



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

The Super-Actor Machine: A Hybrid Dataflow/von Neumann Architecture

by
Herbert Hing-Jing Hum

School of Computer Science
McGill University, Montréal
Québec, Canada

May 1992

a thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Copyright © 1992 by Herbert Hing-Jing Hum



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-74897-4

Canada

Abstract

Emerging VLSI/ULSI technologies have created new opportunities in designing computer architectures capable of hiding the latencies and synchronization overheads associated with von Neumann-style multiprocessing. Pure Dataflow architectures have been suggested as solutions, but they do not adequately address the issues of local memory latencies and fine-grain synchronization costs. In this thesis, we propose a novel hybrid dataflow/von Neumann architecture, called the *Super-Actor Machine*, to address the problems facing von Neumann and pure dataflow machines. This architecture uses a novel high-speed memory organization known as a *register-cache* to tolerate local memory latencies and decrease local memory bandwidth requirements. The register-cache is unique in that it appears as a register file to the execution unit, while from the perspective of main memory, its contents are tagged as in conventional caches. Fine-grain synchronization costs are alleviated by the hybrid execution model and a loosely-coupled scheduling mechanism.

A major goal of this dissertation is to characterize the performance of the Super-Actor Machine and compare it with other architectures for a class of programs typical of scientific computations. The thesis includes a review on the precursor called the *McGill Dataflow Architecture*, description of a *Super-Actor Execution Model*, a design for a Super-Actor Machine, description of the register-cache mechanism, compilation techniques for the Super-Actor Machine and results from a detailed simulator. Results show that the Super-Actor Machine can tolerate local memory latencies and fine-grain synchronization overheads—the execution unit can sustain 99% throughput—if a program has adequate exposed parallelism.

Résumé

L'émergence de nouvelles technologies de VLSI/ULSI a créé de nouvelles possibilités pour la conception d'architectures d'ordinateurs capables de cacher les temps d'attente et de synchronisation associés au multitraitement de style von Neumann. Des architectures de flux de données pur sont suggérées, mais elles ne résolvent pas les problèmes de temps d'attente de la mémoire locale et des coûts de synchronisation à grains fins. Dans cette thèse, nous proposons une nouvelle architecture hybride de flux de données/von Neumann, appelée la "Super-Actor Machine", pour résoudre les problèmes auxquels sont confrontées les machines de von Neumann et de flux de données pur. Cette architecture utilise une nouvelle organisation de la mémoire rapide appelée "register-cache" pour tolérer les temps d'attente de la mémoire locale et diminuer les exigences de largeur de bande de la mémoire locale. Le "register-cache" est unique car il apparaît comme un bloc de registres à l'unité de traitement tandis que pour la mémoire locale, leurs contenus sont référencés comme pour une antémémoire conventionnel. Les coûts de synchronisation à grains fins sont allégés par le modèle de traitement hybride et un mécanisme d'ordonnancement couplé librement.

Un but majeur de cette thèse est de caractériser la performance de la "Super-Actor Machine" et de la comparer avec d'autres architectures pour une classe des programmes typique des applications scientifique. Cette thèse comprend une présentation de l'architecture de base, appelée "McGill Dataflow Architecture", un modèle de traitement "super-actor", la conception de la "Super-Actor Machine", une description du "register-cache", les techniques

de compilation pour la “Super-Actor Machine”, ainsi que des résultats d’un simulateur détaillé. Les résultats montrent que la “Super-Actor Machine” peut tolérer les temps d’attente de la mémoire locale et les coûts de synchronisation à grains fins (l’unité de traitement peut soutenir un débit de 99%) si le programme a un niveau de parallélisme apparent suffisant.

Acknowledgments

I am eternally grateful to my supervisor, Guang Gao, for giving me this opportunity to pursue my dreams of designing the next generation of computers. Without his infinite enthusiasm, his broad and insightful knowledge of computer systems, and his never-ending pursuit for excellence, this thesis would not have been possible. I am also grateful for the guidance from Jack Dennis who has breathed excitement into dataflow computing and illuminated the grand possibilities of architectures based on those ideas. I would also like to thank the folks in the Advanced Computer Architecture and Program Structures Group (ACAPS) who have influenced my work and provided many valuable comments in my endeavours: Philip Wong, for helping us substantiate our ideas; Rene Tio, for his insightful comments on the McGill Dynamic Dataflow Architecture; and Robert Yates, Guy Tremblay, Russ Olsen, Erik Altman, Kevin Theobald, Laurie Hendren, Jean-Marc Monti and other members of ACAPS for their interest and comments on the Super-Actor Machine. Indeed, the *corp d'esprit* that permeates through this group is very conducive to productive research par excellence.

I would like to thank the Natural Sciences and Engineering Research Council for supporting my studies. I would also like to thank the people at Centre de recherche informatique de Montréal (CRIM) for sponsoring and giving me this chance to pursue my doctoral degree. With their generosity, I was able to conduct my research with state-of-the-art equipment and in relative tranquility for those moments when I needed it most. Notably, I would like to thank Renato de Mori for offering me a research position at CRIM and for introducing me to Gao; Frances de Verteuil and Bernard Turcotte for always believing in and supporting me; Jacqueline Bourdeau for her valuable French lessons; Thina Nguyen

for helping me with the diagrams, Justin Bur for his help with L^AT_EX and other system stuff, Darren Kinley and Ron Hall for more system stuff, and Claude Achcar, Anne Gisiger, Daniel Gorham, Diane Goupil, Darren Kinley, Asnat Macoosh, Jennifer Muise and Charles Snow for their friendship and long talks on politics. Je voudrais remercier tous mes collègues au CRIM.

Lastly, I would like to express my deepest appreciation to my wife, Jeanne, my two daughters, Justine and Lauren, and my son, Emerson for their love and understanding while I was writing yet another paper. If there was ever a contest for the ideal wife, Jeanne would definitely get my vote. I am also very thankful to my parents, Hector and Shang-Foon, for their love and the perfect role models they have provided. They have shown me that hard work and commitment to one's beliefs will result in much satisfaction.

Dedication

To my wife, Jeanne, and our children, Justine, Lauren, and Emerson

Contributions

Listed below, we summarize the contributions of this dissertation. The contributions are:

- the definition of an abstract machine model for the Super-Actor Machine and the accompanying Super-Actor Execution Model.
- The proposition of a novel hybrid dataflow/von Neumann architecture—the Super-Actor Machine[66].
- The invention of a novel high-speed memory organization known as the “Register-Cache”[65, 67, 68]¹. It is employed in the new architecture to tolerate local memory latencies and reduce local memory bandwidth requirements in a multi-threaded architecture.
- The examination of compilation techniques for generating Super-Actor Machine code from a well-formed dataflow graph. And,
- the construction of a detailed simulator for verifying the effectiveness of the SAM on loop kernels typical of scientific applications, and the ensuing simulation study.

Contributions to the understanding and advancement of the McGill Dataflow Architecture (MDFA)—the architecture on which the Super-Actor Machine is based—also resulted from my doctoral studies, but they are not elaborated here in this thesis. Instead, those contributions are simply listed here:

¹Currently, there is a patent pending on the register-cache mechanism. The patent application is based on [65].

- the simulation study of the execution efficiency of the MDFA [48] which led to the identification of the excessive cumulative fine-grain synchronization cost problem.
- The study of the *Limited Balancing Technique* [45, 46, 49] as a compilation technique to address the excessive cumulative fine-grain synchronization cost problem.
- The proposal for extending the MDFA to handle concurrent function invocations which lead to the McGill Dynamic Dataflow Architecture Model [51, 47]. And,
- the construction of a prototype simulator for investigating the efficiency issues in the McGill Dynamic Dataflow Architecture [71]. Results from this simulator have further contributed to our understanding of dynamic dataflow executions.

Contents

Abstract	ii
Résumé	iii
Acknowledgments	v
Dedication	vii
Contributions	viii
1 Introduction	1
1.1 Two Fundamental Issues in Multiprocessing	3
1.1.1 The Memory Latency Problem	4
1.1.2 The Synchronization Problem	7
1.2 Addressing the Problems	7
1.2.1 The Dataflow Model of Computation	9
1.2.2 Existing Dataflow Architectures	12
1.2.3 Advantages of the Dataflow Concept	16
1.2.4 Problems in Paradise	16
1.3 Synopsis	17

2	The Problems	18
2.1	The Problem of Tolerating Local Memory Latencies	19
2.2	The Problem of Fine-Grain Synchronization Costs	21
2.3	Discussion	23
2.3.1	Multi-Threaded Architectures	24
2.3.2	Objectives	25
3	The Argument-Fetching Dataflow Model	27
3.1	The Argument-Fetching Principle	27
3.1.1	Argument-Fetching in a Multiprocessor Context	29
3.1.2	Previous Argument-Fetching Dataflow Work	31
3.2	The McGill Dataflow Architecture	32
3.2.1	The Program Format for the MDFA	34
3.2.2	Example Program Tuples for the MDFA	36
3.3	Why the MDFA?	39
3.3.1	Data Value Movement Analysis	41
3.3.2	The Cost of Bundling Data and Signal Processing Information . .	43
3.3.3	Cost Analysis of Conditional Expressions	45
3.3.4	Summary	45
3.4	Summary	46

4	A New Architecture	48
4.1	Addressing the Locality Issue	49
4.1.1	A New Execution Model and Novel Memory Organization	50
4.2	Format of Textual Information in the Pseudo-Code	51
5	The Abstract Model of the Super-Actor Machine	52
5.1	The Abstract Program Execution Model	53
5.1.1	The Super-Actor Graph	53
5.1.2	Examples of Super-Actor Graphs	68
5.1.3	Determinate Super-Actor Graphs	77
5.1.4	Discussions	80
5.2	The Abstract Machine Model	81
5.2.1	Symbols Used in the Models	82
5.2.2	The Basic Abstract Machine Model	85
5.2.3	Instruction Set of the Abstract SAM Model	89
5.2.4	The Intermediate Abstract Model	95
5.2.5	The Advanced Model of the SAM	108
5.3	Summary	113

6	The Architecture of the Super-Actor Machine	115
6.1	Mechanisms Needed for Super-Actor Processing	116
6.2	A Processing Element of the Super-Actor Machine	118
6.2.1	Tuple Definitions	120
6.2.2	The Register-Cache Architecture	124
6.2.3	The Actor Scheduling Unit	135
6.2.4	The Actor Preparation Unit	139
6.2.5	The Long-Latency Actor Execution Unit	145
6.2.6	The Super-Actor Execution Unit	148
6.2.7	Local Main Memory	154
6.3	Summary	158
7	Generating Code for the Super-Actor Machine	160
7.1	Well-Formed Dataflow Graphs	162
7.1.1	Encapsulators in a Dataflow Graph	163
7.2	The Partitioner	166
7.2.1	The Partitioning Phase	167
7.2.2	The Location Assignment Phase	177
7.2.3	Deadlock-Free Super-Actor Graphs	178
7.2.4	An Example Partitioning	179

7.3	Considerations in Partitioning	183
7.3.1	Machine Specific Constraints for Partitioning	183
7.3.2	Some Optimizations in Partitioning	185
7.4	The Translator	195
7.4.1	An Example	197
7.5	The Assembler	199
7.5.1	Calculating the Count Signal Weights	199
7.5.2	Removing Merge Nodes	201
7.5.3	Packing the Attributes, Instructions, etc.	202
7.6	Summary	203
8	Simulations	204
8.1	The Simulated Architecture	205
8.2	The Test Programs	207
8.2.1	Simulation Results	210
8.2.2	A Performance Measure for Comparisons Between Various Architectures	216
8.3	Summary	218

9	Related Work	219
9.1	The Denelcor Heterogeneous Element Processor	219
9.2	Horizon and the Tera Computer	221
9.3	The Hybrid Dataflow/von Neumann Architecture	222
9.4	P-RISC and *T	223
9.5	The EM-4	225
9.6	APRIL	225
9.7	A Modern "Static" Architecture	226
9.8	The Decoupled Graph/Computation Architecture	227
9.9	The LGDG Architecture	228
10	Conclusion	230
10.1	Future Work	232
A	Dataflow Software Pipelining	233
A.1	Dataflow Software Pipelining for Idealized Machines	235
B	Functions of the Advanced Machine Model	237
C	An Assembly Language for the SAM	241
C.1	Super-Actors	241
C.1.1	Support-Actors	243
C.1.2	Long-Latency Actors	244
C.1.3	Miscellaneous	247

D SAMAL Code for the Benchmark Programs	248
D.1 SAXPY	249
D.2 SAXPBYPC	251
D.3 Livermore Loop1	254
D.4 SAXPY3	256
Bibliography	272

List of Tables

8.1	Results for SAXPY, SAXPBYP, SAXPY3, and Loop1.	211
8.2	Varying the maximum number of active super-actors.	215

List of Figures

1.1	Types of multiprocessors.	2
1.2	Von Neumann-type processing.	3
1.3	Dataflow processing.	9
1.4	Successive snapshots of a dataflow execution.	10
1.5	A dataflow graph representing a conditional statement.	11
1.6	A static dataflow processing element.	13
1.7	A tagged-token dataflow processing element.	15
2.1	The trend in device development.	19
2.2	An abstract dataflow architecture.	22
2.3	A processing element of a multi-threaded architecture.	25
3.1	The argument-fetching and argument-flow principles.	28
3.2	Interprocessor communications for the argument-fetching and -flow principles.	30
3.3	The McGill Dataflow Architecture Model.	32

3.4	A program tuple.	37
3.5	A program tuple representing a conditional expression.	38
3.6	A program tuple representing a loop construct.	40
5.1	Components of a super-actor graph.	55
5.2	States of a super-actor instance.	60
5.3	A def encapsulator in a super-actor graph.	63
5.4	An if-then-else encapsulator in a super-actor graph.	65
5.5	A loop encapsulator in a super-actor graph.	67
5.6	Examples of super-actor graphs.	69
5.7	Conditional super-actor graphs.	71
5.8	A loop encapsulator in a super-actor graph.	73
5.9	Function applications in a super-actor graph.	75
5.10	The Basic Abstract Machine Model.	86
5.11	The Basic Memory Model.	86
5.12	Super-actors in the Intermediate SA Graph Model.	95
5.13	A super-actor graph with the new super-actor syntax.	97
5.14	Super-actor instance states in the intermediate machine model.	98
5.15	The intermediate abstract machine model of the SAM.	104
5.16	The intermediate memory model of the SAM.	105
5.17	The abstract machine model of the SAM.	111

5.18	The memory model of the abstract machine.	112
6.1	The Super-Actor Machine.	116
6.2	A processing element of the Super-Actor Machine.	118
6.3	The Register-Cache.	125
6.4	The registering process.	126
6.5	APU components which interface with the R-caches.	127
6.6	The Actor Scheduling Unit.	136
6.7	The Actor Preparation Unit.	140
6.8	The Support-Actor Execution Unit.	144
6.9	The L-Actor Execution Unit.	146
6.10	The Super-Actor Execution Unit.	148
6.11	The map of local main memory.	155
6.12	The program segment and data segment maps.	156
7.1	A possible organization of a compiler for the SAM.	161
7.2	The <i>def</i> and <i>if-then-else</i> encapsulators.	164
7.3	The <i>loop</i> encapsulator.	165
7.4	An example of an improper aggregation of actors which can lead to deadlock.	168
7.5	Example of a redundant arc.	173
7.6	A partitioning example.	180
7.7	Part 2 of the partitioning example.	182

7.8	An example where support-actors are used.	186
7.9	A super-actor graph for the parallelizing example.	189
7.10	The parallelized ('vectorized') super-actor graph.	191
7.11	A super-actor graph representing a 'vectorized' and software-pipelined loop.	193
7.12	The overlay map of the previous SA graph.	194
7.13	The overlay map for the partitioning example in figure 7.6.	198
7.14	An example of weighted count signals.	201
7.15	Encoding merge nodes into signal lists.	202
8.1	SA graph of unrolled-4 version of SAXPY.	209
8.2	SAXPY3.	210
8.3	Execution pipe utilization rate for different versions of SAXPY.	212
8.4	Execution profile for the 8-loop unrolled-16 version of SAXPY.	213
8.5	Varying memory access times for the 8-loop unrolled-16 version of SAXPY3.	214
A.1	An example of a dataflow software pipeline.	234
A.2	Balancing a dataflow software pipeline.	236
D.1	Overlay layout for SAXPY.	249
D.2	Overlay layout for SAXPBYP.	251
D.3	Overlay layout for Livermore Loop1.	254
D.4	Overlay layout for SAXPY3.	257

Chapter 1

Introduction

Since the beginning of the computer era, multiprocessor systems have been heralded as solutions to the ever increasing computational needs of the user community. The promise of multiprocessors lies in their ability to attain shorter execution times for parallelizable tasks than if they were executed on a uniprocessor system. In multiprocessor systems classified as MIMD machines (Multiple-Instruction stream Multiple-Data stream) [39], multiple independent processing elements (PEs) linked together by some interconnection network work together to process a single task.¹ Memory units for storing program and data can be directly attached to each PE (node) or they can form separate units on the network. Typically, such memory placements in the system differentiate the system as a *distributed memory multiprocessor* or a *shared memory multiprocessor*, respectively (fig. 1.1). There has been work on *distributed shared memory* systems [88] where physical memory units are distributed amongst individual PEs, but through software and/or hardware means, a single logical memory space is presented to the user. Such work attempts to address the common complaint that distributed memory machines are difficult to program while at the same time, retaining the scalability property of distributed memory multiprocessors.²

¹Henceforth, the word multiprocessors will mean MIMD machines; SIMD (Single-Instruction stream Multiple-Data stream) machines are not included.

²The novel architecture presented in this dissertation utilizes a distributed shared memory system.

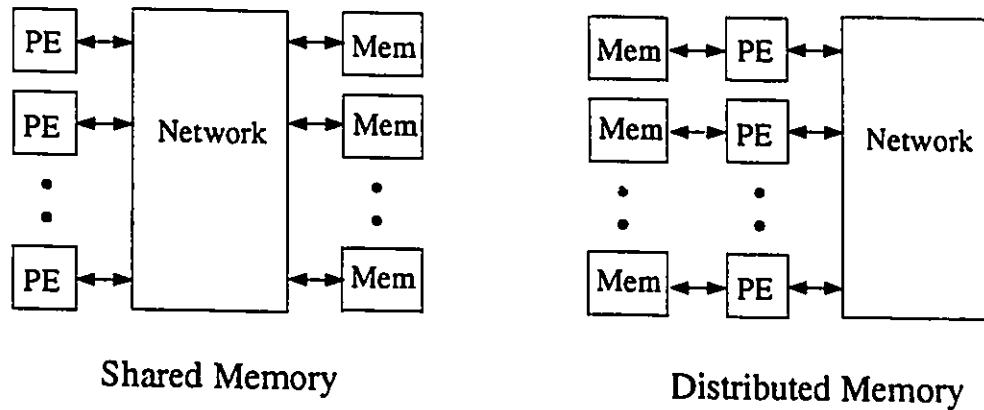


Figure 1.1: Types of multiprocessors.

The power of a multiprocessor is realized by breaking down a task into multiple collaborating processes (this is the act of *parallelizing* the task), distributing the processes to individual PEs in the multiprocessor system, and letting the PEs compute in parallel. The *speedup*—the ratio of the time for executing the task on a single processor to the time it takes on the multiprocessor—is dependent on the amount of useful parallelism in the task and on the underlying hardware: the interconnection network and the PEs.

Generally, today's multiprocessing systems, e.g., the BBN Butterfly, the Intel Hypercube, the Meiko Computing Surface, etc., use proprietary networks and off-the-shelf microprocessors.³ In designing these multiprocessors, much effort has been expended on the design of the networks, whereas the processors used as PEs are the same type as those found in uniprocessor systems. The question which begs to be asked is: does using processors typical of those found in uniprocessor systems as PEs in a multiprocessor impose a limitation on the system from attaining maximum speedup? The answer to that question is an unequivocal yes! The reason is that such processors are based on an inherently sequential execution model—the von Neumann model of computation. In the von Neumann model of computation, the abstract machine model consists of a Central Processing Unit (CPU)

³The transputer microprocessor used in the Meiko Computing Surface consists of a processing unit and a network interface unit integrated on one chip. The unique feature is the onchip network interface which we consider to be a part of the interconnection network.

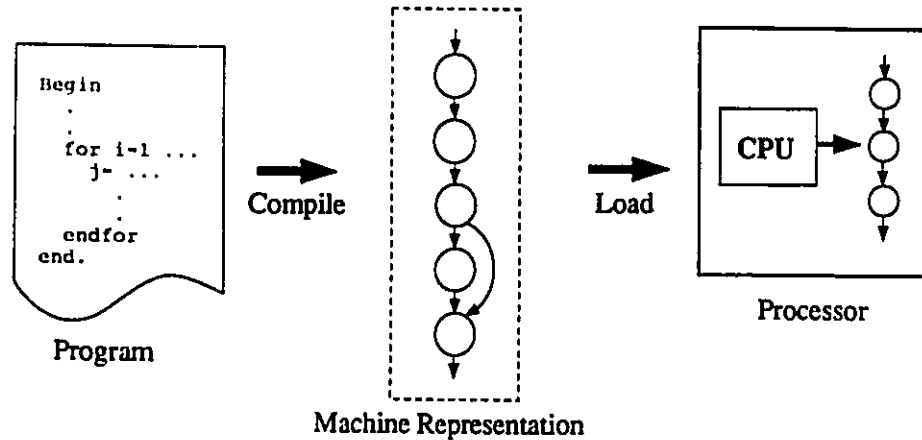


Figure 1.2: Von Neumann-type processing.

containing a *program counter*. The program counter is used to step through the program—a totally ordered list of instructions generated by the compiler—and the CPU processes the data according to the instruction currently designated by the program counter (fig. 1.2). When such processors are used in a multiprocessing context, inefficiencies will arise due to *processor idling*, i.e., the time when a processor does no useful work while waiting for a memory request to be fulfilled or a synchronization event to occur.

1.1 Two Fundamental Issues in Multiprocessing

Arvind and Iannucci [15] have eloquently argued that von Neumann-type multiprocessing systems are prone to performance degradations arising from two fundamental problems:

- memory latencies for non-local memory access, and
- synchronization costs—costs incurred when context-switching from a waiting process to a ready process instead of idling for synchronization events.

1.1.1 The Memory Latency Problem

Memory latency is defined as the time between the issuing of a memory access request and the fulfillment of that request. When a PE wants to access non-local memory—memory which is not within the same PE—the memory access must traverse the interconnection network resulting in a latency which is generally greater than that for a local memory access. Furthermore, as the number of processing elements in the system increases, the latency can become greater and more varied. One way to hide⁴ this memory latency is to arrange the thread of instructions such that instructions following the memory accessing one are independent of the data requested, thus the CPU can continue executing instructions while the memory request is serviced. This is termed the *delayed load* technique and is commonly employed in compilers for RISC (Reduced Instruction Set Computer) processors [92, 63]. This technique would yield limited benefits since it may be difficult to find enough independent instructions to follow the memory accessing one. One reason for the difficulty is that when a memory accessing instruction is encountered, a compiler usually looks for an independent instruction in the vicinity of that instruction, where the vicinity is generally restricted to the block of instructions belonging to a function/procedure which contains that memory accessing instruction. Furthermore, the latency for remote accesses can be long and unpredictable, thus limiting the compiler's effectiveness in knowing how many independent instructions should be inserted after a delayed load. Some interconnection networks can guarantee a fixed response time for a remote access, however, the latency still remains long relative to the cycle time of the processor and having the compiler fill two or more delay slots after a memory load instruction is difficult—Gross[58] has found that a compiler can fill 70% of the first delay slot with a useful instruction, 30% for the second, and 10% for the third. If there are not enough executable instructions to overlap a non-local memory request, the processor has no recourse but to execute no-ops (no operation instructions).

Hardware solutions to tolerate memory latencies use mechanisms similar to a reservation

⁴Note that we can only hide the effects of memory latency, and we cannot eliminate it altogether due to technological constraints.

station⁵ [111, 113] or load/store queues as used in *decoupled architectures*[102]. These devices allow instructions to be issued continuously as long as they do not require the data from a pending memory request. The reservation station mechanism checks whether instructions can proceed after a memory request rather than relying on a compiler to do it at compile time. Decoupled architectures which use load/store queues allow an instruction to be processed as long as its operands are in registers or on top of the load queue—operands are fetched into the load queue and later popped off when needed (an empty load queue implies that the data has not been fetched yet). One of the premises of run time solutions is that memory latencies are variable, thus a statically determined instruction schedule would not be ideal. In general, reservation station-like mechanisms can only support a limited number of instructions issued out of order which can be less than that required to hide a non-local memory access. The major disadvantage of load/store queues is that they cannot handle memory requests returning out of order. Another problem with both of these hardware solutions is that when a context-switch⁶ must be performed (e.g., the current process must wait till some event happens and decides that another process should be executed instead), the processor must wait until all pending instructions in the reservation stations (load/store queues) have been processed before the next process can commence, thus increasing the synchronization overhead.⁷

Another means of decreasing the effects of memory latency is to use high-speed buffers close to the CPU in the form of registers and caches [103], i.e., to organize the memory into a hierarchy of small fast memory close to the CPU, and slower and larger memory farther away. Such an organization is used to exploit the *principle of locality* which states that a program tends to exhibit two phenomena during its sequential execution:

- *temporal locality*, that is, when a memory location has been addressed, it is likely to be addressed again in the near future, and

⁵Reservation stations are also used to allow a processor to perform out-of-order execution to avoid data hazards; hazards which can arise when multiple instructions want to modify/read a piece of data simultaneously.

