y.M. Teo. The Heterogeneous Multi-Ring Dataflow Simulator. Tech. Rept. UMCS089-3-1 University of Manchester, Dept. of Computer Science (June 1989).

Böhm90a. A.P.W. Böhm and J.G. Gurd. Iterative Instructions in the Manchester Dataflow Computer. *IEEE Transactiona on Parallel and Distributed Systems 1*, 2 (April 1990) pp. 129-139.

Böhm92a. A.P.W. Böhm and R.E. Hiromoto. *The Data-Flow Time Complexity of FFT*, To appear in the Proceedings of the Hamilton Island Workshop on Data-Flow Computing (May 1992).

Brob86a. S. Brobst. Instruction Scheduling and Token Storage Requirements in a Dataflow Supercomputer. Tech. Rept. CSG-Memo-264 Massachusetts Institute of Technology (June 1986).

Cann88a. D.C. Cann and R.R. Oldehoeft. Reference Count and Copy Elimination for Parallel Applicative Computing. Tech. Rept. CS-88-129 Colorado State University, Fort Collins Colorado (November 1988).

Cann89a. D.C. Cann and R.R. Oldehoeft. High Performance Parallel Applicative Computing. Tech. Rept. CS-89-104 Colorado State University, Fort Collins Colorado (Febuary 1989).

Cull87a. D.E. Culler and Arvind. Resource Requirements of Dataflow Programs. Tech. Rept. 280, Computation Structures Group Memo, Massachusetts Institute of Technology (December 1987).

Cull89a. D.E. Culler. *Managing Parallelism and Resources in Scientific Dataflow Programs*, Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge Mass. (June 1989).

Cull91a. D.E. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Finegrain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Procceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991).

Cull92a. D.E. Culler, K.E. Schauser, and T. von Eicken. Two Fundamental Limits in Data-Flow Multiprocessing. Tech. Rept. UBC/CSD 92/716 Computer Science Division, University of California, Berkeley, Elsevier (1992).

Cull93a. D.E. Culler, K.E. Schauser, and T. von Eicken. Two Fundamental Limits in Data-Flow Computing. In *IFIP WG 10.3, Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism,* IFIP, Elsevier (January 1993).

Dall87a. W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers C-36*, 5 (May 1987), pp. 547-553.

daSi83a. J.G.D. da Silva and I. Watson. Pseudo-associative Store with Hardware Hashing. *IEE Proceedings 130*, 1 (January 1983) pp. 19-24.

Davi78a. A.L. Davis. The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine. In *Proceedings of the 5th Annual Symposium on Computer Architecture,* ACM, ACM Press (April 1978) pp. 210-215.

DEC92a. *DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet*, Digital Equipment Corporation, Maynard, Mass. (April 1992).

Dene81a. *Heterogeneous Elemnet Processor Principles of Operation*, Denelcor Inc., Denver, Co. (April 1981).

Denn73a. J.B. Dennis. First Version of a Dataflow Procedure Language. Tech. Rept. MAC 61 Massachusetts Institute of Technology, Cambridge Mass. (1973).

Denn80a. J.B. Dennis. Dataflow Supercomputers. *IEEE Computer 13*, 11 (November 1980) pp. 48-56.

Denn91a. J. Dennis. The Evolution of a Static Data-Flow Architecture. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 2, Englewood Cliffs, N.J. (1991) pp. 35-94.

Dong88a. J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Tech. Rept. ANL-MCS-P1-0888 Argonne National Laboratory (August 1988).

Egan90a. G.K. Egan. Some Shallow Experiences. Tech. Rept. 31-004 School of EE, Swinborne Institute of Techniology (August 1990).

Egan91a. G. Egan, N. Webb, and W. Böhm. Some Architectural Features of the CISRAC II Data-Flow Computer. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 5, Englewood Cliffs, N.J. (1991) pp. 143-173.

Fish79a. J.A. Fisher. The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources. Tech. Rept. COO-3077-161, U. S. Department of Energy, Courant Laboratory, N. Y. Univ. (October 1979).