⁶Context-switching, which is implemented via software routines, involves the selection of an executable thread and the swapping to and from memory values like the program counter, temporaries stored in registers, etc.

⁷The *synchronization overhead* is defined in section 1.1.2.

- *space locality*, i.e., when a memory location has been addressed, neighbouring locations are likely to be addressed in the near future.

Registers are the fastest memory in the hierarchy and are programmer (compiler) visible, that is, they can be directly addressed and the programmer (compiler) is responsible for managing their use. Since the instruction sequence is ordered at compile time, techniques can be employed in analyzing the lifetimes of scalar variables such that register usage can be effectively managed by performing *register allocation* [22, 23]. Caches are slightly slower memory than register memory, are usually not programmer (compiler) visible, and work on the principle of keeping the most recently referenced data in the high-speed buffer. When a memory request arrives at the cache, the cache memory is associatively searched for the requested address, i.e., address tag(s) associated with the cache's contents are matched with the incoming request's address. If it's there, the time for the memory response will just be the delay to access the cache. But if it's not there, the response time will be the associative search time plus the time to go to main memory to fetch the requested data.⁸ All this management of cache memory is handled by dedicated hardware so that cache access times are as close to the CPU cycle as possible.

To tolerate increasing memory latency, larger register files can be employed, but this leads to a larger processor state which in turn, leads to a larger context-switching overhead. As for using caches, there is the problem of *cache coherence* [21] where multiple copies of the same data can be kept in the caches of different PEs and all these copies must be kept consistent. This problem can be solved with hardware mechanisms which implement some coherency protocols [8] and work best in *shared bus* multiprocessors (a system where a common bus is used as the interconnection network). However, questions concerning the limitations imposed by these coherency protocols on the system scalability have been raised. Furthermore, it might appear that using a cache will not influence the switching overhead as does the use of registers, but the time it takes to bring in new blocks for the executable process and possibly swapping them with blocks of pending processes should be accounted for—time which should be added to the switching overhead. With these

⁸If the cache is filled, then some cache block must be replaced. If some data in that block has been altered, then the block must be written back to memory, thus possibly increasing the response time.

problems, it is not surprising that von Neumann processors can spend an inordinate amount of time doing no useful work when employed in a multiprocessor context.

1.1.2 The Synchronization Problem

Associated with a process is its *state* which contains things like a program counter, processor status bits, and temporary values stored in registers. When a process must be suspended due to some event and another ready process executed, some instructions are required to select the ready process, store the state (context) of the suspended process to memory and bring in the context of the ready process from memory. The time to perform the context-switch, i.e., suspend a process and start executing another, is termed the *synchronization overhead*. This synchronization overhead can be decreased if the size of the process state is decreased, but the only reducible entities of the state are the number of assigned registers and the blocks of memory in the cache which belong to the process. Limiting the number of memory blocks for each process is generally not within the power of the programmer, so this is a very difficult task. The reduction of the number of assigned registers may worsen the memory latency problem since a process might have to access main memory more often. Another solution would be to keep the states of multiple processes in fast memory so that the synchronization overhead can be decreased. Architectures like the HEP [105] have done just that, but such architectures have swayed far away from the von Neumann model and have spawned a separate architectural class called *multi-threaded architectures* (These architectures will be described in a later section.) So for von Neumann based multiprocessors, synchronization costs and memory latency are intertwined and vexing problems which appear to be its nemeses forever.

1.2 Addressing the Problems

For the next generation of processing elements, a multitude of architectural options to solve the two fundamental problems will become feasible. Furthermore, with the emergence of

ULSI (Ultra Large Scale Integration) technology which promises the capability of 50–100 million transistors on a chip [54], many camps are debating how such enormous hardware parallelism should be exploited. The method of squeezing multiple processors onto a chip immediately and obviously comes to light (e.g., the Micro 2000 project at Intel which proposes to squeeze four processors on one chip). However, the two fundamental problems still remain, albeit they are pushed to a smaller scale. Another possibility is the combination of *superscalar* (ability to issue multiple instructions per cycle) and *superpipelined* (arithmetic and logic units performing base operations such as integer add are pipelined with multiple stages) features into one processor [76]. This solution still does not satisfactorily address the two fundamental problems as stated above. The superscalarity permits the processor to issue memory request(s) concurrently with ALU operations, so the memory latency problem can be somewhat alleviated by the possible issue of multiple memory accesses and the overlapping of memory accesses with ALU operations. However, there can be a limit to the instruction-level parallelism when the programming model and underlying execution model is sequential [118]⁹, a limitation which restricts the effectiveness of overlapping memory requests with ALU operations. Moreover, the problem of context-switching overhead on synchronization events remains. In fact, the switching overhead could be larger due to the increased contextual information required in supporting superscalarity.

With the inherent problems of utilizing von Neumann processors as computing engines in a multiprocessing system, some leading edge research in parallel processing hardware are focused on overcoming the two fundamental problems by investigating different models of computation. One such model which has received much attention is the dataflow model of execution [53, 10, 4, 28, 115, 109, 27]. Multiprocessor systems consisting of dataflow processing elements appear to circumvent the problems associated with von Neumann-type multiprocessor systems because they are based on an inherently parallel model of execution. In dataflow processing, the user's program is first compiled into a *dataflow graph* in which the directed arcs between actors (instructions) indicate the flow of data, not the flow of control. There are no extraneous dependency arcs for imposing an order of instruction

⁹The programmer expresses the application in a sequential fashion which makes the compiler's task of finding the inherent parallelism that more difficult. Superpipelining further increases the burden on the compiler to find more instruction-level parallelism.

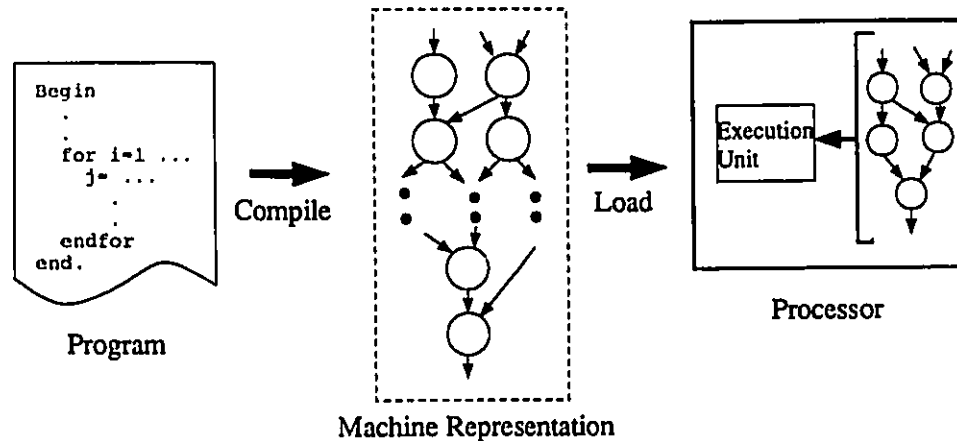


Figure 1.3: Dataflow processing.

execution; impositions which can hide the implicit parallelism from being exploited by the underlying machine. This dataflow graph, actually a machine code representation of the graph, is then loaded into a dataflow machine and the execution of instructions will be governed by the arrival of data flowing into their input arcs; not by some program counter (fig. 1.3).

1.2.1 The Dataflow Model of Computation

In the following subsections, we will review the dataflow model and its associated architectures so that the readers may get a taste of the simplicity and elegance of the model.

The dataflow model describes computations in terms of locally controlled events; activations (*firings*) of individual instructions (*actors*) [27]. A group of actors linked by dependency arcs forms a dataflow graph (a program module in conventional terminology), where the actors are nodes and the arcs indicate destinations of an instruction's result. The decision to fire an actor is based on the availability of the operands of that actor. Therefore, there is no concept of a single locus of control (program counter concept in von Neumann

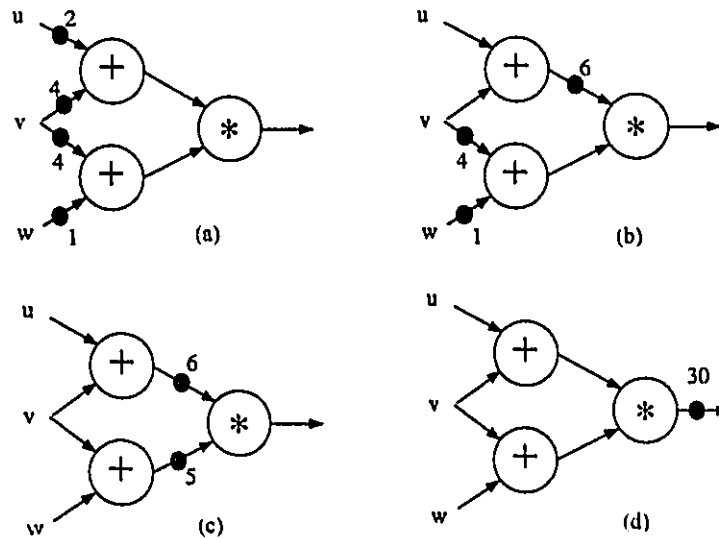


Figure 1.4: Successive snapshots of a dataflow execution.

computers). Data flows through the dataflow graph on entities called *tokens*, and when all the input tokens of an actor have arrived, that actor becomes *enabled* and is subsequently fired. Instruction-level parallelism can be easily exploited because multiple actors can be fired simultaneously.

Let us look at a program example for the expression $(u + v) \times (v + w)$. Figure 1.4 details successive snapshots of the graph as data tokens are flowing through it. In figure 1.4(a), data tokens representing values of u , v , and w are presented; (b) shows the result of the $+$ actor firing and producing a token on its output arc; (c) shows the other actor firing; and finally (d) indicates the completion of the multiply actor and creation of its output token.

A dataflow graph can represent a conditional expression with a *switch* actor, *decider* actors and *T-* or *F-gate* actors. For the conditional expression

if $p(y)$ then $f(x, y)$ else $g(y)$ endif

the dataflow graph is shown in figure 1.5. The decider actor is represented by the diamond

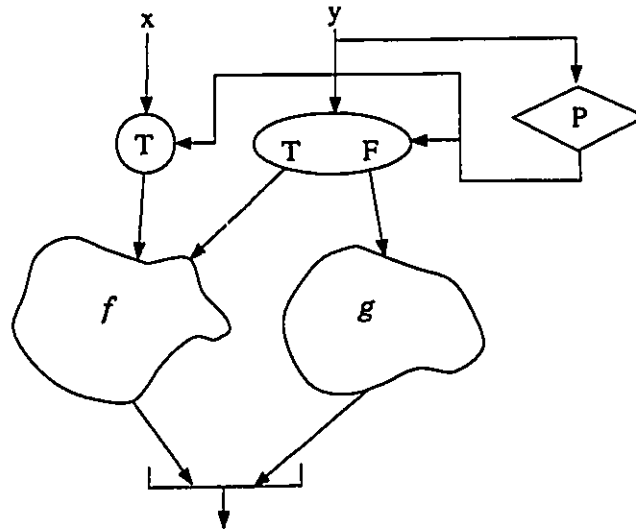


Figure 1.5: A dataflow graph representing a conditional statement.

shaped node, the T- gate by the circular node, and the switch actor by the oval-shaped node. A dataflow graph constructed with the well-formed graph constructs [27] like the elementary actor and the conditional graph block, has the expressive power for representing all of the common features of a computer language, for example, conditionals, iterations, recursions, etc.

A nice property of the dataflow model is that no matter which order the actors are executed (as long as they follow the rules of firing), the model guarantees that given a set of inputs, a unique set of outputs is produced from the execution of a well-formed dataflow graph [31]. This *determinacy property* indicates that the dataflow model is functional in nature, i.e., the input/output behaviour is unaffected by the history of computational events, and is a boon to programmers who must deal with parallel processing where outcomes must be repeatable.

1.2.2 Existing Dataflow Architectures

A typical architecture which supports the dataflow paradigm is a collection of dataflow processors (PEs) linked by some routing network(s). In each processor, there is a 'parking area' for the actors (instructions) waiting for their input tokens (operands). Once all the necessary tokens have arrived for the node, the node is activated for execution (sent off to the execution unit in the processor). From there, a token or tokens are generated. A token is then sent through the routing network(s) if it is destined for a remote PE, otherwise, it is routed to the local PE. This process is repeated until all nodes which can be fired are executed.

There are two mainstream dataflow architectures which have been extensively researched: the static dataflow architecture[34], and the dynamic or tagged-token dataflow architecture [17, 10]. In the next two sections, we describe the structures of *pure dataflow architectures*, that is, architectures which require each basic instruction (e.g., an add, compare, etc.) be scheduled for activation via a mechanism which checks for the arrival of the necessary operands.

Static Dataflow Architecture

The static dataflow model specifies that there are at most one data token on each dependency arc. This restriction is achieved by the following execution rules:

- an actor is fired if and only if its input arcs have all received a token and its output arcs are all empty; and
- when the actor is fired, it removes one token from each of its input arcs, and notifies its predecessor nodes of the completion of the execution by putting a token on each of its output arcs.

This restriction of one token per arc provides an elegant manner for pipelining the data (This is the basis of dataflow software pipelining which is reviewed in appendix A). Function

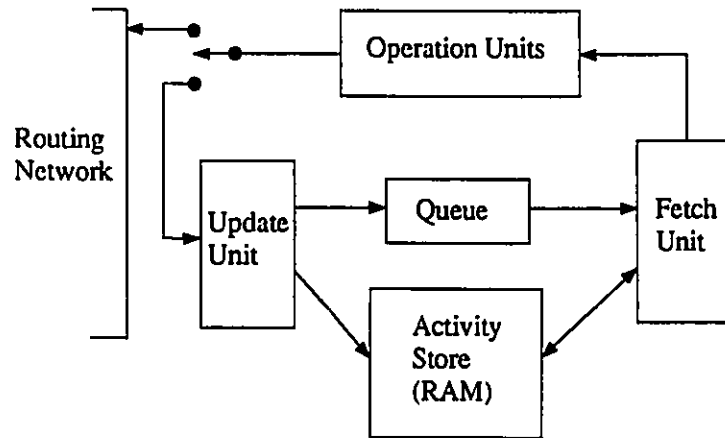


Figure 1.6: A static dataflow processing element.

invocations are generally supported by in-line copies of the function data graph, although limited function sharing is possible [112]. General function recursions are not directly supported, however tail recursions can be supported through iteration loops. The simplicity of the model and the architecture makes implementing a static dataflow machine feasible and effective in handling many problems of array computation in numerical supercomputing [29].

In general, a static dataflow PE can be represented by a circular pipeline as shown in figure 1.6. The update unit is responsible for updating instruction templates representing dataflow actors. The templates assigned to the PE are stored in the activity store. When a token arrives for a given actor, the template is updated and if it is firable—all its tokens have arrived—its address is put onto the queue. The fetch unit is responsible for fetching the information of firable actors (pointed to by the addresses in the queue) from the activity store and sending them off to the operation units. Once an actor has been fired, its result is packaged into token(s) and sent either to a PE on the network or back to the update unit within the PE.

Machines built to support this static dataflow model include: Dennis' Engineering Model [36], the LAU [93], etc. References for other static dataflow machines can be found

in [117].

Tagged-Token Dataflow Architecture

The tagged-token dataflow model allows multiple occurrences of tokens on a dependency arc; therefore, it requires that the tokens be coloured (tagged) to indicate the invocation instance. Thus this model allows function recursion and multiple invocations of a function with one copy of the function graph. (*Loop unraveling* [11], where a loop is unrolled dynamically at run time, is also supported.) However, with this generality comes the need for specialized hardware—typically implemented with associative memory—for token matching. A major problem with this model has been the overhead of token matching and the overhead of resource management required for handling large amounts of parallelism at run time. A solution to the resource management problem has been proposed by Arvind and Culler [24, 11] and a solution to avoid colour matching hardware has been detailed to alleviate but not completely eliminate the token matching (fine-grain synchronization) overhead [89].

A tagged-token dataflow PE can also be represented by a circular pipeline (fig. 1.7). The PE consists of: a waiting-matching section where incoming tokens are matched to their partners via associative hardware (the newer generation of tagged-token architectures uses only a directly addressable memory in a scheme called the *Explicit Token Store* [89]), an instruction fetch section which fetches the instruction of the actor once the actor has received all its operands, the ALU for executing the instructions, and an output section for generating tokens with the results of the instructions and their associated tags.

Machines built to support the dynamic dataflow model include: the Manchester Dataflow Prototype [59], the Sigma-1 [64], the Monsoon [89]¹⁰, etc. References to other dynamic dataflow machines can also be found in [117].

¹⁰Actually, the Monsoon also supports hybrid dataflow/von Neumann processing.

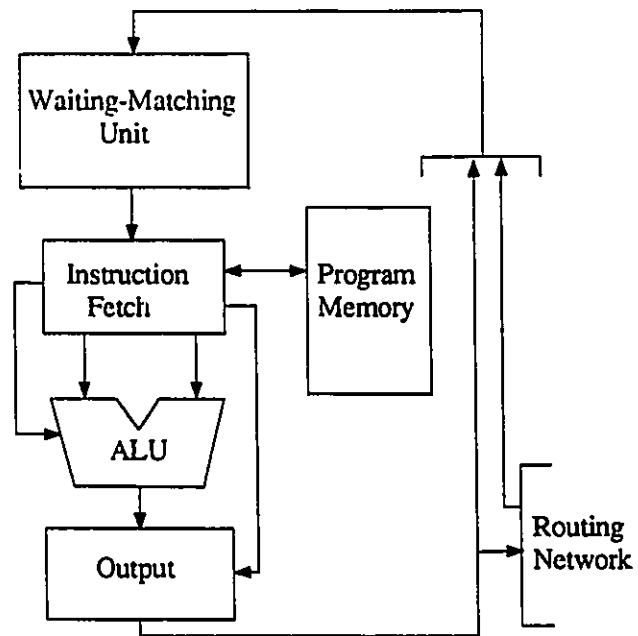


Figure 1.7: A tagged-token dataflow processing element.

1.2.3 Advantages of the Dataflow Concept

The nature of the dataflow model stipulates that a single processor support multiple candidate instructions for concurrent execution. By having multiple instructions ready for execution, processor idling can be minimized by rapidly switching to other ready instructions. Unpredictable latencies associated with non-local memory requests can be hidden by quickly switching to another executable instruction instead of waiting for the memory response; in this sense, the request is done *off-line*. That is to say, some other unit other than the execution unit is responsible for the memory access while the execution unit can continue processing other instructions which do not require a non-local memory access. When the request has been fulfilled, the memory requesting instruction becomes executable again and is put back into a pool (or queue) where it waits to be processed by the execution unit.¹¹ As for synchronization costs during context-switching, the switch can be rapidly performed in a dataflow processor because the contextual information (state) associated with an individual instruction is minimal, and the next ready instruction is easily moved from the ready pool to the execution unit.

The above arguments in favour of dataflow processors are also elaborated by Arvind and Iannucci in their classic paper dealing with the two fundamental problems of multiprocessing [14].¹²

1.2.4 Problems in Paradise

Pure dataflow computers, though, are not free of problems. Since Gajski et al.[40] voiced their reservations about pure dataflow machines, most concerns brought forward in that article have been addressed save for a couple of serious ones.

¹¹This type of memory request is commonly referred to as a *split-phase transaction* [16].

¹²They have since written five more versions of the same paper.

1.3 Synopsis

In the next chapter, we present the problems of pure dataflow implementations, and outline the objectives of this dissertation. In chapter 3, we review the *McGill Dataflow Architecture* (MDFA) since it forms the basis of our research work. The MDFA is based on the *argument-fetching principle* which allows it to address one of the problems raised in the next chapter; thus the architecture model, along with the argument-fetching principle is detailed. Chapter 4 introduces the work on the Super-Actor Machine—a hybrid dataflow/von Neumann architecture—which attempts to address the problems as outlined in chapter 2. In chapter 5, we describe the abstract model of the SAM which serves as the SAM's behavioural specification as well as a foundation for future compiler and hardware development. The architecture of one processing element of the SAM is detailed along with the design of the register-cache mechanism—a high-speed memory organization tailored for hybrid dataflow/von Neumann computing (multi-threaded computing)—in chapter 6. Chapter 7 examines some compiling techniques for generating code for the SAM. To examine the performance characteristics of the SAM, experiments with a detailed simulator were performed and their results are presented in chapter 8. This architecture is put into perspective with a review of related multi-threaded architectures in chapter 9. Finally, we conclude this dissertation by summarizing this thesis and outlining future work in chapter 10.

Chapter 2

The Problems

As was pointed out in the Introduction, two fundamental problems which von Neumann multiprocessors fail to simultaneously address are: tolerating non-local memory latencies and high synchronization costs arising from high context-switching overheads. In pure dataflow implementations, there are two parallel problems, namely:

- *tolerating local memory latencies*, that is, how does the PE architecture hide the latency of a local main memory accesses from the execution pipeline(s)? Put another way, how does the architecture provide fast instruction and operand access to keep up with the execution pipeline(s)? And,
- *tolerating fine-grain synchronization costs*.

For a dataflow machine to efficiently exploit ‘fine-grain’ parallelism, fast operand access [13, 47], and tolerable fine-grain synchronization costs [48, 12] are crucial.

In this chapter, we examine the two parallel problems in pure dataflow implementations and propose the use of a *hybrid dataflow/von Neumann architecture* as an approach to solving the problems facing von Neumann and pure dataflow multiprocessor systems.



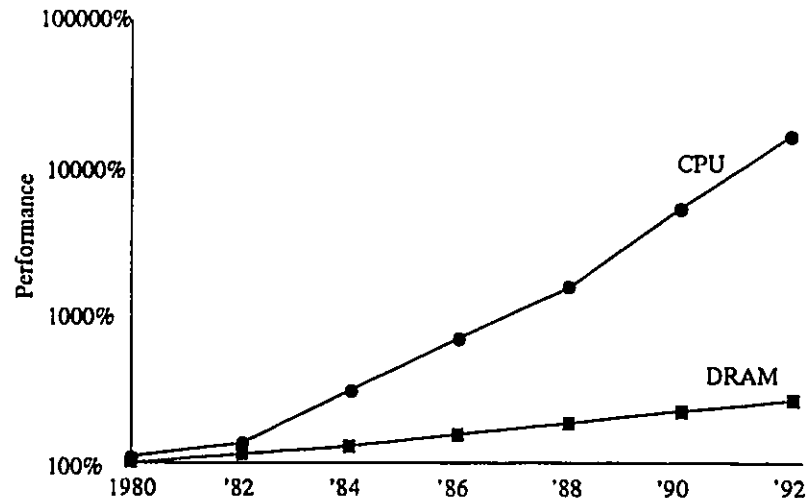


Figure 2.1: The trend in device development.

2.1 The Problem of Tolerating Local Memory Latencies

If we look at today's trend in device technology development (fig. 2.1—duplicated from [62]), we note that the processor speed—the basic cycle time—is increasing at a rate much quicker than the memory access time of Random-Access Memory (RAM). The main emphasis of RAM chip development is the squeezing of more and more memory cells onto a single die, whereas processor chip development emphasizes smaller and smaller clock cycles at the same time as squeezing more and more peripheral circuits onto the same chip. Without a radical enabling technology to address the memory access time in RAM chips, the gap will continue to grow. In modern von Neumann machines, a programmer (compiler) visible register file and a high-speed cache memory are used to substantially diminish the impact of local memory latencies—mechanisms which have not been examined for dataflow machines until recently [89].

The absence of a single locus of control in the pure dataflow model of computation presents new challenges to conventional register allocation techniques. Effective reuse of high-speed registers in pure dataflow machines, that is, deciding which values should

be kept in registers and which should not, is not obvious. As for a conventional cache, allowing firable actors from different parts of the dataflow graph to execute in any order can have detrimental effects. The reason is that a cache is designed to exploit the principle of locality exhibited in the execution of a totally-ordered list of instructions; firable actors executed in any order may diminish that locality effect. One solution is the use of large fast memory—memory with register-like speed. However, the huge size of the register file can lead to increases in access latency and make them less attractive. A variation on the theme is the use of multiple register sets for storing active stack frames¹ as detailed in the Monsoon architecture[89]. A problem with this solution is that structure memory elements cannot be stored in frames. The solution proposed in this thesis involves a moderate size register file (say 1K to 4K words) which uses hardware guided by software directives for register management. However, the effectiveness of the devices in the Monsoon and the one proposed in this thesis rely on the principle of locality, an effect which may be diminished in the pure dataflow model of execution.

Another competitive solution to tolerating local memory latencies would be the use of *parking stores*² for each memory accessing stage in the execution pipeline so that *local memory latencies* can be tolerated. A parking store mechanism continuously accepts instructions and issues memory requests for them. Pending instructions are put in a *parking area*, and only when an instruction has its requests fulfilled is the instruction permitted to leave the parking store and advance to the next stage. If the memory system has a varying access time (e.g., the memory system is hierarchical and contains a cache³), then instructions can leave the parking store out of order. These memory latency hiding devices are limited in their effectiveness, because it may restrict the effective utilization of the pipeline and the scalability of the basic pipe beat. Introducing parking stores into the execution pipe can limit the reduction of the basic pipe beat, and in the worst case, it might even increase it. The reason is that the parking store must be associatively searched when

¹A *stack frame* is a contiguous set of memory locations used to contain a function activation's temporary values.

²In [71], the parking store is called a *reservation pool*. In fact, a parking store functions similarly to a reservation station.

³In this scheme, lockup-free caches [100, 79] would be used. A lockup-free cache is a conventional cache with the following improvement: a cache miss will not necessarily cause the execution pipeline to freeze, and other memory requesting instructions can be accepted by the cache while a cache miss is processed.

looking for a position to park a pending instruction. Granted that with today's technology, the associative search space can be small—on the order of four to eight locations—but as the speed differential between the processor and local main memory increases, the parking store will have to be larger in order to tolerate the increased local memory latency. Some tradeoffs will have to be examined since a larger parking store can lead to a longer pipe cycle due to the increased associative search space.

From the above review, we can see that known mechanisms for tolerating local memory latencies in von Neumann machines are unsatisfactory for pure dataflow machines. This was one motivation in carrying out the research of this dissertation.

2.2 The Problem of Fine-Grain Synchronization Costs

Though the context-switching overhead of synchronizations per actor is minimal, the *total* fine-grain synchronization cost is another challenge for pure dataflow machines. The dataflow model stipulates that each and every instruction monitor the arrival of operands and/or signals for firing information. This implies that the machine must perform a synchronization operation⁴ for every waiting instruction which receives a datum (or signal). In [48], we reported our experimental studies which show that the fine-grain synchronization requirements can easily overwhelm the underlying machine and lead to a considerable degradation in PE utilization. Let us see why this can happen. A typical dataflow PE is a circular pipeline which can be separated into an *execution unit* responsible for the actual execution of the actor's instruction, and a *scheduling unit* responsible for signaling other actors that an actor has been executed and for determining which actors are fireable⁵ (fig. 2.2). Let us assume that the execution and scheduling units are fully pipelined, i.e., each one can accept an input every cycle, and the memory accesses are ideal. In order to keep the execution unit filled with useful work, the throughput of both units must match with each other (since

⁴This requires architecture support of one counting semaphore-like operation per instruction activation. Binary semaphores used in conventional dynamic dataflow machines are a special case of counting semaphores.

⁵This can be easily visualized from the two figures of generic static and dynamic dataflow architectures.

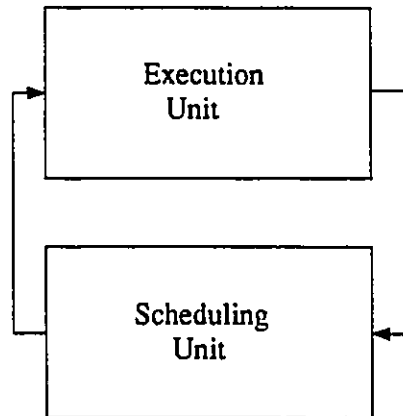


Figure 2.2: An abstract dataflow architecture.

they are hooked up as a circular pipe). However, when the average fan-in/fan-out factor of actors is greater than one, then the scheduling unit must have a throughput equal or greater than that of the execution unit in order to keep it busy. This will require a scheduling unit capable of processing more than one signal every cycle, thus leading to greater hardware complexity and increased costs as compared to a scheduling unit which only has to have a throughput less than the execution unit. We have called this, the *excessive cumulative fine-grain synchronization cost problem*.

Another fine-grain synchronization problem is related to a group of actors with low fan-in/fan-out factors (a fan-out of one indicates sequential code). For these groups of actors, the order of operand/signal arrivals can be determined a priori (i.e., at compile time and if the execution of the sequential thread is performed on one PE), so performing the fine-grain synchronization check for actors within the group is inefficient and unnecessary since the actors can be arranged in a totally ordered list and executed sequentially. By arranging groups of actors into ordered sequences, there is no need to *universally* schedule all actors via the fine-grain synchronization hardware. Nevertheless, pure dataflow machines must schedule all actors in this manner, and we have termed it the *universality of fine-grain synchronization problem*.

One last problem deals with the implementation of fine-grain synchronization support mechanisms. In many tagged-token dataflow architectures proposed to date, the data for synchronization (the *semaphore value* which guards the execution of the actor) is embedded along with the operand values in the data structure which represents a token. As a result, the scheduling mechanism and execution pipeline in a processor design are tightly coupled—a *tight-coupling* of two units implies that a link between the units has no buffering mechanism. This causes unnecessary restrictions like the replication of data in multiple tokens, and the tight-coupling of the semaphore update and operand fetch hardware; a coupling which may lead to unwanted pipeline bubbles [90].⁶ This problem is called the *implementation of fine-grain synchronization support problem*.

2.3 Discussion

In view of the fine-grain synchronization problems and the problems of tolerating local memory latencies, we decided to investigate a different class of architectures, called the *hybrid dataflow/von Neumann architecture*, for its applicability in addressing the issues. Hybrid dataflow/von Neumann architectures are a sub-class of *multi-threaded architectures* [60, 110].

Recently, investigating multi-threaded architectures has gained popularity in the dataflow research community as well as in the von Neumann research community [3, 7]. The problem of fine-grain synchronization, i.e., excessive cumulative costs and the inefficiency of universality, has spurred recent interest in hybrid dataflow/von Neumann architecture models [30, 43, 84, 57, 96, 73]. (The problem of implementing an efficient fine-grain synchronization mechanism has already been addressed in the McGill Dataflow Architecture—to be reviewed in chapter 3).

The attraction is simple: in a dataflow graph, each individual actor is the basic unit of work and scheduling quantum for the underlying machine, and fine-grain synchronization

⁶A *pipeline bubble* refers to the no-operation (NOP) instruction inserted into the pipeline by the instruction issuer due to some foreseen hazard in the execution pipe or because there were no executable instructions to dispatch at that moment.

is performed to schedule each instruction. However, some actors can be logically grouped into *threads* so that the cost of synchronizations can be reduced by performing the synchronizations only among the threads, while actors within a thread can be scheduled via the conventional and more efficient technique of sequencing with a program counter, à la von Neumann. A *thread* is both a compiler and run time entity: at compile time, program instructions are aggregated into totally-ordered sequences called threads, and at run time, instances of threads are created on demand and the activation of the thread instances⁷ are governed by the dependencies as stipulated at compile time.

2.3.1 Multi-Threaded Architectures

A *Multi-threaded architecture* [60, 110] is characterized by a node architecture which attempts to support multiple thread instances efficiently. Via the introduction of multiple copies of fast memory required for executing a sequential thread (e.g., the utilization of multiple sets of registers and multiple program status words, etc.), a mechanism for storing active thread instances, hardware for selecting which active thread instance to process, and some mechanism for processing synchronization events, a multi-threaded computer can exploit the parallelism at a desired level in an application (fig. 2.3). This exploitation is effected by pipelining and simultaneously executing instructions from different active threads, and by rapidly switching among the active threads. As we can see, the fine-grain synchronization problem associated with pure dataflow machines is less of an issue in these machines because instructions within a thread are sequenced with a counter while only the activation of thread instances require explicit synchronization via the synchronization mechanism. However, the issue of efficient synchronization support, i.e., the introduction of a minimal number of synchronization operations and the employment of scheduling mechanisms which can hide the synchronization overhead by overlapping the execution of useful instructions, must still be addressed. Furthermore, these architectures must effectively tolerate local memory latencies and provide fast operand access, though the challenge is a lesser one than it is for pure dataflow machines (ordinary caches can be used

⁷We will use the word 'threads' and 'thread instances' interchangeably to describe instances of threads.

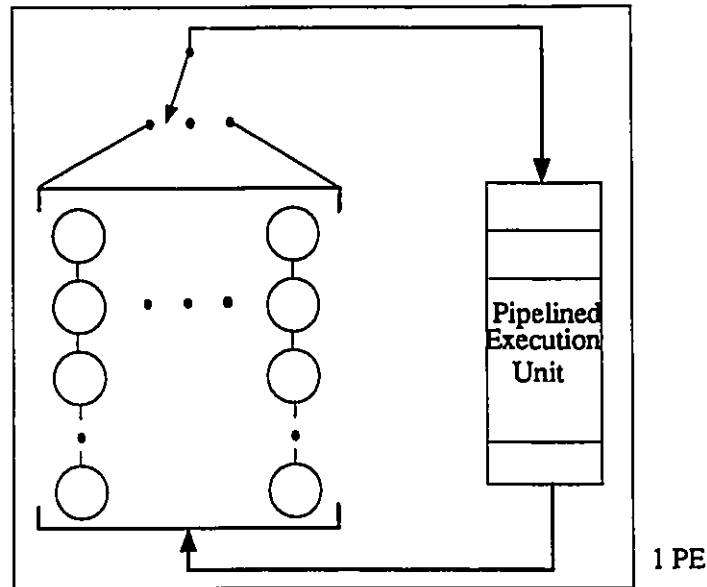


Figure 2.3: A processing element of a multi-threaded architecture.

to exploit the locality effect when executing the totally-ordered sequence of instructions within a thread).

2.3.2 Objectives

To address the issues as outlined above, we propose a novel hybrid dataflow/von Neumann architecture called the *Super-Actor Machine* (SAM). Since this is a novel architecture, our primary objectives are:

- to define a Super-Actor execution model and an abstract machine model for the SAM,
- to outline a proposed implementation of the SAM, in particular, the novel high-speed memory organization called a *register-cache*, and
- to examine how machine code can be generated for the SAM.

Once those objectives have been fulfilled, our next one is to examine the performance of one node of the Super-Actor Machine for a certain class of programs, that is, to investigate whether the node architecture of the Super-Actor Machine can tolerate local memory latencies and fine-grain synchronization costs. The last objective is explicitly focused on the performance of a single PE of the SAM because we believe that:

- the efficient utilization of one PE in the SAM is necessary if the multiprocessor system is to be used efficiently, and
- the speedup of a hybrid dataflow/von Neumann architecture in a multiprocessing context should be similar to those obtained by dataflow architectures since they also have support for tolerating global memory latencies and synchronization costs.⁸

A detailed simulator is constructed for the purpose of carrying out the last objective.

⁸This is not to say that the SAM's multiprocessing abilities should not be investigated, except that it is beyond the scope of this thesis. Monti's work[82] on interprocessor communications is one step in that direction.

Chapter 3

The Argument-Fetching Dataflow Model

In this chapter, we outline the argument-fetching dataflow model which is the starting point for our research on the Super-Actor Machine, and describe one implementation of an argument-fetching architecture called the McGill Dataflow Architecture (section 3.2). In section 3.3, we compare via a quantitative analysis the argument-fetching paradigm to the typical argument-flow model on which a majority of previously proposed dataflow machines are based. Lastly, we conclude this chapter by describing a variant of the McGill Dataflow Architecture which supports the dynamic dataflow model of computation, and a study indicating that the architecture can be overwhelmed by fine-grain synchronization requirements of a computation.

3.1 The Argument-Fetching Principle

An actor in the dataflow model has two roles: one is the processing of operands and the production of a result when all the inputs have arrived, and the other is the signaling of other actors that the result is available. Many proposed dataflow machines are based on the *argument-flow principle* where data generated by an actor must flow in a token to an assigned location of the successor actor which requires it as an operand. For example,

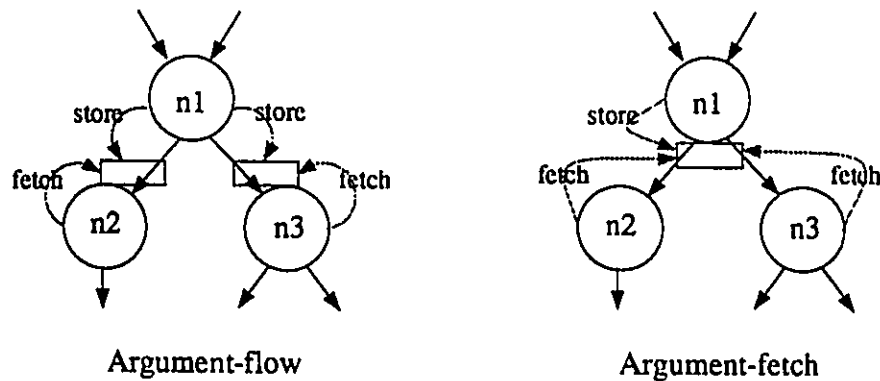


Figure 3.1: The argument-fetching and argument-flow principles.

the location for an actor's operand is within an *instruction template* which contains the actor's instruction, pointers to actors which require its result, and space for a guard value which indicates when the actor has received all its inputs. By having tokens containing the data and the tokens themselves acting as signals to the receiving actors, the roles of a dataflow actor can be neatly accomplished. However, the direct implementation of this argument-flow principle leads to two inefficiencies, namely

- excess data movement and unnecessary data replication, and/or
- the imposed tight-coupling between the mechanism for instruction execution and that for scheduling.

The paper by Dennis and Gao [32] proposes an architecture based on the *argument-fetching principle* where data generated as a result of an actor execution is directly stored in data memory, and when a subsequent actor requires that data, it directly fetches it from data memory just as in conventional processor architectures. The diagram in figure 3.1 best illustrates the two concepts. From the diagram, one can immediately see that excess operand/result movement and replication are eliminated in the argument-fetching case since it only has to store its result once even if it is required by multiple actors. Another benefit is that since data which the instructions execute upon are logically stored together,

so too can the scheduling information—the pointers to successor actors and the guard (semaphore) value—be logically and physically stored together. This separation of data and scheduling information has now freed the associated architectures from tightly-coupling their execution and scheduling logic. It may not be obvious why this is a benefit, but if one considers the case where an actor requires two or more inputs before it can be executed, then a tight-coupling of execution and scheduling logic would force the introduction of a ‘bubble’—a no-op instruction—in the execution mechanism whenever an n -ary actor ($n > 1$) receives an operand but is not ready for execution. This tight-coupling has forced the Monsoon machine [89] (an implementation of a tagged-token dataflow architecture) to incur a significant amount of wasted no-op cycles, e.g., 29% for a simulated annealing approach to the traveling salesman problem [25]. An architecture with a loosely-coupled execution and scheduling mechanism would have the opportunity of processing a ready actor instead of the no-op.¹

3.1.1 Argument-Fetching in a Multiprocessor Context

An inherent property of the argument-flow principle is that it is tailored for multiprocessing since the result of an actor is packaged into a token along with the destination information and sent to the successor actor regardless if its local or not. If the argument-fetching principle is used for data passing between actors residing on different PEs, then the number of interprocessor communication messages would be double that for the argument-flow case. Let us analyze this situation. In the argument-fetching case, the actor ($n1$) generating the result must first store it in local memory and send the first message telling the successor actor ($n2$) that it generated a value (fig. 3.2). Then $n2$ must send the second message asking for the value; some mechanism will fetch the value and package it in a third message destined for (PE y). After the value has been consumed, the successor actor sends an acknowledgement signal (the fourth message) to the actor which generated the value so that it can proceed to generate the next result.² As for the argument-flow case, it would

¹Loosely-coupled units imply that buffering mechanisms exists between the units such that the throughput of a unit can be relatively independent of another.

²We are assuming that both the argument-fetching and argument-flow cases are based on the static dataflow model of computation.

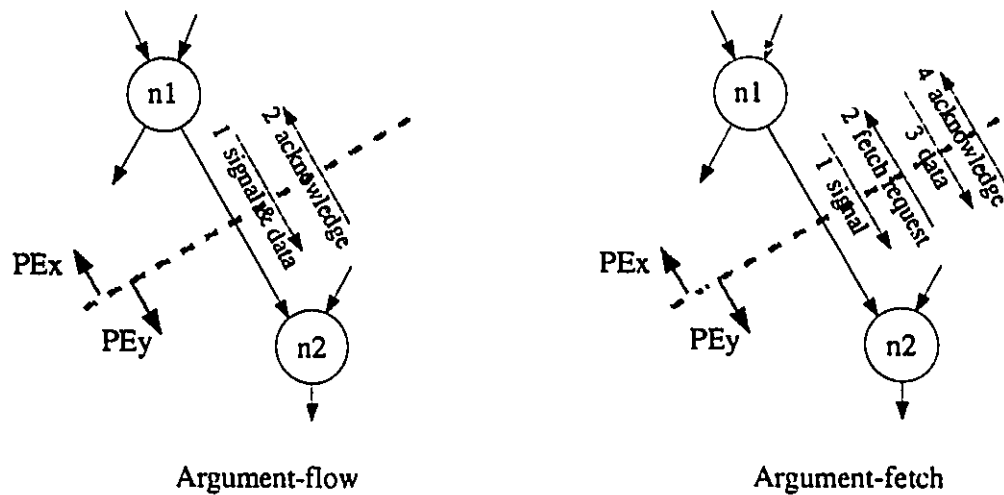


Figure 3.2: Interprocessor communications for the argument-fetching and -flow principles.

require only two interprocessor messages, the first containing the result (and signal) to the successor and the second, the acknowledgement signal from the successor.

Interprocessor communications are costly (in terms of hardware resources, especially when high bandwidths are required) and the message traversal times are generally long and variable. Excessive interprocessor messages tend to increase total execution times (due to network contention, packaging of the messages, etc.) and shift the bottleneck to the communication network. Thus in our work [44, 82], we advocate the argument-flow principle for data-passing between actors on different PEs and the argument-fetching principle for actors residing on the same PE. The determination of which arcs can possibly and do cross processing element boundaries can be done at compile time. At run time, an actor which notices that one or more of its emanating arcs are tagged for argument-flow, will activate some mechanism which fetches the data from local memory, packages it, and sends it to a location where the successor actor resides.

3.1.2 Previous Argument-Fetching Dataflow Work

The argument-fetching principle is not a new idea, it was first proposed in Rumbaugh's dissertation [95], and later, researchers from France and England also brought forward this concept [94, 116]. (This principle is currently the basis of three other proposed multi-threaded machines, one by Dennis [30], another by Dai and Giloi [26] and one by Evripidou and Gaudiot[37]. These architectures will be reviewed in chapter 9.)

In Rumbaugh's dissertation, he described an architecture called the *Intermediate Dataflow Machine* in which the notion of storing result data into memory instead of circulating it around the machine as data tokens was introduced. To the author's knowledge, he was the first to document the idea. He mentioned the use of *token memory* to store the data produced by an activated actor and successor actors requiring that result value will access it via an indexed pointer in the corresponding 3-address instruction. Signals produced by a recently fired actor are sent to an updating unit where respective count values are decremented. When a count value has reached zero, the corresponding address of the actor's instruction is sent to the execution unit. The information stored in the updating unit corresponds to a *control graph* which contains the same partial ordering information as found in the original dataflow graph. This architectural description is quite similar to the McGill Dataflow Architecture Model which we describe in the next section.

What was elucidated in the Dennis and Gao paper [32] was that an architecture can be built on the principle of a *clear separation* of the execution and signaling & scheduling component. The signaling component can be thought of as a simple function unit which can be pipelined to attain maximum throughput and the execution unit can be a conventional processing unit minus the program counter. This concept provides us the opportunity to incorporate ideas from more than forty years of research in von Neumann architectures into a dataflow machine.

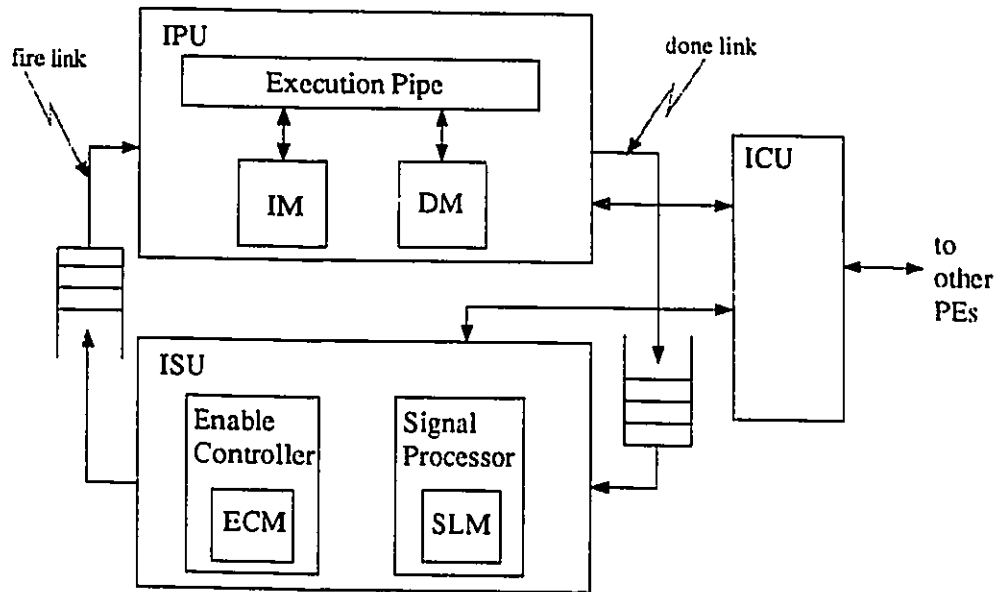


Figure 3.3: The McGill Dataflow Architecture Model.

3.2 The McGill Dataflow Architecture

The McGill Dataflow Architecture (MDFA) [51, 50] was conceived with the argument-fetching principle in mind, thus it has an instruction synchronizing and scheduling mechanism which is completely separate from the critical instruction processing data path (fig. 3.3). The latter, which is called the *instruction processing unit*, or IPU, consists of essentially similar hardware stages for pipelined instruction execution as found in a conventional pipelined processor, e.g., instruction fetch/decode, effective address calculation and operand fetch, execution (or the arithmetic and logic unit, ALU), and result store. The major difference is that the IPU does not contain a program counter to control the instruction sequencing, hence, we chose not to use the term central processing unit for the IPU. A separate unit called the *instruction scheduling unit*, or ISU, plays the role of determining which instructions are to be executed by the IPU. Furthermore, the IPU presents some advantages over a traditional execution pipeline:

- the dataflow model of computation guarantees that no pair of simultaneously enabled instructions can be in conflict over data, thus simplifying the pipe design (pipe interlocking mechanisms are unnecessary);
- pipeline gaps caused by conditional branches cannot arise since those facilities are implemented 'off-line' by the ISU; and
- the arithmetic and logic unit (ALU) stage can consist of multiple sub-pipelines of varying lengths for handling different types of instructions (e.g., floating-point and fixed-point adds, etc.) since the order of instruction completion in the execution pipeline does not matter.

In the figure, the IPU and ISU are linked in a circular path—the typical circular pipeline of dataflow processors. However, the fire and done links are buffered with first-in-first-out (FIFO) queues so that the throughput rates of the two units can vary and be relatively independent during program execution. The *fire link* delivers enabled instruction addresses (a *fire signal*) to the IPU. When an instruction has been completed, its address is sent back with a condition code (together they form the *done signal*) to the ISU via the *done link*. When an actor is being executed in the IPU, its instruction is fetched from the Instruction Memory (IM) and the operand fetch stage is used to retrieve its operands from Data Memory (DM). After the execution stage, the result is stored directly into DM in the result store stage and finally, a done signal is emitted.

The ISU consists of a Signal Processor and an Enable Controller. The signal graph (a graph representing the partial ordering of actors in a dataflow graph) of a program is represented in the ISU by the signal lists stored in the Signal List Memory (SLM) of the signal processor. Each signal list represents a set of signal arcs emanating from the associated node of the signal graph. The Enable Count Memory (ECM) of the enable controller holds *enable count* (the guard value) and *reset count* values for each node in the signal graph. In response to a done signal from the IPU for instruction n , the signal processor retrieves the signal lists for n and sends a *count signal* for each entry in the active lists—the set of active lists is determined by the condition code returned with the done signal. The signal processor interprets the condition code using these simple rules:

- the addresses in the unconditional signal list are always signaled;

- if the condition code is true (false), the addresses in the true (false) list are also signaled.

When the enable controller receives a *count* signal, it decrements the count value of the indicated node. If this enable count value reaches zero, an “enable” flag for this instruction is set and the reset-count value is copied back into the enable count value to prepare it for the next firing cycle of the instruction. Enable flags are continuously monitored by the enable controller; for each enabled node, a fire signal is sent and its enable flag reset.