Fish88a. J.A. Fisher. Replacing Hardware that Thinks (Especially about Parallelism) with a Very Smart Compiler. In *International Specialist Seminar: The Design and Application of Parallel Digital Processors,* J.B.G. Roberts, IEE, Short Run Press Ltd. Exeter, London (1988) pp. 153-159.

Fole89a. J. Foley. Manchester Dataflow Machine: Benchmark Test Evaluation Report. Tech. Rept. UMCS-89-11-1 University of Manchester, Dept. of Computer Science (January 1989).

Gott91a. A. Gottlieb. An Outsider's View of Data-Flow. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 22, Englewood Cliffs, N.J. (1991) pp. 537-582.

Gott92a. I. Gottieb and L. Biran. *Locality in the Data-Flow Paradigm*. To appear in the proceedings of the Hamalton Island Workshop in Data-Flow Computing (May 1992).

Graf91a. V.G. Grafe, J.E. Hoch, G.S. Davidson, V.P. Holmes, D.M. Davenport, and K.M. Steel. The Epislon Project. In *Advanced Topics in Data-Flow Computing*. Perntice Hall, J.L. Gaudoit and L. Bic eds., Ch. 6, Englewood Cliffs, N.J. (1991) pp. 175-205.

Gurd77a. J.R. Gurd. *Iteration and Recursion, Notes.*. John Gurd's private notes which formed the basis of function colouring in the Manchester Data-Flow Machine (1977, Approximately).

Gurd85a. J.R. Gurd, C.C. Kirkham, and I. Watson The Manchester Prototype Dataflow Computer. *Communications of the ACM 28*, 1 (January 1985) pp. 34-52.

Gurd87a. J.R. Gurd, W. Böhm, and Y.M. Teo. Performance Issues in Dataflow Machines. In *Future Generations Computer Systems*. Elsevier Scientific (1987) pp. 285-297.

Gurd87b. J.R. Gurd, C. Kirkham, and W. Bohm. The Manchester Dataflow Computing System. In *Experimental Parallel Computing Architectures*. North-Holland, J.J. Dongarra, Ch. 6, Amsterdahm (1987) pp. 177-220.

Gurd92a. J.R. Gurd and D.F. Snelling. Manchester Data-Flow: A progress Report. In *1992 International Conference on Supercomputing,* ACM, ACM Press (1992) pp. 216-225.

Hama90a. V.C. Hamacher, Z.G. Vranesic, and S.G. Zaky. *Computer Organization,* McGraw-Hill , 3rd (1990).

Henn92a. J. Hennessey, A. Gupta, and K. Gharachorloo. Hiding Memory Latency using Dynamic Scheduling in Shared Memory Multiprocessors. In *Proceedings of ISCA-19 May 1992,* ACM and SIGARCH (May 1992) pp. 22-33.

Hira91a. K. Hiraki, S. Sekiguchi, and T. Shimada. Status Report on SIGMA-1: A Data-Flow Supercomputer. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 7, Englewood Cliffs, N.J. (1991) pp. 207-224.

Hock88a. R.W. Hockney and C.R. Jesshope. *Parallel Computers 2,* Adam Hilger, Bristol (1988).

Hock92a. R.W. Hockney. A Framework for Benchmark Performance Analysis. *Supercomputing 48* (March 1992), pp. 9-22.

Hoff88a. G.R. Hoffmann, P.N. Swarztrauber, and R.A. Sweet. Aspects of Using Multiprocessors for Meteorological Modelling. In *Multiprocessing in Meteorological Models,* G.R. Hoffmann and D.F. Snelling eds., ECMWF, Springer-Verlag, Berlin (1988) pp.125-196.

Hwan89a. K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing,* McGraw-Hill , Computer Science Series (1989).

Inmo89a. *Transputer Technical Notes*, Inmos Limited, (1989).

Inmo91a. *The T9000 Transputer Products Overview Manual*, Inmos Ltd. / SGS-Thomson, Marlow (1991).

Karp66a. R.M. Karp and R.E. Miller. Properties of a Model for Parallel Computatins: Determinacy, Termination and Queueing. *SIAM Journal of Applied Mathematics 11*, 6 (November 1966), pp. 1390-1411.

Kirk82a. C.C. Kirkham. *The Manchester Prototype Dataflow System Basic Programming Manual*, University of Manchester, Manchester, 3rd, (April 1982).