The Interprocessor Communications Unit (ICU) is used to send packets across the network. When the signal processor notices that a particular count signal is destined for an actor residing on a remote PE, the signal is routed to the ICU instead. In the ICU, corresponding information indicates where the data should be fetched from local data memory and the data is packaged with the destination address and sent out on the network. At the remote PE, the data is stored locally and the receiving actor is notified. When an acknowledgement packet is received by the ICU (indicating that the successor actor has consumed the value), the ICU notifies the actor specified in the packet. A more detailed description of the interprocessor communications process can be found in Monti’s thesis [82].

3.2.1 The Program Format for the MDFA

A dataflow program graph G for the McGill Dataflow Architecture is represented by a *program tuple*:

$$G ::= \langle P, S \rangle$$

where P is a set of IPU instructions and S is a *signal flow graph* represented by a set of signaling instructions. Each actor in the dataflow program graph has an entry in both P and S sections of the program tuple. The instructions in P (p-instructions) contain no information pertaining to the sequence of execution. Instead, the sequencing information appears separately in the signal flow graph S .

Instruction Format for the IPU

The instruction graph P is a list of instructions where:

$$\begin{aligned}
 P &::= \langle \text{p-inst-list} \rangle \\
 \langle \text{p-inst-list} \rangle &::= \langle \text{p-instruction} \rangle \\
 &\quad | \langle \text{p-instruction} \rangle \langle \text{p-inst-list} \rangle \\
 \langle \text{p-instruction} \rangle &::= \\
 &\quad \langle \text{opcode} \rangle \langle \text{op-address} \rangle \langle \text{op-address} \rangle \langle \text{result-address} \rangle \\
 \langle \text{op-address} \rangle, \langle \text{result-address} \rangle &::= \langle \text{mode} \rangle \langle \text{address} \rangle
 \end{aligned}$$

(In the above modified BNF notation, the expression $\langle x \rangle, \langle y \rangle ::= z$ implies that the tokens x and y both have the form z .) Each p-instruction is a three address instruction commonly used in conventional architectures. These instructions are stored in the instruction memory and are executed by the IPU. The mode in the address field can either indicate that a constant is in the address field or that an actual address is there.

Signal Graph Format for the ISU

The signal graph S of the program tuple determines the sequencing of the instructions. Formally, the graph consists of a list of signal nodes:

$$\begin{aligned}
 S &::= \langle \text{s-node-list} \rangle \\
 \langle \text{s-node-list} \rangle &::= \langle \text{s-node} \rangle \\
 &\quad | \langle \text{s-node} \rangle \langle \text{s-node-list} \rangle \\
 \langle \text{s-node} \rangle &::= \langle \text{signal-count} \rangle \langle \text{signal-list} \rangle \\
 \langle \text{signal-count} \rangle &::= \langle \text{reset-count} \rangle \langle \text{enable-count} \rangle
 \end{aligned}$$

$$\langle \text{signal-list} \rangle ::= \langle \text{u-list} \rangle \langle \text{t-list} \rangle \langle \text{f-list} \rangle$$

$$\langle \text{u-list} \rangle, \langle \text{t-list} \rangle, \langle \text{f-list} \rangle ::= \langle \text{s-list} \rangle$$

$$\langle \text{s-list} \rangle ::= [\langle \text{address} \rangle]^0$$

(Expressions of the form $[\langle xx \rangle]^0$ imply zero or more tokens of type xx .) Each signal node, or s-instruction, contains three address lists designated the unconditional, true and false signaling lists. The true and false lists are used in the implementation of conditional expressions. An element of a signal list is an address of an actor. The signal count consists of both the enable count and reset count fields to facilitate the dataflow firing rules.

3.2.2 Example Program Tuples for the MDFA

The program tuple representing the expression:

$$z := (x + y) * (x - y + 3)$$

is shown in figure 3.4. (From now on, we will express example programs as SISAL expressions [80].) The p-code represents the unordered set of 3-address instructions corresponding to the nodes in the s-code. The number in the upper left-hand corner of each s-node is the initial enable count value and the one in the lower left-hand corner is the reset count value. The backward signal arc, shown as a gray directed arc (e.g., the directed arc from $n4$ to $n3$), is an acknowledgement arc. The reception of all acknowledgement signals by a node indicates that it may overwrite its result location since all its successor(s) have consumed it.

A conditional expression like:

$$z := \text{if } i < 3 \text{ then } x + y \text{ else } x - y \text{ endif}$$

can be converted to the program tuple shown in figure 3.5. The arcs emanating from the U ,

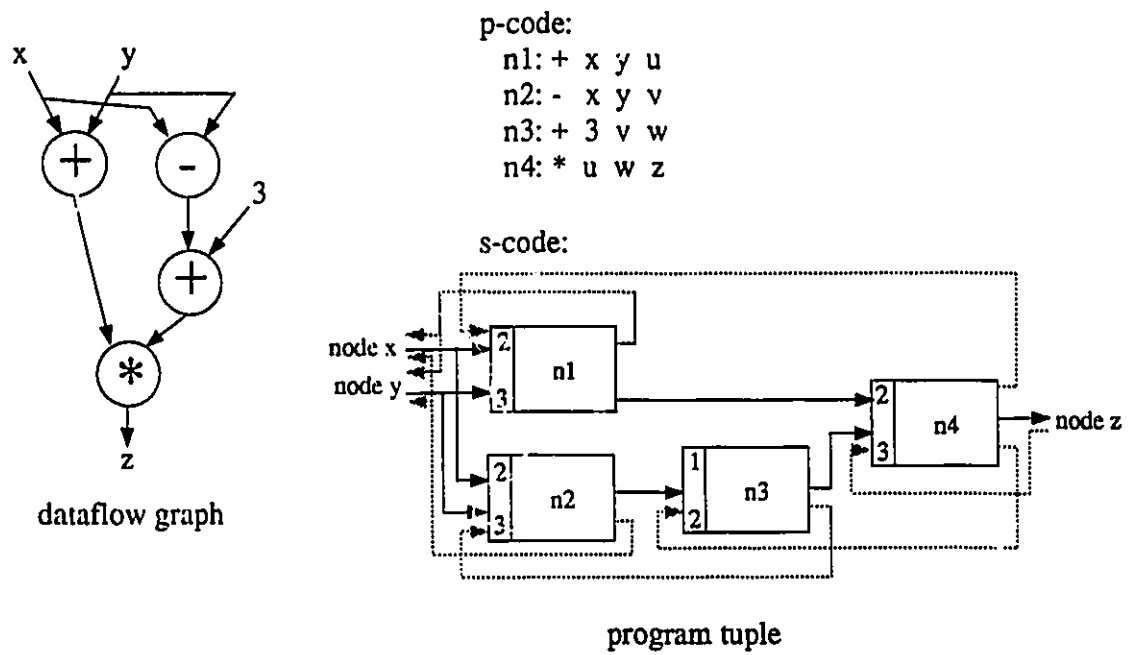
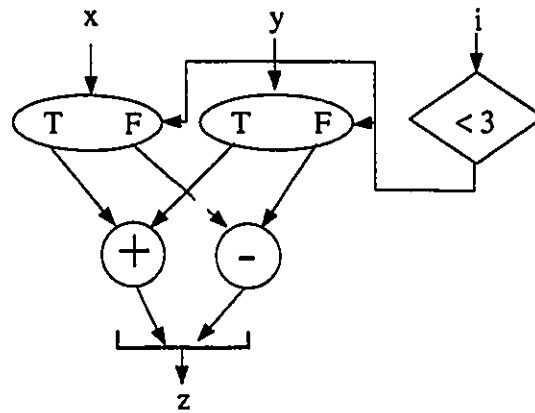
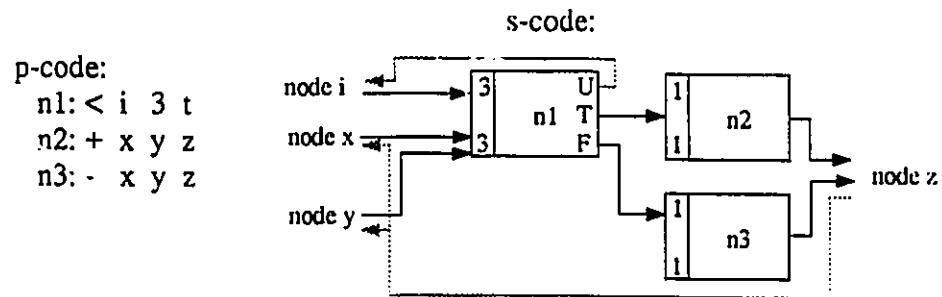


Figure 3.4: A program tuple.



dataflow graph



program tuple

Figure 3.5: A program tuple representing a conditional expression.

T , and F , of $n1$ in the s-code correspond to the elements found in the unconditional, true and false signal lists of $n1$. Unlabeled arcs emanating from other s-nodes are unconditional arcs. Note that the true and false gates used in conditional expressions of a dataflow graph are not required.

With the basic conditional expressions, loop constructs can be easily represented. For example, the loop:

```

z := for i in 1, N
      j := old j + i
      return j
    endfor

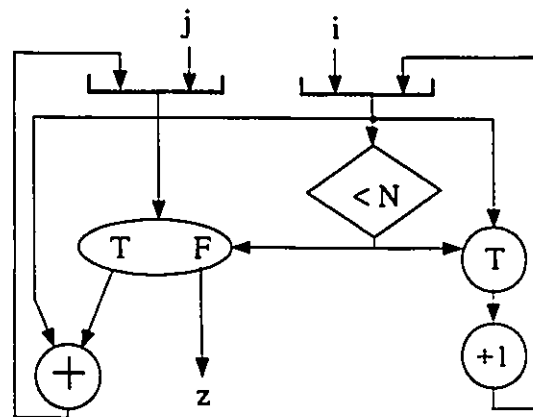
```

can be represented by the program tuple in figure 3.6.

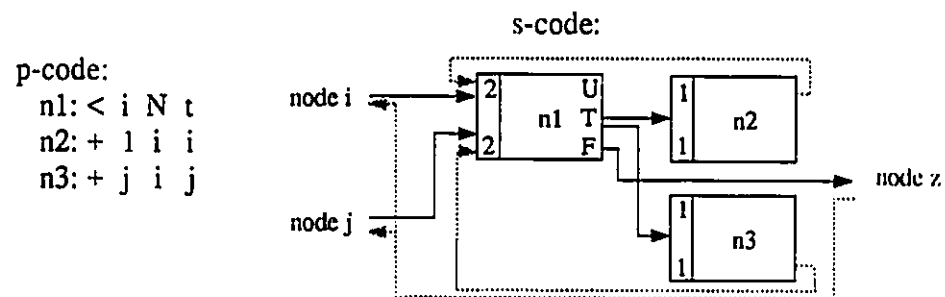
From the above examples, we have shown that converting a dataflow graph representation to its corresponding program tuple is a straightforward one-to-one transformation. It is clear from the examples that if the original dataflow graph is *well-behaved*—a set of inputs produces a unique set of outputs, the graph returns to its initial state after an execution, and the graph is conflict-free (i.e., the merge nodes do not receive two inputs simultaneously on its input arcs)—then the program tuple is well-behaved and thus retains the determinacy property of the dataflow model.

3.3 Why the MDFA?

The advantages of the argument-fetching machine over the argument-flow ones are most evident in a single processing element, and thus, the comparisons will be focused on one PE. Multiprocessing issues of argument-fetching versus argument-flow have already been discussed in section 3.1.1 where no advantage is claimed for argument-fetching. In the following sections, we will first analyze the excess data value movement in the argument-flow machines, then we will compare the memory bandwidth requirements in the MDFA



dataflow graph



program tuple

Figure 3.6: A program tuple representing a loop construct.

to a straightforward implementation of a static argument-flow machine. Lastly, we will compare the savings in actor firings arising from the elimination of true- and false-gates in conditional constructs.

3.3.1 Data Value Movement Analysis

The most obvious advantage the MDFA has over the static argument-flow architectures is the decreased amount of data value movement—traffic that is accounted to the passing of data values between actors. As we have already mentioned, a result value is stored only once in the argument-fetch model, whereas for the argument-flow one, it can be stored many times depending on the fan-out of the actor which produced it. Now let us analyze quantitatively the savings in data value movement in the argument-fetching architecture for one processing element. The data traffic (implicitly, the memory bandwidth requirements) for data value movement will be calculated for a typical actor.

For this analysis, let f_o represent the average fan-out of a typical actor, and let f_i be the average fan-in. The sizes of the operands, results, and memory addresses are set to x bytes.³ Assume that the static dataflow graph is executed once and each actor is to be fired only once. This implies that actors are not required to signal their predecessor actors that they have executed. Since we are only examining the case for one PE, let us assume that the entire dataflow graph is to be executed on one processing element.

For the static argument-flow architecture, each “firing” of an instruction must read its operand slot(s) once through the Fetch unit and store its result into the operand slot(s) of the successor actor(s) through the Update unit (see section 1.2.2). So for each actor, fetching an operand takes $2x$ bytes, x bytes for sending the address to memory, and x bytes for the return of the data value itself. For a typical actor, the data traffic to and from the activity store for operand fetching would be:

$$DT_{of} = f_i 2x \quad (3.1)$$

³In today's 32-bit microprocessors, the size of a memory address is typically equal to the size of an operand or result.

To store the result, we also require $2x$ bytes, thus a typical actor's data traffic for storing the result is:

$$DT_{rs} = f_o 2x \quad (3.2)$$

The total data traffic attributed to data value movement for a typical actor is:

$$\begin{aligned} DT_{tot} &= DT_{of} + DT_{rs} \\ &= (f_i + f_o)2x \end{aligned} \quad (3.3)$$

In the McGill Dataflow Architecture, a typical actor requires $f_i 2x$ bytes for operand fetching, but since we only store the result once, the storing only generates $2x$ bytes of traffic, for a total of:

$$DT_{tot} = (f_i + 1)2x \quad (3.4)$$

In general, $f_o > 1$ (how else will the parallelism be generated?), so let us assume that $f_i = f_o = 1.7$ and that $x = 4$ bytes (assuming that we are working with a 32-bit machine).⁴ Then an actor's data traffic for the argument-flow machine is ≈ 27 bytes whereas the MDFA generates ≈ 22 . This $27/22 \approx 1.23$ or 23% savings in data traffic reduces the memory bandwidth requirements accordingly, so at execution time, data memory will be accessed less frequently. This decreases the chances of memory contention and thus the execution pipe may stall less often. The net result should be a decrease in execution time⁵, assuming that the scheduling mechanism in the MDFA is as good as the one in the argument-flow machine. Next, we will see how the scheduling mechanism in a naive implementation of an argument-flow machine can perform worse than the one in the MDFA.

⁴In Ghosal and Bhuyan's performance analysis [55] of the Manchester Dataflow Machine, they assumed an average fan-out factor of 1.4 where only monadic and dyadic actors are considered and an actor's fan-out is restricted to 2. If the fan-out of an actor is not restricted, then the average fan-in fan-out should be somewhat greater, thus we estimate the value to be 1.7.

⁵We have taken the liberty of assuming that the execution pipe is a conventional multi-stage pipeline without any buffering mechanism to and from data memory.

3.3.2 The Cost of Bundling Data and Signal Processing Information

In the MDFA, the separation of the instruction processing mechanism and the instruction scheduling mechanism is clearly encouraged along with the use of the Harvard architecture type of memory structure, i.e., the separation of instruction and data into separate memory units. In many argument-flow dataflow architectures, this separation of memory is not as obvious, thus a straightforward implementation of a static argument-flow architecture—let us call this the *naive argument-flow architecture* (NAFA)—may bundle the signal and data processing information for a particular actor into a contiguous space of memory.⁶ That is, the NAFA requires that the operator, operands, destination addresses and enable count value of an actor be stored contiguously such that only one address is required to fetch the information for an enabled actor. This grouping of data and signaling information into the activity store causes memory contentions between the instruction scheduling (the update unit) and the memory interfacing stage of the data processing mechanism (the fetch unit).

Let us analyze the memory bandwidth requirements of the NAFA's activity store for a typical actor. As we have shown in eqn. 3.3, operand fetching and result storing produces $(f_i + f_o)2x$ bytes of traffic. Let us assume that the operator is $x/2$ bytes and the enable count is $x/8$ byte.⁷ When an actor is enabled, the fetch unit sends one address (x bytes) to get the operator, operands and f_o destination addresses ($f_o x$ bytes). After the actor has been executed, the update unit must store the result and retrieve the enable count f_o times on average ($f_o x/8$ bytes). After decrementing the enable count value, it must be stored; thus, the actor will produce $f_o(x + x/8)$ bytes of data traffic (address plus enable count). In total, an actor generates:

$$\begin{aligned}
 DT_{\text{Total}} &= DT_{\text{tot}} + DT_{\text{sig}} \\
 &= (f_i + f_o)2x + \\
 &\quad (x + (f_o x) + (f_o x/8) + (f_o(x + x/8))) \\
 &= f_i 2x + f_o 4.25x + x
 \end{aligned} \tag{3.5}$$

⁶The actual implementations of the argument-flow architectures may not bundle the signal and data processing information together. However, since we are not aware how much information separation actually takes place, then we will just analyze the base case where there is no information partitioning.

⁷If operands are four bytes long, then the enable count would be .5 bytes or four bits.

bytes of data traffic to and from the activity store.

As for the MDFA, all memory references by the ISU are addressed to the SLM and ECM. (SLM, ECM, IM and DM are physically separated from each other.) The data traffic to and from the SLM by an actor is $3x + f_o x$ bytes; $2x$ bytes for retrieving the signal list offsets (x bytes for the address and x bytes for the offset) plus $x + f_o x$ to retrieve the addresses in the list. As for the ECM, the data traffic there is $f_o 2.25x$ bytes; $x + x/8$ bytes to fetch and the same amount to store. In the IM, the memory traffic is $2.5x + f_i x$ bytes; x bytes for the address, $x/2$ for the operator, $f_i x$ the operand addresses, and x bytes for the result address. As analyzed in eqn. 3.4, the data traffic generated by an actor to and from data memory is $f_i 2x + 2x$.

Assuming the same values for f_i , f_o , and x as in the previous analysis, data memory has the greatest bandwidth requirement at ≈ 22 bytes per actor. In the NAFA, the activity store must support ≈ 47 bytes of traffic per actor. As we can see, the bandwidth required from the single memory subsystem in the NAFA is more than double that of the most used memory subsystem in the MDFA.⁸ This implies that the McGill Dataflow Architecture can execute faster due to fewer memory contentions. Furthermore, the scheduling mechanism in the NAFA may stall more often because its memory accesses must contend with the accesses from the data processing mechanism. The ISU in the MDFA does not have that problem.

The argument-fetching principle allows one to conceptually separate different types of distinct information, and thus at the architectural level, it was natural to physically store each type in their own memory subsystems for an overall increase in memory bandwidth. Traditional static dataflow architectures could also have utilized different memory units to store the operators, operands, etc., but with the argument-flow principle, it was not as obvious. This development of the argument-fetching principle in dataflow computing is similar to the development of the Harvard-type architecture in the von Neumann camp—separating instructions and data into different memory subsystems—the trade-off being

⁸Though the *total* data traffic in the MDFA is greater than that of the NAFA, a lot of that traffic is in the addresses required for fetching data. However, those addresses are directed to each separate memory subsystem without fear of contention.

increased hardware requirements for faster execution. Today, all fast conventional von Neumann processors utilize the Harvard-type architecture [62] in one form or another.

3.3.3 Cost Analysis of Conditional Expressions

A less obvious advantage of the MDFA is in its handling of conditional expressions. Figure 3.5 shows a conditional expression expressed in a dataflow graph and corresponding program tuple. The true- and false-gates are not required for the signal graph, so for every conditional expression, a minimum of two actors are not required. The savings in memory usage is not the main advantage; the savings is at execution. Since every conditional expression executed must traverse through its true or false arm, at least one extra actor (e.g., multiple values sent into one arm will require multiple gate actors) is processed each time. Studies have shown that during a run of a typical program, conditionals are executed 11 to 51% of the time [78]. Therefore, the MDFA would execute 11 to 51% less actors than other dataflow architectures which require true- and false-gates in conditional constructs.

3.3.4 Summary

We have quantitatively analyzed the advantages of the McGill Dataflow Architecture and we have shown that:

- on average, there is substantially less memory bandwidth requirement for data value movement in a single processing element as compared to a static dataflow architecture which implements the argument-flow principle. This results in less contentions for memory cycles and thus less chance of stalling the execution pipe. Hopefully, this results in a faster execution time.
- In a naively implemented static dataflow architecture, where data, signal, and actor information are grouped contiguously, increased memory contentions may arise between the data processing and signal processing parts of the processing element due

to increased memory bandwidth requirements. It has been analyzed, that the memory bandwidth requirements on the activity store in the naive implementation is nearly double that of the most heavily accessed memory unit—the data memory—in the MDFA.

- At the dataflow graph level, the MDFA will execute 11 to 51% less actors in a typical program as compared to a dataflow architecture which requires the use of true- and false-gates in conditional expressions.

3.4 Summary

In this chapter, we have reviewed the argument-fetching principle and the McGill Dataflow Architecture which supports it. It was shown that the argument-fetching principle is beneficial within a processing element of a multiprocessor system, and that passing data between actors on different processing elements should adhere to the argument-flow principle.

The architecture described in this chapter implements the static dataflow model of computation. A variant of the MDFA which can support concurrent and recursive function invocations at run time has also been proposed and investigated in the course of our research [47, 71]. The argument-fetching architecture which supports the dynamic dataflow model of computing is called the *McGill Dynamic Dataflow Architecture* (MDDFA), the precursor of the Super-Actor Machine. The basic notion behind a dynamic argument-fetching architecture is the frame of contiguous memory locations—called a *function overlay*—associated with each function invocation. An overlay, akin to a *stack frame* as used in conventional von Neumann architectures, contains function linkage information and *overlay slots* for containing results and enable count values of actor instances. By associating a *base address* of an overlay with each fire and done signal, instances of actors belonging to different function invocations can access their associated overlays. And by introducing an ‘apply’ and a ‘return’ actor, functions can be invoked and values returned to the caller function dynamically. Simulations of a possible configuration of the MDDFA were performed where function-call intensive benchmark programs were used to examine

its effectiveness in supporting function applications. Interested readers are referred to [71, 70].

However, all was not fine with the MDFA. In a study of the efficiency issues of the McGill Dataflow Architecture [48], we found that the synchronization overhead for scheduling at the individual instruction level can easily overwhelm the instruction scheduling unit. The experiments examined dataflow software pipelined loop bodies (see appendix A) on a simulator of the MDFA. We discovered that the signal processing capacity (C) of the scheduling unit(s) must be greater than or equal to the average signal density (S) of the computation multiplied by the processing capacity (P) of the instruction processing unit(s). C represents the number of count signals the ISU can process per cycle and P represents the number of instructions the IPU(s) can process per cycle. The average signal density is computed by the total count signals emitted during the computation divided by the total instructions executed. In general, S is greater than one (dyadic instructions already require two signals to be enabled) and this implies that the throughput of the scheduling unit must be greater than the throughput of the processing unit in order that the IPU(s) can be kept busy. That is, if the IPU can accept a fire signal every cycle, then the ISU must be able to process multiple count signals per cycle to keep up. Another way of keeping the IPUs busy is to generate code which reduces unnecessary signaling by grouping instructions into *threads* or “macro dataflow nodes”. Instructions inside a thread can be scheduled with a simple counter, while the activations of thread instances are explicitly synchronized, i.e., processed by the ISU. This requires the extension of the basic MDFA to support hybrid execution and leads us to the focus of this dissertation: *what evolutionary steps should be taken to modify the MDFA such that it can efficiently support, at the architectural level, the notion of an aggregation of dataflow actors (threads)? How do we supply a low and fixed memory access time to the ISU which was assumed in the experiments we performed in our study?* To answer these questions, we propose a novel architecture called the Super-Actor Machine.

Chapter 4

A New Architecture

In the next four chapters, a novel pipelined multi-threaded architecture called the *Super-Actor Machine*[65, 67, 66, 68] is described and analyzed. The Super-Actor Machine (SAM) is to be a multiprocessor system consisting of processing elements linked together by some interconnection network. The SAM implements the argument-fetching principle and supports dynamic function application. A processing element of the SAM consists of heterogeneous processing units responsible for processing short and fixed latency instructions, processing long latency instructions, scheduling aggregates of instructions, etc.

The primary application domain of the SAM is scientific numerical computations where sustainable and efficient floating point performance is crucial. An important feature of the Super-Actor Machine which allows it to attain high floating point performance is its ability to simultaneously issue and overlap floating-point arithmetic operations with many other operations in one processing element. This ability is coupled with a highly pipelined execution unit which allows it to have a fast cycle time. Moreover, a processing element also has the capability of issuing multiple instructions from multiple streams of instructions. Machines with these features represent a new direction in architecture research where the two fundamental problems of multiprocessing can be addressed in a processing element which has the capability of multiple instruction issue and fully pipelined function units—features which are heralded for the next generation of uniprocessor RISC machines

[61, 108].

4.1 Addressing the Locality Issue

For this multi-threaded machine and others in its class, the ability to exploit the principle of locality (temporal and spatial) is challenging and is necessary to tolerate local memory latencies. We have looked towards modern von Neumann machines for ways to tolerate local memory latencies and to provide sufficient local memory bandwidths in multi-threaded architectures. . . unfortunately, no satisfactory answers were found.

In modern RISC architectures, the reduction in local memory latencies is achieved by the extensive use of (explicit) programmable registers and (implicit, i.e., generally not programmable) high-speed caches. A number of programmable registers alone can only provide a partial solution because:

- increasing programmable registers will increase the context of a thread, resulting in an increased overhead for context-switching; and
- the task of allocating registers for multiply active threads is complicated by the non-deterministic arrival times of thread triggering events.

As for the conventional cache solutions, there are many limitations which have been studied and reported for modern von Neumann machines. Below, we list the important limitations:

- the published high hit ratios have been reported mostly on non-scientific benchmark programs. For scientific applications where large arrays (vectors) of data are accessed and manipulated in the computation, the cache performance can be less than adequate [19, 107]. The major reason is that typical caches cannot contain the large arrays so that the throughput of the execution pipe is effectively limited by the time for loading a line into cache.
- When a cache miss occurs, the instruction pipeline usually stalls or freezes, causing considerable performance degradation [62]. This degradation will become more severe as the mismatch in processor speed and memory access times continues to grow—as witnessed in the new generation of processors.

- The fact that most conventional caches are transparent to the programmers (compilers) makes performance improvements by optimizing compilers difficult.
- Lastly, conventional cache memory was not designed with multi-threaded architectures in mind. Frequent switchings between instruction threads have a negative impact on the locality of reference. Moreover, multiple active contexts contend for limited cache space which further erodes the benefits of the cache because of unwanted cache line replacements. Some notable examples of multi-threaded architectures have rejected the use of caches, e.g., the Tera Computer[7] and the HEP[105].

4.1.1 A New Execution Model and Novel Memory Organization

To address the issue of tolerating *local memory latencies*, the Super-Actor Machine supports a new model of execution called the *Super-Actor Execution Model* and incorporates a novel high-speed memory organization known as the *register-cache*. As the word “register-cache” suggests, it is organized both as a register file and a cache. Viewed from the execution unit, its contents are addressable similar to ordinary CPU registers using relatively short addresses. From the main memory perspective, it is content addressable, i.e., its contents are tagged just as in conventional caches. The basic idea of the register-cache organization is simple: a number of registers are grouped into a block where a register in a block is accessed using an offset from the address of the block; an offset value embedded in the compiler generated code. The binding of a register block to a register-cache line address is adaptively performed at run time, thus resulting in a dynamically allocated register file.

The Super-Actor Machine effectively utilizes this new memory organization by supporting the notion of non-preemptable instruction threads, called *super-actors*, at the instruction set architecture level. A super-actor becomes ready for execution only when:

1. the *data dependence is satisfied*, i.e., all its input data are logically generated; and
2. *space locality is satisfied*, that is, its input data are physically residing in the register-cache and space is reserved there to store its result.

The first condition is similar to the so-called *firing rule* in a traditional dataflow machine, however each scheduling quantum in the SAM is an instruction thread instead of one instruction. The second condition, a feature unique to the SAM architecture, ensures that an enabled super-actor can be scheduled for execution only when all memory accesses of its instructions are guaranteed to be in the high-speed buffer memory. In a processing element of the SAM, a processing unit is responsible for enforcing the second condition by ensuring that the necessary data is in the register-cache. In this manner, the load/store between fast memory and the slower main memory is a non-blocking off-line operation, i.e., other super-actors which already have their data in fast memory can proceed while the super-actor requiring a memory load to the register-cache is processed by another unit other than the execution unit. By supporting this new model of execution, the execution unit in a processing element will never freeze when accessing instructions or data, thus eliminating one main source of pipeline performance degradation. Furthermore, since the throughput of the execution unit is not dependent on a cache hit ratio, the execution time of applications are not at the mercy of the cache performance if there is enough exposed parallelism in the application.

4.2 Format of Textual Information in the Pseudo-Code

In the following chapters, pseudo-code will be used to illustrate algorithms, operations of function blocks, etc. We would now like to explain the font usage in the pseudo-code.

Type	Font	Example(s)
keywords	sans-serif-bold	function, while
function calls	sans-serif	label(<i>xyz</i>)
identifiers	italic	<i>abc, D[i+j]</i>
textual directives	roman	clear registers

Comments are enclosed in /* ... */ delimiters.

Chapter 5

The Abstract Model of the Super-Actor Machine

In our view, the abstract model of the Super-Actor Machine should serve three purposes:

- to be a behavioural specification of the Super-Actor Machine,
- to be a foundation for the development of future compiler optimization techniques which are specific to the SAM, and
- to be a foundation for future architectural developments and improvements of the SAM.

To fully define an abstract model of the Super-Actor Machine, we define an abstract program execution model and an abstract machine model. The program execution model provides a high-level picture to a compiler writer while the machine model is more for the hardware implementors.

The abstract program execution model is detailed in the next section and the abstract machine model is described in section 5.2. We conclude this chapter in section 5.3 by discussing other related execution models.

5.1 The Abstract Program Execution Model

The abstract program execution model is defined by giving the syntax of a Super-Actor Graph—an organization of a program's instructions which the abstract machine model of the SAM can execute—and its operational semantics. The transformation of a well-formed dataflow graph to a super-actor graph which retains the determinacy property of the dataflow model is outlined in chapter 7.

In the next section, we describe the super-actor graph which can be executed on the base abstract machine model. Syntactic extensions and semantic refinements to the base execution model are introduced as the base abstract machine model is enhanced. Section 5.1.2 illustrates the concept of super-actor graphs with examples, and section 5.1.3 examines how a super-actor graph can be determinate.

5.1.1 The Super-Actor Graph

The Super-Actor Machine executes a *Super-Actor Program*, P , which is a set of program graphs called *Super-Actor Graphs*, G .

$$P = \{G_1, G_2, \dots, G_n\}$$

G_1 is the super-actor graph which starts the execution of P and is called the *main* super-actor graph. A super-actor graph G_i represents a function definition in P and consists of two parts: a signal flow graph and an associated overlay map.¹ A *signal flow graph* consists of virtual nodes and super-actors which are linked by directed edges. A *super-actor* contains a list of one or more instructions and *virtual nodes* are similar to super-actors but with no list of instructions. They are virtual in the sense that they do not exist in the actual machine code which is executed by an implementation of the Super-Actor Machine. They are part of the SA graph syntax for explaining high-level constructs such as function definitions, if-then-else constructs, and loop constructs. The directed arcs in the signal flow graph

¹Indeed, the super-actor graph is not a 'graph' in the graph-theoretic sense. For the lack of better terminology, we will continue to call the signal flow graph and its associated overlay map a super-actor graph.

indicate the data and/or control dependencies and each one specifies a directed path for the traversal of *signals*, as in the argument-fetching dataflow model; they are not used for the traversal of tokens containing data.

An *overlay map* of a SA graph specifies the locations of super-actors' results in an overlay. This *overlay* is a set of locations in *overlay space* and consists of one or more overlay blocks. An *overlay block* is a group of one or more locations in an overlay and has an attached label for identification. The primary purpose of an overlay is to be a storage medium for conveying data between super-actors belonging to the same activation of a SA graph, i.e., for super-actors in the same function activation. (We will discuss how super-actors in different function activations can communicate with each other when we discuss function application instructions on page 58.) Results of each super-actor are assigned unique locations within an overlay, although multiple super-actors depositing their results to the same locations are possible (e.g., certain super-actors in a conditional construct must deposit their results in the same locations so that successor super-actors of the conditional construct can access them). Note that overlay space is not used for storing structure memory objects—vectors, arrays, etc.—instead, *structure memory space* is used for those purposes (these will be discussed when structure memory operations are introduced).

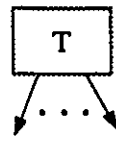
The super-actor graph can best be defined by first describing its components and then the compile-time entities called *encapsulators* which are used by the compiler to group super-actors and virtual nodes together to represent high-level constructs such as conditionals and loop expressions. Encapsulators are not part of the super-actor graph's syntax, but are solely used to explain the structuring of SA graphs.

Components of a Super-Actor Graph

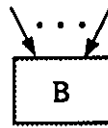
The notation and terminology for the three types of *virtual nodes*, the two types of super-actors, and the overlay map are shown in figure 5.1 and are explained below.

Overlay Map In an overlay map, blocks associated with the same SA graph have unique *block identifiers* (*block-ids* for short) such that instructions within a super-actor in the SA

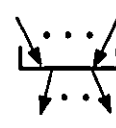
virtual nodes



top node

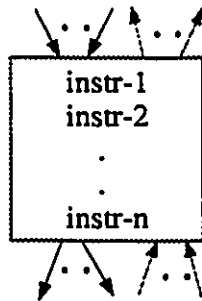


bottom node

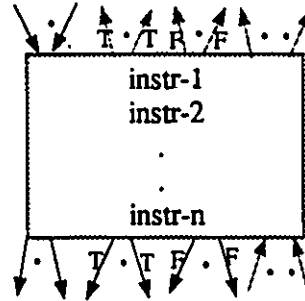


merge node

super-actor



switch super-actor



overlay map

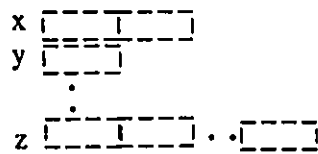


Figure 5.1: Components of a super-actor graph.

graph can identify which overlay block(s) it operates on and access the appropriate locations. However, blocks associated with two different SA graphs can have the same label (we will see how they can be differentiated when we discuss function activations). The first dashed box beside a block-id (e.g., x , y or z in figure 5.1) is termed the first location of the block, and successive locations are placed to the right of the preceding one. A location within an overlay is also called an *overlay slot*.

Super-Actors and Their Instructions A super-actor has one or more *incoming (input) edges* and one or more *outgoing (output) edges*—solid directed arcs in figure 5.1—which connect it to other super-actors or virtual nodes. It also has zero or more *incoming acknowledgement edges* and zero or more *outgoing acknowledgement edges*—hashed directed arcs—which are used to convey acknowledgement signals. Acknowledgement signals are generally used when a super-actor wants to acknowledge its predecessor(s) (ancestor(s)) that it has fired so that the predecessor (ancestor) can be re-activated once it has received another set of inputs. In our discussions, outgoing acknowledgement edges are classified as outgoing edges while incoming acknowledgement edges are separately identified from incoming (input) edges. Within a super-actor is a totally ordered list of n instructions where an instruction is of the form:

$$\text{optri } [op]^0[res]$$

The above notation indicates that there can be zero or more operand fields and an optional result field, depending on the operation as specified in **optri**. The **optri** field contains an operation like ‘add’, ‘abs’, etc., and an operand field contains either an immediate value, or an *overlay slot pointer*—value indicating a location within an overlay. The *res* (result) field contains an overlay slot pointer. In the illustration of SA graphs, we represent an immediate value with the format ‘# i ’; an overlay slot pointer is represented by ‘*block-id.offset*’. The *offset* value is an integer where zero indicates the first location in the overlay block identified by *block-id*, one, the next location, and so on. (In the following discussions, we will sometimes use the word ‘actors’ to imply super-actors.)

Other instructions within a super-actor include a 'nop' instruction, branching instructions, relational and logic instructions, structure memory operations, and function application operations. The instruction 'nop'—no operand nor result fields—represents “no operation”.² Branching instructions are used to branch to an instruction within the same super-actor. A branch instruction, 'br' has one field containing an immediate value which indicates the offset of the next instruction from the current one to branch to. A conditional branch instruction within a super-actor, e.g., 'brt' for “branch when true”, requires two address fields: the first one pointing to a location containing the condition code (generated by a previous instruction), and the second, an immediate value indicating an offset from the current instruction. Instructions which generate a condition code (a boolean value) are called *relational* or *logic* instructions, e.g., '<', 'and', etc.

Structure Memory Instructions Four instructions are provided for structure memory operations. Structure memory (SM) objects are stored in structure memory space and are used to represent vectors, matrices, etc. A SM object is identified by a *structure memory object identifier (SMO-id)* and each element within the object can be accessed via an offset value; an offset from the identifier. 'SMalloc' and 'SMdealloc' are used to allocate and deallocate structure memory space for a SM object. The SMalloc instruction takes one operand which specifies the size of the SM object, and returns a SMO-id which identifies that SM object. The SMdealloc instruction takes one operand which is a SMO-id, and deallocates that space, i.e., the space is freed up for allocation to another SM object.

'SMread' and 'SMwrite' are instructions used to access values within a SM object. The SMread instruction takes three or four arguments where the first one specifies the structure memory object identifier of the SM object and the second, the offset into the object locating the first element to be read. The third argument specifies the id of the overlay block in which the read value(s) are to be stored. The first read value is stored in the first overlay block location, the second, in the second block location, and so on. The optional fourth argument specifies the number of successive elements to be read; an absence defaults the number

²The 'nop' instruction is only used in the abstract model so that examples can be clearly illustrated. In an actual machine code, 'nop' instructions within a super-actor are not necessary.

of values to read to one. The SMwrite instruction also takes three or four arguments: the first one specifies the SMO-id of the SM object, the second, the offset into the SM object indicating the first destination location of the write, and the third, the block-id of the overlay block which contains the values to be copied. The optional fourth argument specifies the number of successive elements to be written from the overlay block to the SM object; again, an absence implies that there is only one value to be copied.

Function Application Instructions Function applications are supported by three instructions: 'Oalloc', 'Odealloc' and 'send' instructions. The 'Oalloc' instruction takes one operand which points to some information in data memory (the function application data are treated as if they were constants) representing some function definition, say f . Such information includes how many operand blocks are required by f 's SA graph, and the SA graph itself. The 'Oalloc' instruction allocates an overlay from overlay space for the newly invoked f function, creates the function instance—the function instance consists of actor instances (super-actors associated with the function instance)—and deposits an *overlay pointer* value into a location in the caller's overlay as specified by the result field value of the Oalloc instruction. This overlay pointer is basically the base address of the overlay and is attached to a signal—signals sent between super-actors—to uniquely identify the function activation the signal is associated with. It is also used by instructions within a super-actor to identify the overlay to access. The 'Odealloc' has the simple task of deallocating an overlay, that is, declaring that the overlay will not be accessed by actors of a function instance to be terminated. Basically, the space of the deallocated overlay is freed up for assignment to another function invocation.

The 'send' instruction is used to pass a value between the caller and callee functions and can be used to issue a signal to an appropriate super-actor when the value has been sent.³ The 'send' instruction takes three or four operands. The first operand specifies the value to be copied, the second operand specifies the overlay (the overlay pointer), and the third value specifies the overlay slot to receive the value. The fourth operand, if it is specified,

³If multiple super-actors have to be notified, then the one super-actor which the send instruction notifies can be made responsible for notifying the other super-actors.

contains an actor id. Combined with the overlay pointer as specified in the second operand, the fourth operand indicates which actor instance to send a signal to.

A super-actor which invokes a function or returns values for a function instance must have the function application instructions at the end of its list of instructions. Also, a super-actor can only invoke one function; this becomes obvious when we show an example of function applications on page 72. The syntax of the above instructions are clearly described in a later section when the instruction set of the abstract machine model is detailed.

Switch Super-Actor A *switch super-actor* has zero or more unlabeled outgoing edges—these are the same type of output edges as those found in a regular super-actor—and it has one or more true ('T') *labeled output edges* and one or more false ('F') *labeled output edges*. As for acknowledgement arcs, there are input acknowledgement arcs—the same as the ones in a regular super-actor—and labeled and unlabeled output acknowledgement edges. The zero or more unlabeled output acknowledgement arcs are the same type as the output acknowledgement arcs of a regular super-actor. As for labeled (true or false labeled) outgoing acknowledgement arcs, a switch super-actor can have zero or more of each. The instruction list of a switch super-actor is similar to a regular super-actor except that the last instruction in the ordered list is responsible for generating a condition code.

Super-Actor Instances

Since an actor (switch or regular) can be associated with multiple function activations (multiple instances of the same function can be concurrently active), we call the entity of an actor associated with a function instance a *super-actor instance* or *actor instance* for short. An instance of actor *sa* in function activation *f1* is differentiated from another instance of actor *sa* in function activation *f2* by the overlay pointers OP_{f1} and OP_{f2} of the associated overlays. Any function activation and its overlay can be identified by its overlay pointer. We use the pair $\langle \text{overlay-pointer}, \text{actor-id} \rangle$ to identify an actor instance and the expression *overlay-pointer.block-id* to uniquely identify an overlay block. The overlay-pointer associated with an actor instance points to the overlay of that actor instance. Thus

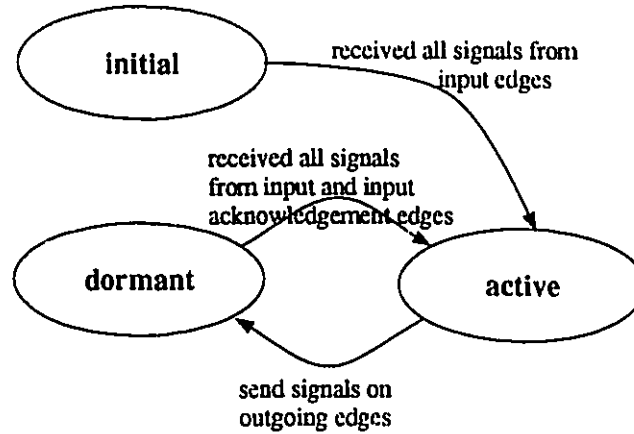


Figure 5.2: States of a super-actor instance.

a signal with an associated overlay pointer OP_{f1} destined for actor sa , will be sent to the actor instance $\langle OP_{f1}, sa \rangle$, and instructions of that actor instance will find their operands and deposit their results in overlay OP_{f1} —the overlay of actor instance $\langle OP_{f1}, sa \rangle$.

Operational Semantics of Super-Actor Instances The operational semantics of a super-actor instance can be described by the following state transitions (fig. 5.2):

- An actor instance $\langle OP_i, A-id \rangle$ is in its *initial* state when it is waiting for a signal from each of its input edges. Once it has received a signal from each one of its input arcs, it becomes active.
- The ordered list of instructions in an *active* super-actor instance $\langle OP_i, A-id \rangle$ is executed until completion. During the actor instance's execution, an instruction accesses its operand(s) from the overlay, OP_i , (overlay access is not necessary for an immediate value) and stores its result(s), if required, in the same overlay. Once the instructions have been executed, a signal with an attached OP_i will be sent along each one of the actor's outgoing edges (this includes the outgoing acknowledgement edges). Lastly, the actor instance (re)enters its dormant state.
- An actor instance is in its *dormant* state when it is waiting for a signal from each of its input edges and from each of its incoming acknowledgement arcs. Once it

has received a signal from each one of its input arcs and input acknowledgement arcs, it becomes active.

Actor instances are in their initial states when an Oalloc instruction creates the function activation to which they belong, that is, all actor instances are put into their initial states when they are created.

Operational Semantics of a Switch Super-Actor The operational semantics of a switch super-actor instance are similar to a regular super-actor instance, the only difference is that after all the instructions have been executed, a signal will be sent along each one of its unlabeled outgoing edges (this includes the outgoing acknowledgement arcs) and along the labeled outgoing edges (including labeled output acknowledgement arcs) which correspond to the condition code as generated by the last instruction.

Virtual Node Instances

Virtual nodes associated with a function instance are called *virtual node instances* or *node instances* for short.

“Operations” of Virtual Nodes The “operational semantics” of virtual node instances are as follows (below, we discuss the instances associated with a function activation with overlay pointer OP_i):

- Top node: once activated, it sends signals with the overlay pointer OP_i along its emanating edges.
- Bottom node: once it has received a signal from each of its incoming arcs, it terminates execution of the function instance associated with OP_i , i.e., actor instances associated with that function activation will no longer exist.
- merge node: when one signal is received at one of its input edges, a signal with OP_i is sent out on each of its outgoing edges.

The astute reader will note that the merge node is non-deterministic and will wonder if the appearance of such merge nodes in a SA graph might render a super-actor graph non-deterministic. In the next sections, we will see how a SA graph structured as encapsulators can be determinate.

Atomicity of a Super-Actor Instance

From the above operational semantics of (switch) super-actor instances, one can note that the execution of an actor instance is *atomic*, that is, once an actor instance becomes active, it will be executed until completion without the possibility of suspension. Treating a super-actor instance as an atomic entity at execution time resembles the treatment of actors in a dataflow machine. In this sense, the super-actors in this execution model are similar to the actors in the dataflow model, except that super-actors can contain multiple instructions; thus the term “super-actor”.

The atomic execution of super-actor instances presents two immediate benefits:

- first, resources required to execute a super-actor instance can be allocated to it only during its active state so that it cannot “hog” the resources while it is waiting on some event⁴, and
- second, the possible introduction of a unique mechanism which allows an enabled super-actor to pre-load its required data into high-speed memory so that local memory latencies can be minimized. (This mechanism is detailed in the next chapter.)

⁴This “hogging” of resources by suspended threads can cause some problems in machines such as the HEP [105]. For example, the HEP has resources for 64 live threads—threads which can be active or suspended. If the program has more than 64 threads, then the programmer must be careful in allocating resources such that deadlock due to resources does not result.

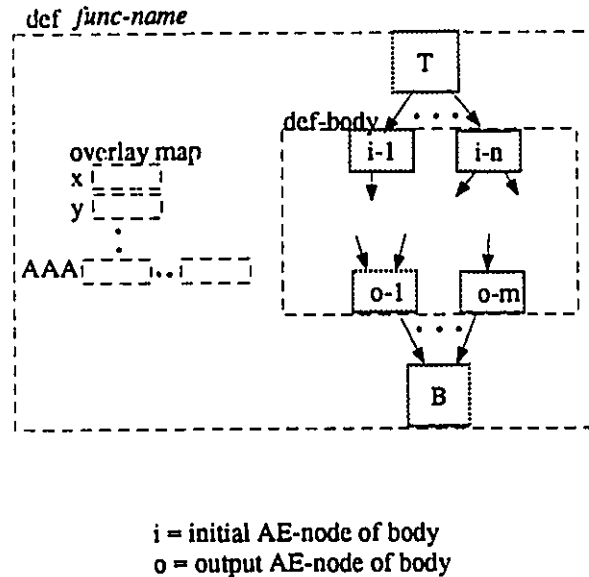


Figure 5.3: A def encapsulator in a super-actor graph.

Encapsulators

To describe the structuring of super-actor graphs, we borrow the notion of *encapsulators*—a compiler notion which groups one or more dataflow actors—from Traub’s graph representation for Id [114].⁵ Three types of encapsulators are used: a *def*, an *if-then-else* and a *loop* encapsulator. In the following descriptions of encapsulators, we use the term *AE-node* to imply either a super-actor (non-switch), an if-then-else encapsulator, or a loop encapsulator (switch super-actors are encapsulated in if-then-else and loop encapsulators).

Def Encapsulator A *def* encapsulator represents a function definition and is shown in figure 5.3. It contains an *overlay map* and a *signal flow graph* representing the super-actor graph of a function definition. The signal flow graph contains a top (T) and a bottom (B) virtual node, and a *def-body* which contains an acyclic graph of one or more AE-nodes.

⁵Traub’s encapsulators are similar to Ackerman’s encapsulators for VAL [1].

The top node has emanating edges to *initial* AE-nodes in the def-body—AE-nodes which can be activated when the function is invoked. The bottom node has input edges from the last AE-nodes (called *output* AE-nodes of the def-body) to be activated in the function. In all functions except the main function, the output AE-node to be activated would be a super-actor which returns values to the caller function.

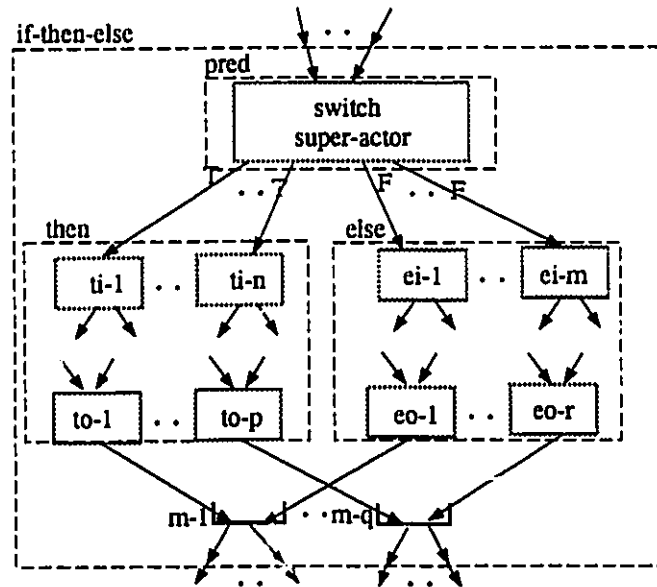
The overlay map specifies overlay locations where results of super-actors belonging to the function definition are stored. An AE-node at the top level of the def-body has unique locations in the overlay for storing its results and successors of that AE-node can find their inputs in those locations. Also, the overlay map contains a special reserved overlay block for function linkage information. We label this block 'AAA' in our discussions of function applications. (The passing of values between caller and callee functions will be illustrated later in an example of function applications.)