Kirk87a. C.C. Kirkham *The Manchester Prototype Dataflow System Basic Programming Manual*, University of Manchester, Manchester, 6th, (April 1987).

Kowa85a. *Parallel MIMD Computation: HEP Supercomputer and its Applications,* MIT Press, Cambridge, Mass. Kowalik, Janusz S. ed. (1985).

KSR91a. *KSR1 Principles of Operation*, Kendall Square Research, 170 Tracer Lane, Waltham, Mass. (October 1991).

KSR91b. *KSR Fortran Programming*, Kendall Square Research, 170 Tracer Lane, Waltham, Mass. (October 1991).

Laga84a. J.C. Lagarias. The 3x+1 Problem and its Generalizations. *American Mathematical Monthly* (June 1984).

Laws79a. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software 5*, 3 (September 1979) pp. 308-371.

Leis85a. C.E. Leiserson. Fat Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers 34*, 10 (October 1985) pp. 892-901.

Leno90a. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of ISCA-17* (May 1990) pp. 148-159.

Love93a. D. Loveman. Draft High Performance Fortran Language Specification, Version 1.0 Draft. Tech. Rept. 92225 Rice University, Houston, Texas (January 1993).

Love93b. D. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology 1*, 1 (Febuary 1993), pp. 25-42.

McMa88a. F.H. McMahon. *The Livermore Fortran Kernels Test of the Numerical Performance Range in Performance Evaluation of Supercomputers,* North Holland: Amsterdahm (1988) pp. 143-186.

Ni93a. L.M. Ni and P.K. Mckinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer 26*, 2 (Febuary 1993) 62-76.

Nikh87a. R.S. Nikhil. *Id Nouveau: Reference Manual Parts I and II*, MIT Laboratory for Computer Science, Cambridge, Mass. (April 1987).

Nikh88a. R.S. Nikhil. ID Reference Manual V. 88.0. Tech. Rept. 284, Computational Structures Group Memo, Massachusetts Institute of Technology (March 1988).

Nikh89a. R.S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing?. In *16th International Symposium on Computer Architecture*, ACM Press (June 1989).

Nikh92a. R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. Tech. Rept. 325-2, Computional Structures Group Memo, Massachusetts Institute of Technology (March 1992).

O'Boy92a. M. O'Boyle and G.A. Hedayat. A Transformational Approach to Compiling SISAL for Distributed Memory Architectures. In *Proceedings of ICS-92*, ACM Press (July 1992) pp. 335-346.

Papa88a. G.M. Papadopoulos. Implementation of a General-Purpose Dataflow Multiprocessor. Tech. Rept. TR-432 Massachusetts Institute of Technology, Cambridge Mass. (August 1988).

Pitt88a. F.M. Pittelli and D.L. Smitley. Analysis of a 3D Toridal Network for a Shared Memory Architecture. Tech. Rept. SRC-TR-88-011 Supercomputing Research Center - IDA, Lanham, MD (July 1988).

Poly87a. C. Ploychronopolous and D. Kuck. Guided Self-Scheduling, A Practical Scheduling Scheme for Parallel Computers. *IEEE Transactions on Computers 36* (December 1987).

Poly90a. C. Polychronopoulos, M. Gitkar, M. Haghighat, B. Lee, and D. Schouten. The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran. In *Languages and Compilers for Parallel Computers*. North-Holland, D. Gelernter, A. Nicolau, and D. Paduda (Amsterdahm 1990).

Ritc88a. H. Ritchie. Application of the Semi-Lagrangian Method to a Spectral Model of the Shallow Water Equations. *Monthly Weather Review 116*, 8 (August 1988).

Rugg87a. C.A. Ruggiero. Throttle Mechanisms for the Manchester Dataflow Machine. Tech. Rept. UMCS-87-8-1 University of Manchester, Dept. of Computer Science (July 1987).

Saka91a. S. Saka, K. Hiraki, Y. Yamaguchi, Y. Kodama, and T. Yuba. Pipeline Optimization of a Data-Flow Machine. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 8, Englewood Cliffs, N.J. (1991) pp. 225-246.