To start the execution of a function instance, the corresponding top node instance is activated. Eventually, the bottom node instance receives all its necessary signals and terminates the function activation.

If-Then-Else Encapsulator An *if-then-else* encapsulator is used to conceptualize the notion of conditional expressions. The if-then-else encapsulator contains a pred-body (predicate body), a then-body, an else-body, and one or more merge nodes (fig. 5.4). Inside the pred-body, there is one switch super-actor, and within a then- or else-body, there is an acyclic graph of one or more AE-nodes. The switch super-actor is the "point of control" for the encapsulator, i.e., it is responsible for triggering initial AE-node(s) in either the then- or else-body via the labeled output arcs.⁶ *Initial AE-nodes* in a body are those which receive all their input data from AE-nodes exterior of the body, and an AE-node in the body which produces data to be consumed by exterior AE-nodes or which triggers AE-nodes exterior of the body is called an *output AE-node*.⁷ Output AE-nodes in the then- and else-bodies deposit their results in the same overlay locations and are responsible for signaling the successors

⁶There is only one switch super-actor in the pred-body because the triggering of the then- or else-body depends on one boolean value; thus multiple switch super-actors are not necessary.

⁷These definitions are similar to the ones in the def-body.



ti = initial AE-node of then body ei = initial AE-node of else body
 to = output AE-node of then body eo = output AE-node of else body
 m = merge node

Figure 5.4: An if-then-else encapsulator in a super-actor graph.

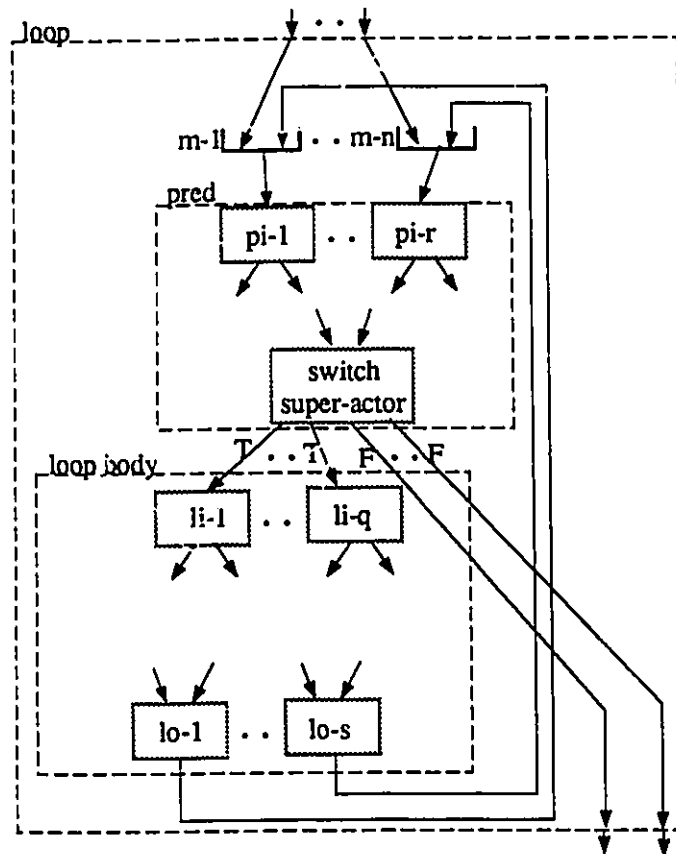
of the encapsulator through the use of merge nodes; successors of the encapsulator can find their input in those overlay locations. An AE-node other than an output AE-node at the top level of the then- or else-body has unique locations in an overlay for storing its results so that successors of that AE-node can find their input from those locations. The switch super-actor in the pred-body also has unique locations for storing its results (if any).

Note that the outgoing signal arities of the then- and else-bodies should be the same so that a merge node has two incoming arcs, one from the then-body and the other from the else-body. This does not imply that the number of output AE-nodes in the then- and else-bodies are the same since an AE-node can have multiple outgoing arcs.

Loop Encapsulator A *loop* encapsulator consists of a predicate body (*pred-body*), a *loop-body*, and one or more merge nodes (fig. 5.5). The pred-body is executed when signals are sent from the predecessor AE-nodes of the encapsulator through the merge node(s) to the initial AE-nodes in the pred-body.⁸ The predicate-body is responsible for triggering initial AE-nodes in the loop-body when a condition is met.⁹ If not, the successor AE-nodes of the encapsulator are signaled. The retriggering of the pred-body—the reiteration of the loop—is the responsibility of the output AE-node(s) in the loop-body, and the signaling from those AE-node(s) go through the same merge node(s) which have incoming edges from the predecessors of the loop encapsulator. Moreover, the output AE-nodes of the loop-body deposit values to be used by the initial AE-nodes of the pred-body into the same overlay locations as do the predecessor AE-nodes of the loop. Successors of the loop which use those values from the output AE-nodes can also find them in the same overlay locations. A non-output AE-node at the top-level of the loop-body has unique locations in an overlay for storing its results; the same applies to the AE-nodes (including the switch super-actor) in the pred-body. Note that the incoming signal arity of the encapsulator is the same as the outgoing signal arity of the loop-body.

⁸Since a switch super-actor can be the sole actor in the pred-body, then it can be considered an AE-node.

⁹Again, only one switch super-actor is necessary since the (re)activation of the loop-body is dependent on one boolean value.



li = initial AE-node of loop body pi = initial AE-node of pred body
 lo = output AE-node of loop body m = merge node

Figure 5.5: A loop encapsulator in a super-actor graph.

5.1.2 Examples of Super-Actor Graphs

In this section, corresponding super-actor graphs of a simple expression, a conditional expression, and a loop expression are presented. Lastly, function applications in a SA graph are illustrated.

Simple Expressions Examples of portions of super-actor graphs which represent the expression $(1 + p + q) - (p + q) * (r - t)$ are shown in figure 5.6. In figure 5.6, the SA graph of part (a) is a direct one-to-one mapping from the dataflow graph representation of the expression, i.e., each super-actor only contains one instruction. Parts (b) and (c) show different representations of the same expression; the difference is in the partitioning. In all three cases, the data to be used by actor *e*—represented in the figure by just the letter ‘e’ for clarity—will be found in the first location of overlay block *s4* (*s4.0*) and the remaining blocks, as shown in the figure, contain the initial values to be used by the super-actors. (Actors *p*, *q*, *r* and *t* are represented by their letters for clarity and the output values generated by those actors are put in blocks *p0*, *q0*, *r0* and *t0* respectively.) In figure 5.6(b), actors 1 and 3 from (a) are grouped to form actor 1 and actors 2, 4, and 5 from (a) form actor 2. In (c), actors 1 and 2 from (a) form actor 1 and actors 3, 4, and 5 form actor 2. The arcs between actors indicate the explicit synchronizations, and as shown in (b) and (c), the synchronization requirements are less than the one in (a).

Let us detail figure 5.6(b) further. For simplicity of the following explanation, we will assume that all signals will have the same overlay pointer attached to them, and that there is only one instance of each actor. Thus, we can call an actor instance simply as an actor and uniquely identify an overlay block by its block id. So let us begin. Instructions in actor 1 access their operands from overlay blocks *s0*, *p0* and *q0*. As for actor 2, its operands are in blocks *s4* and *s0*, and *r0* and *t0*. When signals from actors *p* and *q* are emitted (this implies that the values *p* and *q* would have been deposited in overlay blocks *p0* and *q0*), actor 1 becomes active. It executes its instructions, deposits its results into block *s0*, and signals actor 2. If actor 2 has also received signals from actors *r* and *t*, it can then be activated. Finally, actor 2 will deposit its result in overlay block *s4* and send a signal to actor *e*.

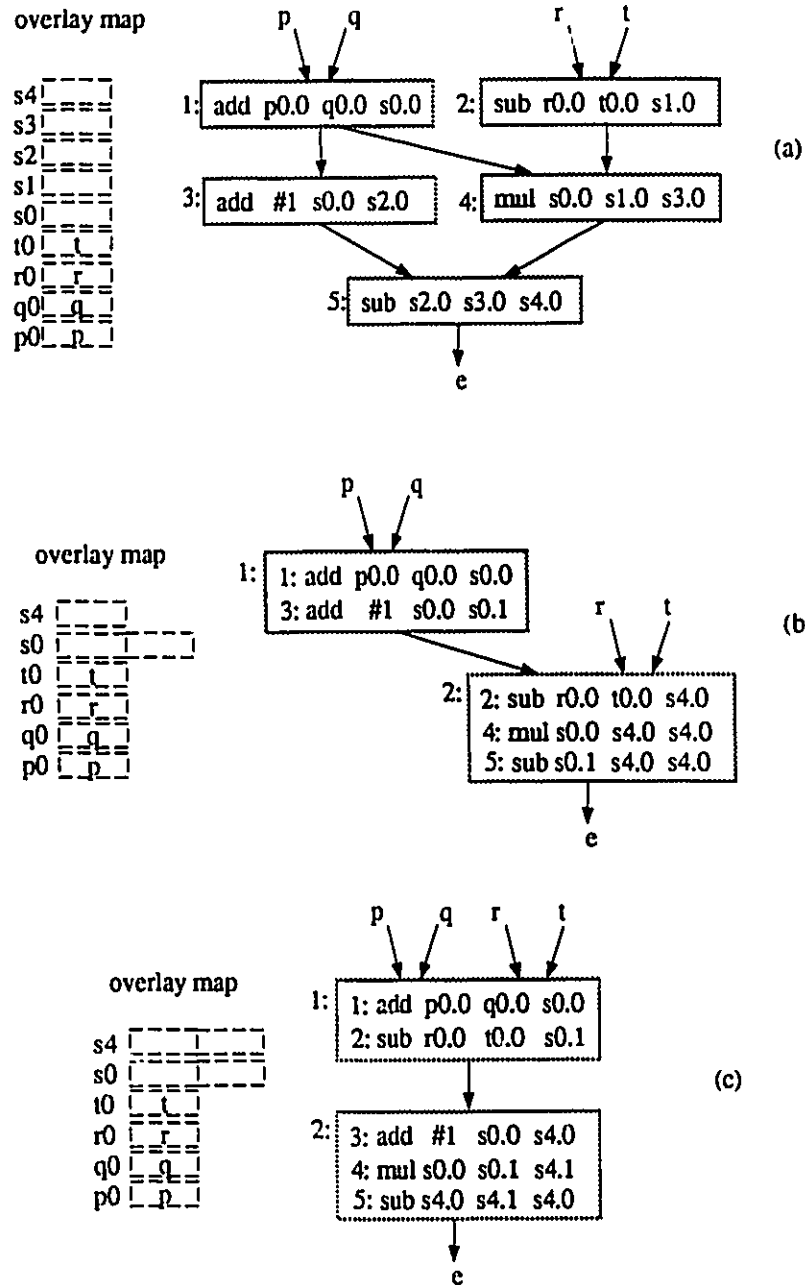


Figure 5.6: Examples of super-actor graphs.

Conditional Expressions There are two ways to represent conditional expressions in SA graphs: one uses the branch and conditional branch instructions within a super-actor, and the other is expressed in the context of an if-then-else encapsulator. In figure 5.7, the super-actor graphs represent the expression:

$$\begin{aligned}
 h := & \text{let} \\
 & a, b := \text{if } p + 1 > n \text{ then} \\
 & \quad p, g + f \\
 & \quad \text{else} \\
 & \quad \quad n, g - f \\
 & \text{in} \\
 & \quad (a + b) * 2
 \end{aligned}$$

Figure 5.7(a) shows a super-actor graph which uses branch instructions within a super-actor. Part (b) of the figure uses an if-then-else encapsulator to show a conditional expression constructed with a switch super-actor (actor 1) of the pred-body and a merge node (node 4). Actor 2 is both the initial and output super-actor (AE-node) of the then-body and actor 3 is also an initial and output super-actor, but of the else-body. Thus actor 2 has a true labeled arc from actor 1 and an output edge to the merge node, and actor 3 has a false labeled arc from actor 1 and an output edge to the merge node. The reader can note that the output super-actors in the then- and else-bodies deposit their data in the same location of the same overlay block ($a0$). Actor 5's counterpart in figure 5.7(a) is actor 2.

Though it is not shown in figure 5.7, the representation of a conditional expression in (b) has an advantage over the one in (a) in that multiple AE-nodes in either the then- or else-bodies can be invoked in parallel for increased parallelism, e.g., multiple true-labeled arcs can emanate from the switch super-actor to trigger multiple initial AE-nodes in the true-body simultaneously.

Loop Expressions A loop expression can also be expressed within a super-actor with branching instructions or in the context of a loop encapsulator. In figure 5.8, the super-actor graph for the loop:

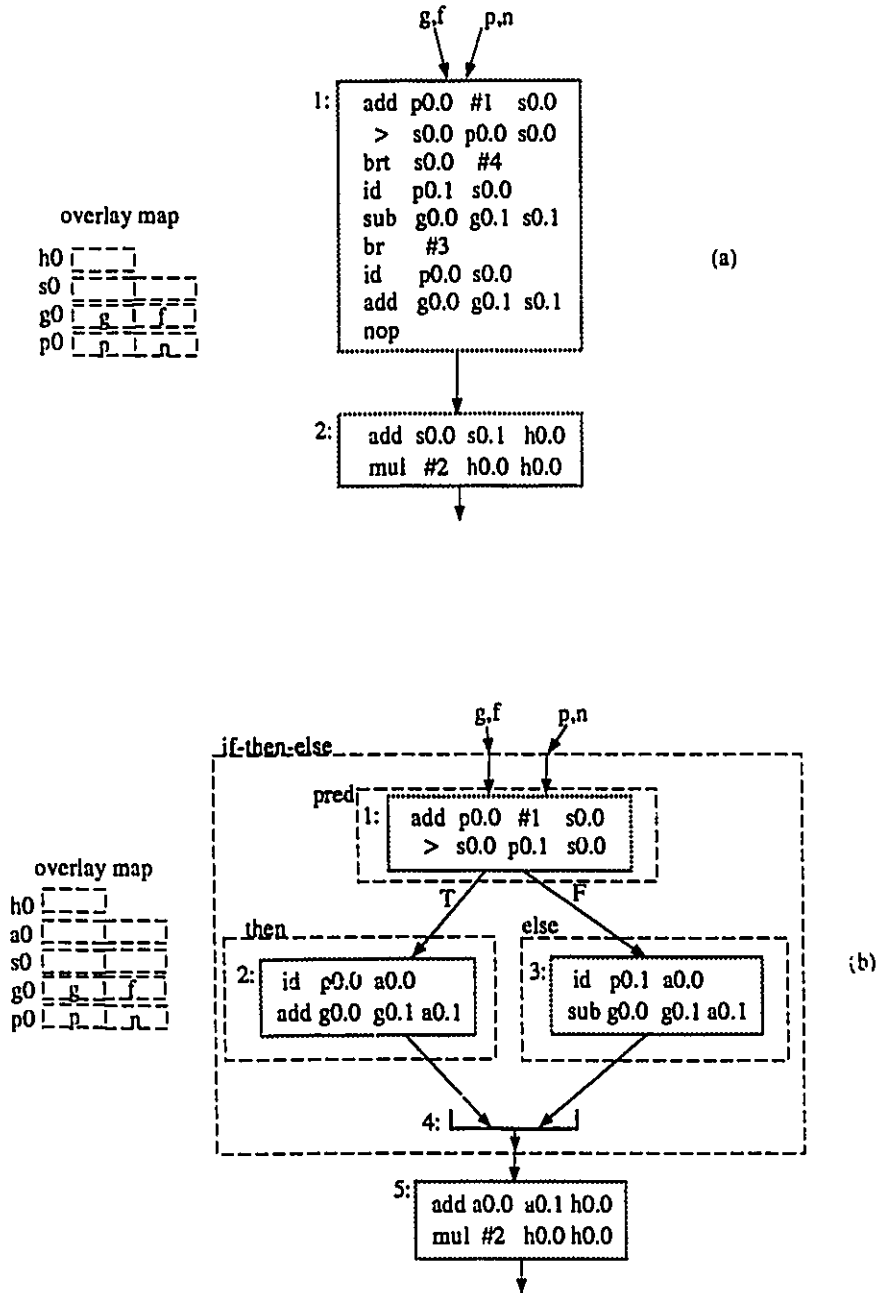


Figure 5.7: Conditional super-actor graphs.

```

for initial
   $j := 0;$ 
   $i := 0$ 
while  $i \leq 64$  repeat
   $j := A[i] + A[i + 1] + A[i + 2] + A[i + 3] + \text{old } j$ 
   $i := \text{old } i + 4$ 
returns value of  $j$ 
end for

```

is shown. Nodes 1 and 2 are merge nodes which receive signals from the predecessor AE-nodes of the encapsulator (AE-nodes i and j) and signal the initial super-actor (actor 3) in the pred-body. The predecessor AE-nodes of the loop encapsulator initialize the values of i and j to zero and the address of array 'A' and deposit them into overlay block $i0$ that is, locations $i0.0$, $i0.1$, and $i0.2$ respectively. The switch super-actor either triggers the initial AE-nodes (actors 4 and 6) in the loop-body when the value in $k0.0$ is less than 64, or it signals the successor AE-node(s) of the encapsulator. Actor 5 is responsible for the sum-reduction of the block of four elements of array A ¹⁰, and the final result of the loop is found in $i0.1$.

Function Applications Function application support in the super-actor execution model is similar to the one in the McGill Dynamic Dataflow Architecture[47]. We can have one super-actor which performs the role of an 'apply' actor and another a 'return' actor. In the super-actor execution model, an 'apply' super-actor has the following tasks:

1. allocate an overlay for the new function activation and create the actor instances (the Oalloc instruction),
2. store the necessary function linkage information in the callee's overlay,
3. copy the arguments to the callee's overlay, and
4. trigger the top node in the callee (the last three steps can be accomplished with the send instruction).

¹⁰Note that actors 4 and 5 can be combined to form one actor; the separation of the **SMread** instruction from other instructions is a prelude to the modifications as proposed in the advanced abstract machine model.

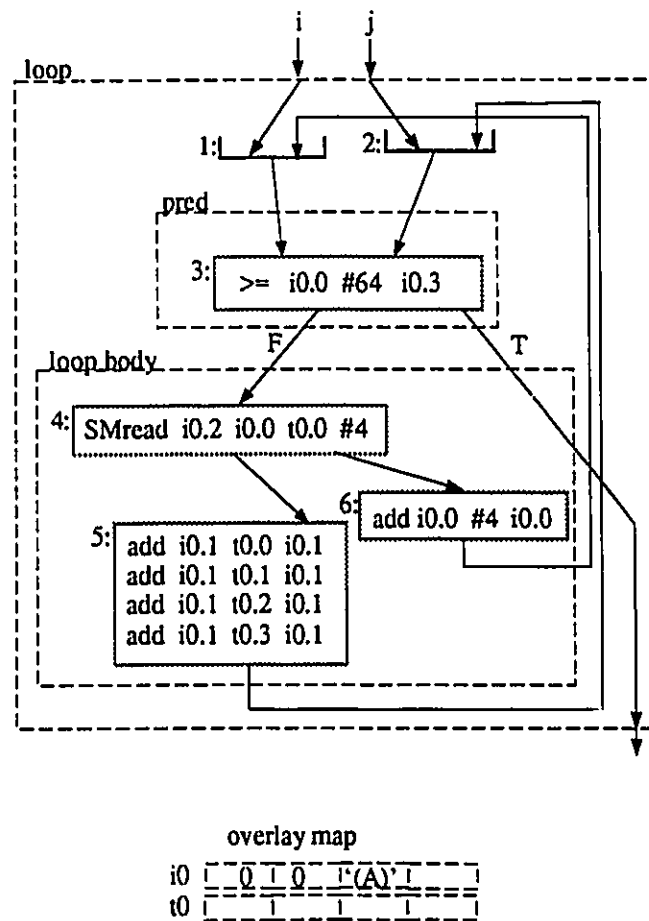


Figure 5.8: A loop encapsulator in a super-actor graph.

The 'return' super-actor has these tasks:

1. copy the return values to the caller's overlay (send instruction), and
2. deallocate the overlay (Odealloc instruction). Terminating the function instance is the responsibility of the bottom node.

Before we see how 'Oalloc', 'Odealloc' and 'send' instructions can be used to implement function applications, let us re-iterate the syntax of those instructions:

Oalloc *func-ptr res-loc*
 send *val overl-ptr loc [actor-id]*

and Odealloc takes no arguments. The argument *func-ptr* is a pointer to the function to be invoked, and *res-loc* is the location (in the caller's overlay) to store the overlay pointer of the newly created overlay. *val* is the value to be sent, *overl-ptr* is the overlay pointer of the overlay to receive the value, *loc* is the location within the overlay pointed to by *overl-ptr* where the value is to be stored, and the optional *actor-id* specifies which actor to notify once the value has been sent; the actor instance is specified by the pair $\langle overl-ptr, actor-id \rangle$.

In figure 5.9(a), we show an 'apply' super-actor in function *g* invoking a function *f*. The apply instruction within actor *appl* is a macro-instruction and the actual instructions are shown in part (b) of the figure. The syntax of the apply macro-instruction is as follows: the first operand specifies the location of the pointer to the information of the function to be invoked, the second specifies the location of the first argument to be sent, the third operand specifies the location where the first return values are to be stored, and the last operand specifies how many arguments are to be sent to the callee. Thus, the apply instruction says, invoke function *f* with the arguments *a* and *b* and store the return values in overlay block *c0*. The return instruction within actor *ret1* of part (a) of the figure is also a macro-instruction and its syntax is as follows: the first operand specifies where the first return value is located, and the second, the number of return values to send back. The dashed directed arc from the actor is not part of the syntax of the super-actor graph but is used to indicate that the return

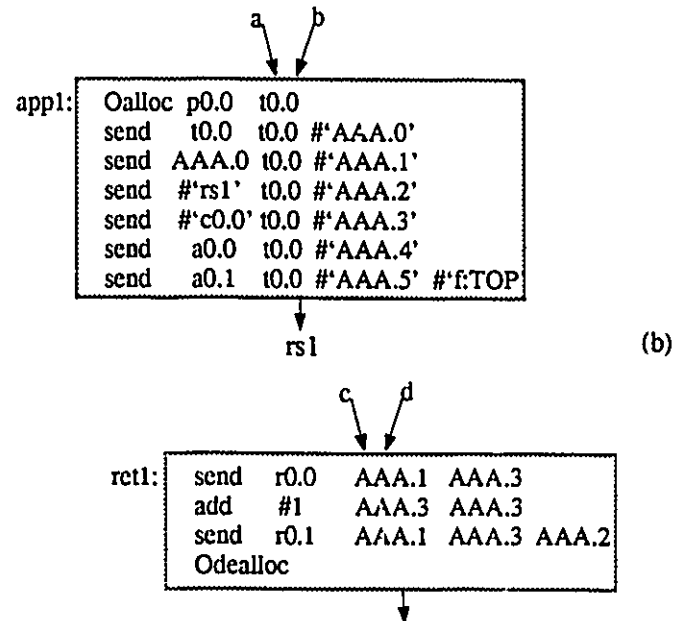
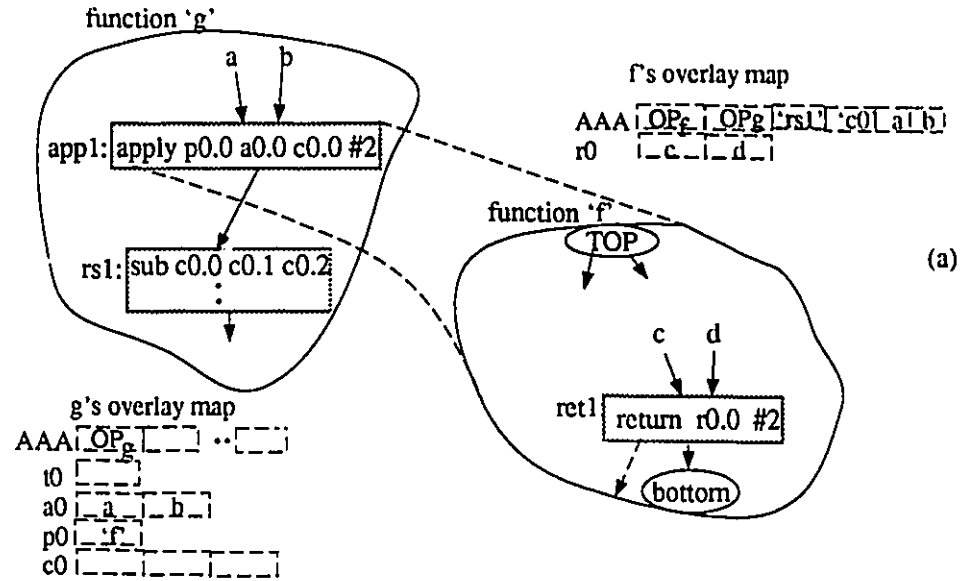


Figure 5.9: Function applications in a super-actor graph.

super-actor is responsible for sending the signals which emanate from the apply super-actor, i.e., to actor *rs1*. Note that an apply super-actor can only invoke one function since the return super-actor in the callee is responsible for signaling the successors of the apply super-actor. Multiple invocations by an apply super-actor would violate the super-actor execution model since signals “from” the apply super-actor can be emitted before the super-actor is actually finished executing, i.e., the multiple function calls have all been terminated.

Figure 5.9(b) shows the super-actors corresponding to the apply and return actors in (a). The apply super-actor first allocates the overlay for *f*. Then it sends the overlay pointer of the new function *f* to the first location in overlay block *AAA* of the new overlay so that actor instances within the new function activation can have access to their overlay pointer.¹¹ Next, the function linkage information is sent. This includes: the overlay pointer of the caller, the id of the actor (actor *rs1*) to trigger once the values are returned, and the location in the caller’s overlay (an overlay block-id and an offset) where the first return value can be found. By convention, all function linkage information are deposited into assigned locations in an overlay block labeled *AAA*. Once the linkage information are sent, the two parameters *a* and *b* (in locations *a0.0* and *a0.1*) can be sent. The last send instruction in the super-actor is responsible for signaling the top node in function *f*.

In the ‘return’ super-actor, values *c* and *d* (in block *r0* of the callee function’s overlay) are copied to the *c0* block in the caller function’s overlay and the second send instruction also triggers the super-actor labeled *rs1* in the caller. The add instruction between the two send instructions is used to increment the destination location from *c0.0* (for value *c*) to *c0.1* (for value *d*). Lastly, the *Odealloc* instruction deallocates the overlay for the function instance. (Note that with the layout of function linkage information in the *AAA* block, the return super-actor can only signal one super-actor for the apply signal actor. If more than one actor must be signaled or acknowledged, then a super-actor which does nothing but signal those actors can be inserted, and the return super-actor signals that super-actor.)

¹¹This will not be necessary in the machine model where every actor instance has access to its overlay pointer.

5.1.3 Determinate Super-Actor Graphs

As was pointed out in the review of the dataflow model (see section 1.2.1), a nice feature for parallel processing is the determinacy property where given a set of input values, the same output values will be produced by the execution of a well-formed dataflow graph [31] regardless of the firing sequence of enabled dataflow actors. This property is guaranteed as long as the execution system follows the firing rule.

We would like to have this determinacy property for the super-actor execution model, and to that end, the SA graph must be well-formed and the underlying machine model must follow the same firing rules as those imposed on a dataflow machine. The only difference is that the unit of execution is an aggregate of instructions instead of one instruction. (From the operational semantics of a super-actor instance, one can see that the firing rules are similar to those of the dataflow model.)

So what is a well-formed super-actor graph? A *well-formed* super-actor graph is an acyclic graph of AE-nodes, where an AE-node is a super-actor, an if-then-else construct (a super-actor graph with the structure of an if-then-else encapsulator), or a loop construct (a loop encapsulator). A body (pred-, then-, else-, and loop-body) in an if-then-else or loop encapsulator contains a well-formed super-actor graph, that is, a well-formed super-actor graph is *recursively defined*.

We argue that an execution of an instance of a well-formed super-actor graph on a machine which adheres to the operational semantics of the super-actor execution model exhibits the determinacy property. Proving this claim formally is beyond the scope of this thesis, so we simply outline a proof which examines the well-behavedness of a well-formed SA graph.¹² A super-actor graph is *well-behaved* if: it is *one-in-one-out*, that is, if given a set of inputs and it is notified by its input signals, a set of outputs is produced and its successors notified via output signals; it is *conflict-free*, that is, no merge node can have multiple input signals simultaneously on its input arcs during any execution of the graph; and it is *clean*, that is, after an execution of the graph, the graph is in the same state as it was

¹²In this informal argument, we refer to a super-actor graph instance as a super-actor graph for short. Also, actor instance is simply referred to as an actor, and node instance, a node.

before the execution.¹³ (Before a graph is executed, its state is such that each super-actor in the graph is waiting to receive a signal from each of its input edges.)

There are two steps to this argument. In the first step, we show that if a well-formed SA graph is given one set of inputs, it is one-in-one-out and clean via structural induction. (A def encapsulator consists of an acyclic graph (body) of one or more AE-nodes, which when invoked is executed only once).¹⁴ In the second step, we argue that a well-formed SA graph is conflict-free via contradiction.

In the first step, there are three constructs to examine: a body of one or more (non-switch) super-actors, an if-then-else construct, and a loop construct. The basis of this induction is that each individual super-actor is functional, that is, given a set of inputs, it produces a set of outputs and that it retains no state which can alter the output for the next set of inputs. Moreover, the state of an executing super-actor can only be altered by instructions of that super-actor. (Chap. 7 will outline an algorithm to generate such functional super-actors from a dataflow graph).

In brief, each super-actor in a body of one or more super-actors fires exactly once during an execution of the body, that is, when the necessary signals appear on the input arcs of the body. When the output values are produced by the body, signals on the output arcs of the body will notify the successors of the body. Therefore, a body of super-actor(s) is one-in-one-out. Since a super-actor is functional and is ready to fire when presented the next set of inputs, then a body of super-actor(s) is also clean if another set of inputs to the body is only presented when the body has finished executing.

The execution of an if-then-else construct is triggered when all inputs are produced by predecessors of the construct and all the necessary input signals are received. One of two bodies (then- or else-body) is triggered for execution by the switch super-actor in the pred-body—signals sent on labeled output arcs of the switch super-actor. Since the selected body is nested within the if-then-else construct, we may assume that the body is one-in-one-out and clean via the induction hypothesis. It is only clean if no other set of inputs

¹³The definition of well-behaved is adapted from [31] and [95].

¹⁴Reusing a function instance, for example in software pipelined code, is possible if the SA graph is structured differently. However, this is a subject for future investigation.

are presented to the if-then-else construct while the if-then-else is still executing. Once the results from the body are produced—results of the if-then-else construct—the successors of the if-then-else are triggered through the virtual merge node(s). Thus the if-then-else construct is one-in-one-out. After the output is produced, the switch super-actor is ready to fire when presented the next set of inputs because it is functional, the virtual merge nodes are ready to fire again,¹⁵ and either body is clean, thus the construct is clean if another set of inputs to the construct is not presented while the construct is executing.

When the loop is triggered for execution, the initial AE-nodes in the pred-body are triggered. Then the pred-body either triggers the loop-body for an iteration or the successors of the loop when a condition is not met. Each iteration of the loop-body uniquely determines the values for the next iteration or the exit values if the loop terminates. Since the pred- and loop-bodies are nested in the construct, we assume that they are one-in-one-out and clean if no other set of inputs are presented to the loop while the loop is executing. Once the results of the loop are produced, the switch super-actor in the pred-body triggers the successors of the loop, thus the loop construct is one-in-one-out. Every component in the construct is ready to fire when the next set of inputs are presented to the loop and no component retains any state, thus the loop construct is clean if no other set of inputs to the loop is presented while the loop is still executing.

From the analysis of if-then-else and loop constructs, they can be regarded as (non-switch) super-actors. That is, they do not logically fire until all inputs are produced and the input signals have arrived, and they send output signals to their successors when their results are produced (one-in-one-out); and they do not retain any state which can alter the output for the next set of inputs after they have executed, and they are ready for the next set of inputs upon termination (clean). Therefore, bodies containing if-then-else and loop constructs in place of super-actors are one-in-one-out and clean if no other set of inputs is presented to the body while the body is executing.

In the second step of this proof outline, we show via contradiction that the merge nodes in a well-formed SA graph cannot be in conflict, i.e., the SA graph is conflict-free. (This

¹⁵In this machine, a merge node is actually executed and its operational semantics observed.

second step is integral in showing that an AE-node does not receive another set of inputs until it has completely processed its current set of inputs; a property which the clean property is dependent upon.) Let us assume that there is conflict in a merge node in either an if-then-else or loop construct, say construct x . This can only occur if x is nested within another construct, say y , in which y produces another set of inputs to x directly and/or through other AE-nodes nested in the same body as x while x is executing (note that the body which contains x is an acyclic graph of AE-nodes). Construct y can be a body of AE-nodes (i.e., a body of a def encapsulator), an if-then-else, or a loop encapsulator. If y is a body of a def encapsulator, then we know that it is executed only once when invoked—only one set of inputs—so x only receives one set of inputs (from the first step of this proof, we note that each AE-node at the top level of a body in a def encapsulator is one-in-one-out) and thus it is impossible for the merge nodes in x to be in conflict. If y is either an if-then-else or loop construct, then the merge nodes in x can be in conflict if the merge nodes in y is in conflict, that is, if y receives a set of inputs from the construct it is nested within while y is executing. However, at the top level of this nesting is a body of a def encapsulator—a function definition (note that in the definition of the super-actor program, every function definition is represented by a def encapsulator). Therefore, x can never be presented with another set of inputs while it is executing. Thus the merge nodes of x are conflict-free and x is conflict-free.

Since a well-formed SA graph is one-in-one-out, clean, and conflict-free, then it is well-behaved. And since a well-formed super-actor graph is well-behaved, and well-behaved implies determinate [27], then the well-formed super-actor graphs are determinate.

5.1.4 Discussions

In chapter 7, we will see how well-formed super-actor graphs can be transformed from dataflow graphs.

We have yet to show the use of acknowledgement arcs in a SA graph. The first example of their use is shown when we describe dataflow software pipelined SA graphs in

section 7.3.2 (see appendix A for dataflow software pipelining). Acknowledgement arcs are used primarily for exploiting the “pipelining of operations” which is possible in the static dataflow model of computation; pipelining operations within a loop is extremely useful in exploiting inter-iteration parallelism. The well-formed SA graphs, as presented above, do not support pipelining of operations. However, adding acknowledgement arcs in the right places will remove this restriction (as shown in fig. 7.11 in section 7.3.2).

5.2 The Abstract Machine Model

In this section, we describe the abstract machine model via step-wise enhancements to the description of the *most* abstract machine model. This base model (sect. 5.2.2) corresponds roughly to the super-actor instance state transitions in the abstract program execution model. (The definition of symbols as used in the models are found in sect. 5.2.1 and the instruction set for the machine models is found in sect. 5.2.3.) The second model, which we have called the “Intermediate Abstract Machine Model”, corresponds to the stage where two extra states are added to the state transitions to facilitate the pre-loading of high-speed memory and the reservation of physical resources required to execute a super-actor instance (sect. 5.2.4). Finally, the last model includes the feature of separating ‘regular’ super-actors from those which have different processing requirements so that the execution of regular super-actors can be unhindered (sect. 5.2.5). Extensions to the syntax of super-actors and refinements to the semantics of actor instances will be introduced accordingly.

Describing the machine model in this manner allows the reader to readily comprehend the basic features of the Super-Actor Machine. Moreover, it also facilitates future compiler optimization efforts and enhancements to the SAM itself by isolating the features which we think are beneficial in today’s technology but which may not be in the future as technology evolves.

5.2.1 Symbols Used in the Models

The symbols appearing in this section are used in conjunction with the associated memory model of the three abstract machine models. Where appropriate, other symbols will be introduced as the machine model is enhanced.

First, let us define an actor in this machine model by describing its components. An instance of an actor can be identified by the base address of the function overlay (overlay pointer = *base address*) the actor instance belongs to and the actor's id (*id*):

$$\langle \text{actor-instance-id} \rangle ::= \langle \text{base-address} \rangle \langle \text{id} \rangle$$

Attributes of an actor, which we term the *static attributes* can be accessed via its *id* value. Attributes of an instance of an actor, called the *dynamic attributes*, can be accessed via a combination of the *base-address* and *id* value. Thus, an actor instance can be wholly defined by its static and dynamic attributes:

$$\langle \text{actor-instance} \rangle ::= \langle \text{static-attributes} \rangle \langle \text{dynamic-attributes} \rangle$$

$$\langle \text{static-attributes} \rangle ::= \langle \text{instr-list} \rangle \langle \text{reset-count} \rangle \langle \text{init-count} \rangle \\ \langle \text{signal-lists} \rangle$$

$$\langle \text{instr-list} \rangle ::= [\langle \text{instruction} \rangle]^0$$

$$\langle \text{reset-count} \rangle, \langle \text{init-count} \rangle ::= \text{integer}$$

$$\langle \text{signal-lists} \rangle ::= \langle \text{uncond-list} \rangle | \\ \langle \text{uncond-list} \rangle \langle \text{t-list} \rangle \langle \text{f-list} \rangle$$

$$\langle \text{uncond-list} \rangle, \langle \text{t-list} \rangle, \langle \text{f-list} \rangle ::= \text{list of actor ids}$$

(To remind the reader, expressions of the form $[\langle xx \rangle]^0$ implies zero or more $\langle xx \rangle$ tokens.)

The static attributes of an actor instance include its list of instructions, a reset enable count,

an initial enable count and its signal lists. Note that the *reset-count*, *init-count*, and *signal-lists* are used to represent the arcs in the signal flow graph of the super-actor graph. The *init-count* specifies the number of input arcs a super-actor has and the *reset-count* specifies the number of input arcs plus the number of input acknowledgement arcs of a super-actor. The *signal-lists* represent the output edges of a super-actor. Merge nodes are encoded within the signal lists, reset-counts and init-counts of actors, thus they do not exist in the machine model. (For example, actor x signals merge node m which in turn signals actor y , actor z also signals y through m . In the unconditional signal-lists of x and z , there is simply an entry to y in each list. And in the reset-count and init-count of y , only one signal is needed to activate y .)

$\langle \text{instruction} \rangle ::= \langle \text{optr} \rangle [\langle \text{opr-res-field} \rangle]^0$

$\langle \text{optr} \rangle ::= \text{'add'} \mid \text{'sub'} \mid \text{etc.}$

$\langle \text{opr-res-field} \rangle ::= \langle \text{opr-res-type} \rangle \langle \text{opr-res-val} \rangle$

$\langle \text{opr-res-type} \rangle ::= \text{'reg'} \mid \text{'immed'} \mid \text{'slot-ptr'}$

$\langle \text{opr-res-val} \rangle ::= \text{integer} \mid \langle \text{slot-ptr} \rangle$

$\langle \text{slot-ptr} \rangle ::= \langle \text{block-id} \rangle \langle \text{offset} \rangle$

$\langle \text{block-id} \rangle ::= \text{offset from overlay base address locating the first element of overlay block}$

$\langle \text{offset} \rangle ::= \text{integer}$

In the super-actor execution model, the type of operand/result field of an instruction could only be an immediate value or an overlay slot pointer (*slot-ptr*) which points to a location within the actor instance's overlay. In the machine model, we have added a register type—*oper-res-type* equals *reg*—where a temporary register can be used to communicate values between instructions of the same super-actor instance; temporary registers can not be used for any other purposes. This addition allows a super-actor instance to use less overlay space than if no temporary registers were used. In an operand/result field of an instruction

in a super-actor graph, the form ' R_y ' is used to indicate the y th register which contains an operand or is the destination of the instruction's result.

The dynamic attributes of an actor instance are:

$\langle \text{dynamic-attributes} \rangle ::= \langle \text{enable-count} \rangle \langle \text{running-status} \rangle \langle \text{cond-code} \rangle$

$\langle \text{enable-count} \rangle ::= \text{integer}$

$\langle \text{running-status} \rangle ::= \text{'terminated'} \mid \text{'non-terminated'}$

$\langle \text{cond-code} \rangle ::= \text{'true'} \mid \text{'false'} \mid \text{'uncond'}$

Each actor instance maintains its own enable count (number of signals to be received before the instance can be enabled), a running status, and a condition code. The *running status* is used by the machine to determine which actor instance has just finished executing and the condition code (*cond-code*) is used to fetch the appropriate signal lists when an actor instance has just been executed.

The Memory Units There are three basic memory units—all having linear addressing—in the machine models: *Actor Attribute Memory* ($AA[]$), *Instruction Memory* ($I[]$) and *Data Memory* ($D[]$). In the following machine models, the static attributes of an actor can be accessed via the actor's id and corresponding attribute pointers into actor attribute memory. Instructions of an actor are stored in the instruction memory and can be accessed with the actor's id. Data memory contains the function overlays (i.e., operands, results and dynamic attributes of actor instances), and structure memory objects. The overlay and structure memory spaces as differentiated in the super-actor execution model are combined into one space—the data memory—in the abstract machine model. That is, overlay pointers and structure memory identifiers (entities from the super-actor graph execution model) become data memory addresses, and overlay block ids (again, from the super-actor execution model) become offset values, i.e., offsets from an overlay pointer (a data memory address). Overlay blocks from the execution model are called *data blocks* in the machine model. Information

within overlays can be accessed with offset values stored in the actor attribute memory and instructions within the instruction memory are accessed with addresses which are also stored in the actor attribute memory. The symbols for component locations of an actor with id AP (called actor AP for short) in the actor attribute memory are:

Symbol	Points to (locations in actor attribute memory $AA[]$):
$AP.I$	address in instruction memory containing first instruction of actor AP
$AP.N_I$	no. of instructions of actor AP
$AP.C_r$	reset count value of actor AP
$AP.C_i$	initial count value of actor AP
$AP.SL_u$	first actor id in the unconditional signal list of actor AP
$AP.SL_t$	first actor id in the true signal list of actor AP
$AP.SL_f$	first actor id in the false signal list of actor AP
$AP.N_{Su}$	no. of ids in unconditional signal list of actor AP
$AP.N_{St}$	no. of ids in true signal list of actor AP
$AP.N_{Sf}$	no. of ids in false signal list of actor AP
$AP.ec$	offset from OP which locates the enable count of actor instance $\langle OP, AP \rangle$ in data memory
$AP.cc$	offset from OP which locates the condition code of actor instance $\langle OP, AP \rangle$ in data memory
$AP.rs$	offset from OP which locates the running state of actor instance $\langle OP, AP \rangle$ in data memory

An overlay pointer (OP) in tandem with an actor's id (AP) uniquely identifies an instance of an actor, $\langle OP, AP \rangle$.

5.2.2 The Basic Abstract Machine Model

The base abstract machine model of the SAM consists of two pools: the Dormant and Active Pools for containing actor instances (fig. 5.10). To move the actor instances between pools, we introduce three agents into the model: the activation, the execution, and the deactivation agents (fig. 5.10). These three agents modify the state of an actor instance and thus directly access and alter the memories as shown in the associated memory model (fig. 5.11). To

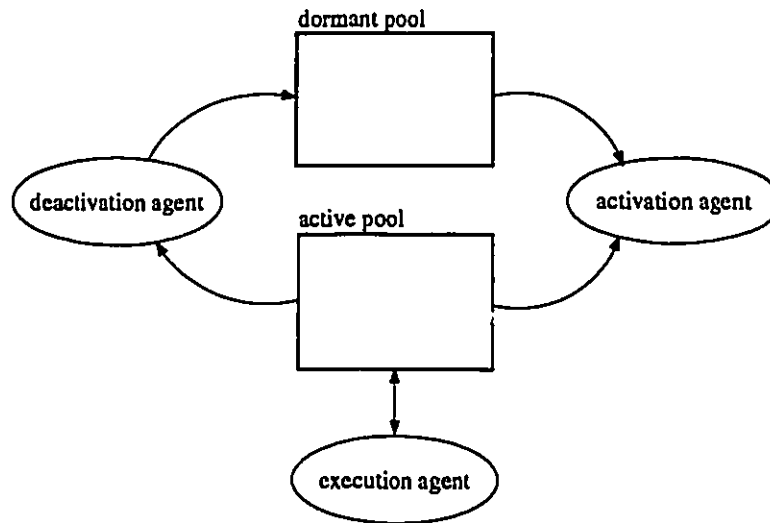


Figure 5.10: The Basic Abstract Machine Model.

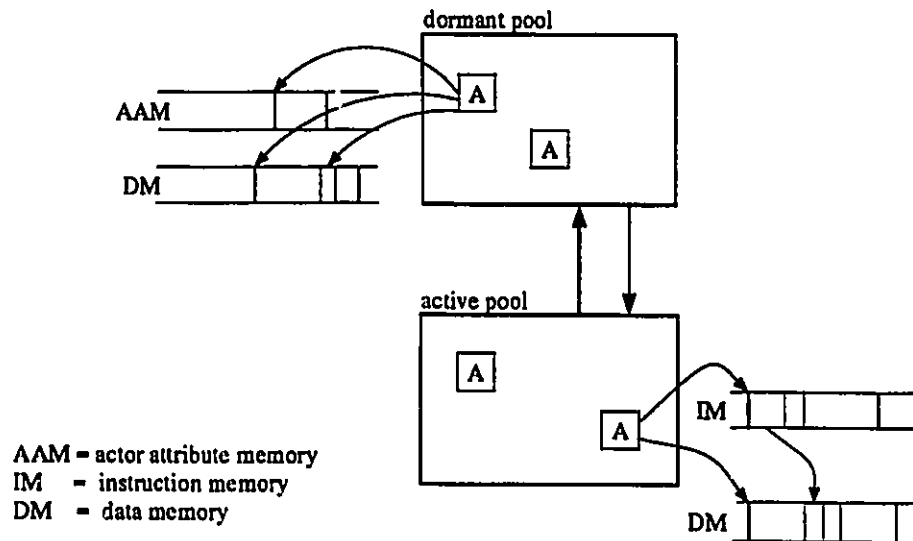


Figure 5.11: The Basic Memory Model.

start the execution of a super-actor program, an overlay for the main super-actor graph is created in data memory and instances of actors belonging to the main SA graph are put into their initial states and deposited into the dormant pool, i.e., an instance of the main function is created. Then the enable count (value in $D[OP + AA[AP.ec]]$) of the top node instance of the main function instance is set to zero. Lastly, the agents are started for execution. Below are the procedures as executed by each agent; procedure *activate* for the activation agent, procedure *execute* for the execution agent, and *deactivate* for the deactivation agent.

```

procedure activate ()
  do (forever)
    from Dormant Pool, pick  $\langle OP, AP \rangle$  with  $D[OP + AA[AP.ec]] = 0$ 
     $D[OP + AA[AP.ec]] := AA[AP.C_r]$ 
    put  $\langle OP, AP \rangle$  in Active Pool
     $D[OP + AA[AP.rs]] := \text{'non-terminated'}$ 
  enddo

```

The activation agent is responsible for picking actor instances in the dormant pool which have their enable counts equal to zero; i.e., they are enabled. It resets an enabled actor's enable count to the actor's reset value and puts the actor instance into the active pool.

```

procedure execute ()
  do (forever)
    from Active Pool, pick  $\langle OP, AP \rangle$  with  $D[OP + AA[AP.rs]] = \text{'non-terminated'}$ 
     $cnt := 0$ 
    while ( $cnt < AA[AP.N_I]$ ) do
      perform ( $OP, AP, I[AA[AP.I] + cnt], AA[AP.N_I], cnt$ )
       $cnt := cnt + 1$ 
    endwhile
    put  $\langle OP, AP \rangle$  back in Active Pool
     $D[OP + AA[AP.rs]] := \text{'terminated'}$ 
  enddo

```

The *execute* agent simply executes the instructions in an active actor instance. The *cnt* variable in the *execute* procedure acts as the program counter for the active actor instance and can be modified by branch instructions, as explained later.

The procedure *perform* is a big case statement handling the different instructions of the abstract machine. A description of the instruction set best illustrates this procedure and appears in the next section.

```

procedure perform (OP, AP, Instr, no.-instr, cnt)
  case Instr
    .
    .
    .
  endcase

```

```

procedure deactivate ()
  do (forever)
    from Active Pool, pick (OP, AP) with  $D[OP + AA[AP.rs]] = \text{'terminated'}$ 
    /* send unconditional signals */
    do ( $i := 0$  to  $(AA[AP.N_{su}] - 1)$ )
       $AP' := AA[AP.SL_u + i]$ 
      decrement  $D[OP + AA[AP'.ec]]$ 
    enddo
    if ( $D[OP + AA[AP.cc]] = \text{'true'}$ ) then
      /* send signals to actors in true signal list */
      do ( $i := 0$  to  $(AA[AP.N_{st}] - 1)$ )
         $AP' := AA[AP.SL_t + i]$ 
        decrement  $D[OP + AA[AP'.ec]]$ 
      enddo
    if ( $D[OP + AA[AP.cc]] = \text{'false'}$ ) then
      /* send signals to actors in false signal list */
      do ( $i := 0$  to  $(AA[AP.N_{sf}] - 1)$ )
         $AP' := AA[AP.SL_f + i]$ 
        decrement  $D[OP + AA[AP'.ec]]$ 
      enddo
    enddo

```

In procedure *deactivate*, the condition code of an actor instance is checked to see if actors in the true or false signal list must be sent signals. Regardless, the actors in the unconditional list are always sent signals. An actor instance which receives a signal simply gets its enable count value decremented. Once all signals for a terminated actor instance have been sent, that actor instance is put back into the dormant pool where it waits to be re-activated again.