Sarg85a. J. Sargent and W. Böhm. *IF1 Compilation System User's Guide*, Department of Computer Science, University of Manchester, 2nd (October 1985).

Sarg86a. J. Sargent and C.C. Kirkham. Stored Data Structures in the Manchester Dataflow Machine. Tech. Rept. UMCS-86-4-3 University of Manchester, Dept. of Computer Science (1986).

Sarg86b. J. Sargent. Load Balancing, Locality and Parallelism in Fine-Grain Parallel Machines. Tech. Rept. UMCS-85-11-5 Universtiy of Manchester, Dept. of Computer Science (January 1986).

Seit85a. C.L. Seitz. The Cosmic Cube. *Communications of the ACM 28* (January 1985) pp. 22-23.

SISAL85. *SISAL: Streams and Iteration in a Single Assignment Language*, University of Manchester, LLNL, DEC, and Colorado State University, Manchester, Version 1.2 (January 1985).

Sked85a. S. Skedzielewshi and J. Glauert. *IF1*, LLNL and the University of Manchester (July 1985).

Smit78a. B.J. Smith. A Pipelined, Shared Resource MIMD Computer. *Proceedings of the International Conference on Parallel Processing* (1978) pp. 6-8.

Smit88a. B. Smith and M.R. Thistle. A Processor Architecture for Horizon. In *Proceedings of Supercomputing '88,* Orlando, Florida, (November 1988) pp. 35-41.

Smit90a. B. Smith, R. Velverson, and D.O. Callahan. The Tear Computer System. In *1990 International Conference on Supercomputing,* A. Sameh, ACM Press, Baltimore, MD (June 1990) pp. 1-6.

Snel85a. D.F. Snelling and T.C. Chambers. NUMELP: A Parallel Implementation of a Nuclear Power Plant Boiler Model. *Parallel Computing'85* (1985).

Snel89a. D.F. Snelling. Experimental Guide-lines for a Shallow Mapping Study. Tech. Rept. 21 Department of Computing Studies, University of Leicester, Leicester (May 1989).

Snel91a. D.F. Snelling. How Long is a MegaFlop?. *The Design and Application of Parallel Digital Processors II* (1991) pp. 145-149.

Snel92a. D. Snelling and J. Gurd. *Regulating Workload in Massively Parallel Computer Systems*. To appear in proceedings of the ISCA Workshop on Data-Flow computing, Hamilton Island (May 1992).

Srin86a. V.P. Srini. An Architectural Comparison of Dataflow Systems. *IEEE Computer 19*, 3 (March 1986) pp. 68-87.

Teo89a. Y.M. Teo. Concurrency Control in the Multi-Ring Manchester Dataflow Machine. Tech. Rept. UMCS-89-11-2 University of Manchester, Dept. of Computer Scinence, Manchester (September 1989).

Teo91a. Y.M. Teo and W. Böhm. Resource Management and Iterative Instructions. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 18, Englewood Cliffs, N.J. (1991) pp. 481-499.

Veen86a. A.H. Veen. Dataflow Machine Architectures. *ACM Computing Surveys 18*, 4 (December 1986) pp. 365-396.

Wats82a. I. Watson and J. Gurd. A Practical Data Flow Computer. *IEEE Computer 15*, 2 (Febuary 1982) pp. 51-57.

Wdal88a. P. Wadler Strictness Analysis Aids Time Analysis. *ACM Symposium on Principles of Programming Languages* (January 1988).

Yang92a. T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors. In *Proceedings of ISC-92,* ACM, ACM Press (July 1992) pp. 429-437.

Yuba84a. T. Yuba, T. Shimada, K. Hiraki, and H. Kashiwagi. Sigma-1: A Dataflow Computer for Scientific Computations. In *Proceedings of the 2nd International Conference on Vector and Parallel Processors in Computational Science,* I.S. Duff and J.K. Reid eds., North Holland, Oxford (August 1984) pp. 141-148.

Yuba90a. T. Yuba, T. Shimada, Y. Yamaguchi, K. Hiraki, and S. Sakai. Dataflow Computer Developments in Japan. In *Proceedings of ISC-90*, ACM Press (June 1990) pp. 140-147.