5.2.3 Instruction Set of the Abstract SAM Model

In the following instructions, operand/result fields are represented by the form *pxx*, where *xx* can be an integer value between one and four. The fields have three addressing modes (see definition of *<opr-res-type>*), and the following macro substitution is used:

```
macro * (pxx)
  case (pxx.opr-res-type)
    reg:    reg[pxx.opr-res-val]
    immed:  pxx.opr-res-val
    slot-ptr: D[OP + pxx.opr-res-val.block-id + pxx.opr-res-val.offset]
  endcase
```

The fields within *pxx* are taken from the definition of *<op-res-field>* as found on page 83. In the following instruction set definition, we use *reg*, *immed*, and *sp* to indicate the register, immediate, and slot pointer addressing modes of *pxx*, respectively. In the semantic description of instructions, the form **pxx* is used as a shorthand representation for the macro call **(pxx)*. For example, the expression “**pl*” in a semantic description is replaced by the expression “*reg[pl.opr-res-val]*” if *pl* specifies the register addressing mode. This macro is similar to macros in Lisp, that is, everytime the macro is encountered, an expression substitution is performed and then the whole statement containing the new expression is evaluated.

Lastly, the semantic description below uses the parameters passed into the *perform* procedure as listed above.

Arithmetic Operations These instructions are of the standard 3-address and 2-address varieties where **abinop** are operations like 'add', 'mul', etc., and **aunop** are operations like 'abs', 'id', etc.

instruction:	abinop $p1\ p2\ p3$
addressing modes:	$p1, p2 ::= \text{reg} \mid \text{sp} \mid \text{immed}$ $p3 ::= \text{reg} \mid \text{sp}$
semantic:	$*p3 := *p1\ \text{abinop}\ *p2$ if ($\text{cnt} = \text{no.-instr} - 1$) then $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'uncond'}$

The *if-then* expression at the end of semantic description is used to check if the instruction is the last one in the super-actor. If so, then the condition code of the actor instance is set accordingly.

instruction:	aunop $p1\ p2$
addressing modes:	$p1 ::= \text{reg} \mid \text{sp} \mid \text{immed}$ $p2 ::= \text{reg} \mid \text{sp}$
semantic:	$*p2 := \text{aunop}\ *p1$ if ($\text{cnt} = \text{no.-instr} - 1$) then $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'uncond'}$

Relational and Logic Operations These operations are also of the standard 3-address or 2-address types. **lbinop** are operations like '<', '≤', 'and', etc., and **lunop** are operations like 'not', 'zero', etc.

instruction:	lbinop $p1\ p2\ p3$
addressing modes:	$p1, p2 ::= \text{reg} \mid \text{sp} \mid \text{immed}$ $p3 ::= \text{reg} \mid \text{sp}$
semantic:	$*p3 := *p1\ \text{lbinop}\ *p2;$ if ($\text{cnt} = \text{no.-instr} - 1$) then if ($*p3 = \text{'T'}$) then $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'true'}$ else $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'false'}$

instruction:	lunop $p1\ p2$
addressing modes:	$p1 ::= \text{reg} \mid \text{sp} \mid \text{immed}$ $p2 ::= \text{reg} \mid \text{sp}$
semantic:	$*p2 := \text{aunop } *p1$ if ($\text{cnt} = \text{no.-instr} - 1$) then if ($*p3 = \text{'T'}$) then $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'true'}$ else $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'false'}$

To control the evaluation of actors, the above relational and logic operations are used at the end of a switch super-actor, as previously mentioned.

Control Operations These instructions—unconditional and conditional branches—only control the stream of evaluation *within* an actor:

instruction:	br $p1$
addressing modes:	$p1 ::= \text{reg} \mid \text{sp} \mid \text{immed}$
semantic:	$\text{cnt} := \text{cnt} + *p1 - 1$ if ($\text{cnt} < -1$ or $\text{cnt} \geq \text{no.-instr} - 1$) then error

Note that after an execution of a branch instruction, the instruction counter, cnt , is one less than the desired counter value, however, it will be incremented by one in the while loop of the *execute* procedure before the next instruction is fetched.

instruction:	brx $p1\ p2$, where $x = \text{t} \mid \text{f}$
addressing modes:	$p1 ::= \text{reg} \mid \text{sp}$ $p2 ::= \text{reg} \mid \text{sp} \mid \text{immed}$
semantic:	if ($*p1 = x$) then $\text{cnt} := \text{cnt} + *p2 - 1$ if ($\text{cnt} < -1$ or $\text{cnt} \geq \text{no.-instr} - 1$) then error

Function Applications The following **send** instruction along with overlay management instructions are used to perform function applications. The **send** instruction can also be used for interprocessor communications. (In interprocessor communication, the base address $p2$ serves a dual role by also indicating which processing element to send the data.) The **send** instruction takes three operands and an optional fourth operand (the square brackets indicate an optional argument). The parameter $p1$ specifies the value to be sent, $p2$ is the overlay pointer indicating which overlay is to receive the value, and $p3$ is the offset value from $p2$ locating where in the overlay the sent value is to be stored. The $p4$ argument is an actor id, and if it is specified, the actor instance $\langle p2, p4 \rangle$ is to be sent a signal when the send has completed.

instruction:	send $p1\ p2\ p3\ [p4]$
addressing modes:	$p1, p2, p3, p4 ::= \text{reg} \mid \text{sp} \mid \text{immed}$
semantic:	$D[*p2 + *p3] := *p1$ if ($p4$ specified) then decrement $D[*p2 + AA[(p4).ec]]$ if ($cnt = no.-instr - 1$) then $D[OP + AA[AP.cc]] := \text{'uncond'}$

In function applications, the **send** instruction is used to pass a parameter, $p1$, between caller and callee.

Overlay Management Operations To allocate and deallocate overlays, **Oalloc** and **Odealloc** instructions are used. The $p1$ argument in **Oalloc** specifies a pointer to function information and $p2$ will contain the overlay pointer of the newly allocated overlay.

instruction:	Oalloc <i>p1 p2</i>
addressing modes:	<i>p1</i> ::= reg sp immed <i>p2</i> ::= reg sp
semantic:	allocate an overlay, its size is specified in the function information pointed to by <i>*p1</i> <i>OP'</i> := base address of new overlay <i>*p2</i> := <i>OP'</i> for (each actor, <i>AP'</i> , of new function) do <i>D[OP' + AA[AP'.ec]]</i> := <i>AA[AP'.Ci]</i> endfor if (<i>cnt</i> = <i>no.-instr</i> - 1) then <i>D[OP + AA[AP.cc]]</i> := 'uncond'

instruction:	Odealloc
semantic:	deallocate overlay pointed to by <i>OP</i> <i>D[OP + AA[AP.cc]]</i> := 'uncond'

As alluded to in the above semantics, the **Odealloc** instruction is to be the last one within a super-actor and that an instance of that super-actor be the last one to be activated in a function instance.

Structure Memory Operations These instructions are for creating, deleting, and accessing data structures which may represent arrays, vectors, etc.

instruction:	SMalloc <i>p1 p2</i>
addressing modes:	<i>p1</i> ::= reg sp immed <i>p2</i> ::= reg sp
semantic:	allocate a block of contiguous memory of size <i>*p1</i> with <i>SMP</i> as the memory address of the first word <i>*p2</i> := <i>SMP</i> if (<i>cnt</i> = <i>no.-instr</i> - 1) then <i>D[OP + AA[AP.cc]]</i> := 'uncond'

instruction:	SMdealloc $p1$
addressing modes:	$p1 ::= \text{reg} \mid \text{sp}$
semantic:	deallocate structure memory object pointed to by $*p1$ if ($\text{cnt} = \text{no.-instr} - 1$) then $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'uncond'}$

In operations **SMread** and **SMwrite**, $p1$ represents the base address of the structure memory (SM) object and $p2$ is the offset. The operand $p3$ indicates a location within the overlay. If $p4$ is specified and is greater than one, then it implies multiple reads or writes. A **SMread** means that value(s) in the SM object are deposited into overlay locations and a **SMwrite** copies value(s) in overlay locations to the SM object. In the semantic description below, the function $\text{address}(x)$ returns the memory address of the data block pointer x .

instruction:	SMread $p1\ p2\ p3\ [p4]$
addressing modes:	$p1, p2, p4 ::= \text{reg} \mid \text{sp} \mid \text{immed}$ $p3 ::= \text{sp}$
semantic:	$*p3 := D[*p1 + *p2]$ if ($p4$ specified and $*p4 > 1$) then $j := \text{address}(p3)$ do $i := 1$ to $(*p4 - 1)$ $D[j + i] := D[*p1 + *p2 + i]$ if ($\text{cnt} = \text{no.-instr} - 1$) then $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'uncond'}$

instruction:	SMwrite $p1\ p2\ p3\ [p4]$
addressing modes:	$p1, p2, p4 ::= \text{reg} \mid \text{sp} \mid \text{immed}$ $p3 ::= \text{sp}$
semantic:	$D[*p1 + *p2] := *p3$ if ($p4$ specified and $*p4 > 1$) then $j := \text{address}(p3)$ do $i := 1$ to $(*p4 - 1)$ $D[*p1 + *p2 + i] := D[j + i]$ if ($\text{cnt} = \text{no.-instr} - 1$) then $D[\text{OP} + \text{AA}[\text{AP.cc}]] := \text{'uncond'}$

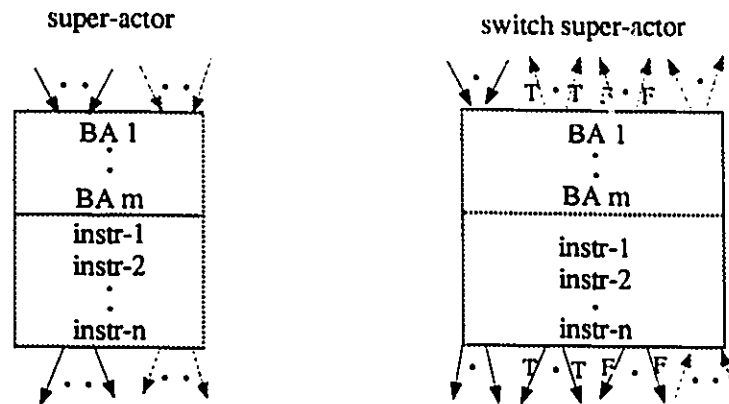


Figure 5.12: Super-actors in the Intermediate SA Graph Model.

Miscellaneous Instructions

instruction:	nop
semantic:	if ($cnt = no.-instr - 1$) then $D[OP + AA[AP.cc]] := 'uncond'$

5.2.4 The Intermediate Abstract Model

From the basic abstract machine model of the SAM, we note that the instructions of a super-actor along with the data blocks containing the operands of a super-actor's instructions can be pre-loaded into fast buffer memory such that the average memory accessing times of instructions can be decreased when a super-actor is active.

Refinements to the Super-Actor Graph

To reflect this enhancement, we first modify the syntax of a super-actor to include a group of block assignments, BA_i (fig. 5.12). A *block assignment* is basically an instruction to the machine indicating which block of data is to be pre-loaded into fast buffer memory. Each

block assignment specifies a block id to be assigned to a temporary variable and has the form

$$\text{temp-block-name} := \text{block-val}$$

where *temp-block-name* has the form '*B*'*xx* and *xx* is an integer value, and *block-val* is an overlay block-id. The reason for introducing *temp-block-name* is so that during the execution of an active actor's instructions, the machine does not have to perform an associative search in the fast buffer memory to look for the appropriate locations to access; it simply accesses some fast memory (actually, a register set assigned to each active actor instance) indexed with *Bxx* values for pointers to the assigned fast buffer memory blocks. Therefore, accessing operands and result locations can all be performed without having to access slower data memory. To facilitate this indirection in retrieving operands or storing results, instructions which access a an overlay slot will now indirectly specify it via a *temp-block-name*, i.e., *Bxx* is used as a block pointer in the operand/result field of an instruction.¹⁶ (In the original definition of the operand/result field—see definition of *op-res-field* on page 83—the *slot-ptr* component for the 'slot-ptr' addressing mode consists of a block-id and an offset. Here, the block-id is replaced with a block pointer, *Bxx*.)

Figure 5.13 shows a super-actor graph with the new super-actor syntax. This super-actor graph is a translation from the super-actor graph of figure 5.6(b). Note that the operand and result fields of instructions indirectly refer to the appropriate data block via a *Bxx* variable. An element within a block is still accessed via an offset.

Refinements to the Super-Actor Execution Model

To reflect the pre-loading enhancement in the super-actor execution model, we introduce two intermediate states separating the dormant and active states of a super-actor instance: the *enabled* and *ready* states (fig. 5.14). The *enabled* state is reserved for actor instances which have just received all their signals and the *ready* state serves as an indication that the actor instance has its necessary instructions and data pre-loaded. The *ready* state also

¹⁶In the implementation of the SAM, *Bxx* values will be used to index into a set of registers containing pointers to memory blocks in a high-speed memory device called the register-cache.

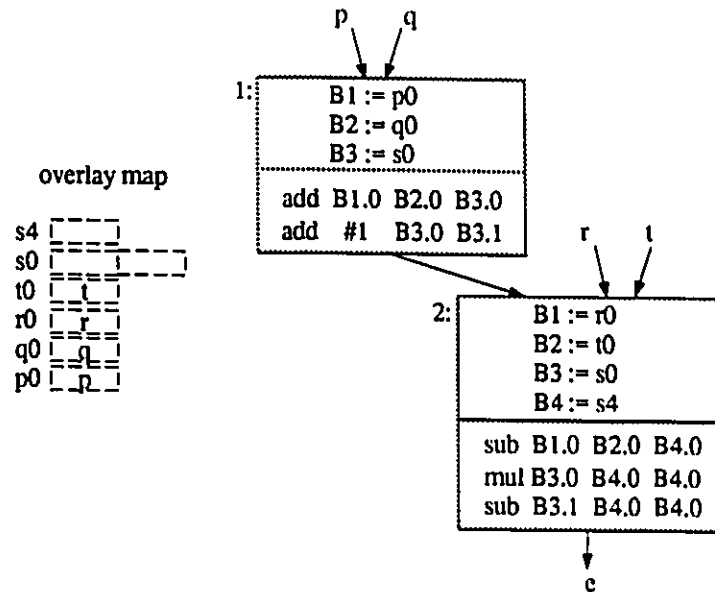


Figure 5.13: A super-actor graph with the new super-actor syntax.

indicates that physical resources such as an instruction counter, and temporary registers need to be reserved before the actor instance can become active. Having the two extra states allows the pre-loading of data and the reservation of resources to be performed simultaneously on two different actor instances since those tasks are independent.

The operational semantics of a super-actor instance in this model is described by the following state transitions:

- An actor instance is in its *initial* state when it is waiting for a signal from each of its input edges. It becomes enabled when it has received a signal from all input edges. (In the machine model, the number of remaining signals to be received before the actor instance is enabled is indicated by its *enable-count*. The *init-count* is used to initialize the enable count of an actor instance when a function is activated, and once a super-actor instance has been enabled—its enable count is zero—the enable count is reset to the *reset-count* value.)
- For an *enabled* super-actor instance to make a transition into the ready state, the following must be prepared: all of the blocks of data as specified in the

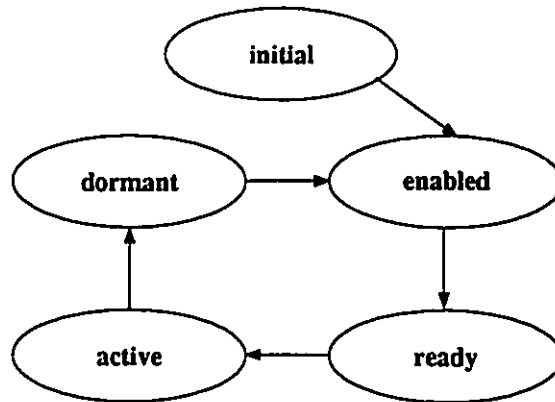


Figure 5.14: Super-actor instance states in the intermediate machine model.

block assignments along with the instructions of the super-actor must be in fast buffer memory.

- A *ready* super-actor instance enters the active state when it is assigned an available physical domain (context) to keep track of its running state during its active phase. (In our machine model, the running state of an actor instance includes a pointer to which instruction is currently being processed, a status word, and some temporary registers.)
- Instructions in an *active* super-actor instance $\langle OP_i, A-id \rangle$ can be executed. When the execution has terminated, its assigned context will be released. Finally, the active super-actor instance will send signals with overlay pointer OP_i on its outgoing edges (in the machine model, signals will be sent to actors in the signal list¹⁷), and (re)enter the dormant state.
- An actor instance is in its *dormant* state when it is waiting for a signal from each of its input edges and input acknowledgement edges. It becomes enabled when a signal from each its input edges and input acknowledgement edges has arrived. (In the machine model, the enable count becomes zero and is reset to the reset count value.)

The reader may note that the original execution model is still preserved in the sense that

¹⁷For a switch super-actor instance, actors in a selected signal list which correspond to a computed condition code will also be sent signals.

an actor instance must still receive all signals before it can be executed, and once enabled, it must execute to completion before it notifies other actor instances. The additions—additional states and transitions—are merely aimed at efficiency issues in the machine model. For example, let us look at figure 5.6(a). If super-actor 1 and 2 are simultaneously enabled, and the execution agent can only process one active super-actor at a time, then while super-actor 1 is executed upon, super-actor 2 is not processed. In the base abstract machine model, the operands as used by the super-actors are not fetched until needed, and if fetching from data memory takes longer than a basic execution cycle, then the execution agent may have to wait for memory requests to be fulfilled. In the intermediate machine model, while super-actor 1 is being processed, the operands as required by super-actor 2 can be pre-fetched into fast memory such that when it is time for actor 2 to be processed, the execution agent does not have to wait for the memory requests of actor 2.

Refinements to the Abstract Machine Model

In the abstract machine model, overlay space and structure memory space are represented as data memory, that is, we can access structure memory objects as easily as we access overlays. Therefore, we can divide a SM object (represented as a contiguous memory space in data memory) into *structure memory blocks* where each block has one or more elements of a SM object and can be identified by an offset value from the structure memory object id (a data memory address). With this representation, the block assignments in a super-actor can specify a structure memory block via an indirection; the *block-val* argument in a block assignment can specify a location within an overlay which contains the memory address of the structure memory block. The reason for introducing the indirect addressing mode is so that the execution of the abstract machine can be more efficient by not always requiring structure memory read and write instructions to copy data back and forth between a structure memory object and locations accessible by a super-actor's instructions. The question now is: why do we bother having the SMread and SMwrite instructions at all? The answer lies in a multiprocessor implementation of the SAM. In a multiprocessor SAM, some SM objects will reside in a local PE while others are in remote PEs. Accessing local SM objects can be efficiently performed using the indirect addressing mode since the access latency is only

the local main memory latency time. However, remote SM objects should use the SMread and SMwrite instructions since they will generally take a longer time for execution and should be processed by a dedicated unit which interfaces to the interconnection network¹⁸ (this enhancement is proposed in the advanced abstract machine model; the advantage is that the fast memory pre-loader mechanism in the machine does not have to interface with the interconnection network).

There are now two addressing modes in a block assignment: *local*, and *indirect*. In the *local* addressing mode, *block-val* is an identifier of the overlay block to be pre-loaded; the overlay block is within the overlay of the actor instance which requests the load. (The local addressing mode is the original and only addressing mode of block assignments as previously described.) In the indirect addressing mode, *block-val* is a pointer to a location in the requesting actor instance's overlay which contains a data memory address (address of a data block—structure memory block or overlay block which may belong to another function instance).¹⁹

In an illustration of a super-actor graph, the indirect addressing mode in a block assignment is indicated if *block-val* has the form '@*block-id.offset*'. Later, the function *block-addr* presents the block addressing modes in the context of the intermediate abstract machine model.

An Indirect Addressing Mode for Instructions Currently, the operand/result fields of an instruction in a super-actor have three addressing modes: immediate, register, and slot pointer. In the slot pointer mode, the offset is fixed at compile time, and if some applications require the ability to access certain data depending on the input, then the execution of such applications can be inefficient. Thus, we now introduce an indirect addressing mode for operand/results field so an instruction can access *any* element within a block of data

¹⁸The same line of reasoning applies to why the 'send' instruction is still used for function application linkage.

¹⁹Note that the possibility of indirectly addressing an overlay block provides another means for actors in different function instances to communicate with each other. This can result in efficient function invocations when both the caller and callee are known to reside on the same PE. That is, the copying of arguments to the callee's overlay can be avoided, only a pointer need be passed.

which was pre-loaded into fast buffer memory. The format of an operand or result field utilizing the indirect addressing mode is: '*Bxx.Ry*' where '*Bxx*' is used as an index into an actor instance's assigned register set which stores pointers to data blocks in the fast buffer memory, and '*Ry*' indicates the *y*th temporary register containing the offset into that data block.

Additions to an Actor's Components

The attributes of an actor are now augmented with a block assignments component:

$\langle \text{attributes} \rangle ::= \langle \text{block-assgt} \rangle \langle \text{instr-list} \rangle \langle \text{reset-count} \rangle \langle \text{init-count} \rangle$
 $\langle \text{signal-lists} \rangle$

$\langle \text{block-assgt} \rangle ::= [\langle \text{b-ptr} \rangle \langle \text{block-val} \rangle]^0$

$\langle \text{b-ptr} \rangle ::= \text{'B' integer}$

$\langle \text{block-val} \rangle ::= \langle \text{block-type} \rangle \langle \text{block-value} \rangle$

$\langle \text{block-type} \rangle ::= \text{'indir' | 'local'}$

$\langle \text{block-value} \rangle ::= \langle \text{overlay-slot} \rangle | \langle \text{block-id} \rangle$

$\langle \text{overlay-slot} \rangle ::= \text{offset from overlay base address indicating the location containing the memory address of the data block to be loaded}$

Below, the *opr-res-type* field is augmented to reflect the indirect addressing mode for operands/results.

$\langle \text{opr-res-type} \rangle ::= \text{'reg' | 'immed' | 'slot-ptr' | 'slot-indir'}$

$\langle \text{opr-res-val} \rangle ::= \text{integer | 'slot-ptr' | 'slot-indir'}$

The *slot-ptr* component in an instruction must be changed to reflect the use of a *Bxx b-ptr*, and a new component *slot-indir* must be added:

$\langle \text{slot-ptr} \rangle ::= \langle \text{b-ptr} \rangle \langle \text{offset} \rangle$

$\langle \text{slot-indir} \rangle ::= \langle \text{b-ptr} \rangle \langle \text{reg-name} \rangle$

$\langle \text{reg-name} \rangle ::= \text{integer}$

Since register sets (see next section) are assigned to an actor instance to store pointers to the fast buffer memory, the *dynamic-attributes* component of an actor instance is now augmented with a *fb-ptrs* component:

$\langle \text{dynamic-attributes} \rangle ::= \langle \text{enable-count} \rangle \langle \text{running-status} \rangle \langle \text{cond-code} \rangle$
 $\langle \text{fb-ptrs} \rangle$

$\langle \text{fb-ptrs} \rangle ::= \text{list of fast buffer memory block pointers}$

Elements in a *fb-ptrs* list can be accessed by using a *b-ptr* value as an index. The other components of dynamic-attributes are still stored in an overlay of the data memory.

Additions to the Memory Model and Related Symbols

Since block assignments are added to an actor's syntax, we use the symbol $AP.BA.i$ to point to a location in the actor attribute memory containing the i th block assignment of actor AP . The symbol $AP.N_{BA}$ locates the number of block assignments of actor AP .

In the memory model for the intermediate abstract machine model, we represent the *fast buffer memory* with the symbol: $FB[]$. The fast buffer memory is divided into blocks and it is assumed that the size of a block is equal to the largest data block (an overlay or structure memory block) required. Each block in $FB[]$ is identified with a unique label Li where $FB[Li]$ is the location of the first element in block Li . The number of blocks in $FB[]$ is greater than or equal to the greatest number of blocks a super-actor instance requests during its pre-loading of fast buffer memory phase (when an enabled actor instance becomes ready). This ensures that the readying agent (described below) does not deadlock when it tries to prepare the fast memory for the execution of an enabled super-actor instance.

The *fb-ptrs* of an active actor instance are stored in a set of registers called *pointer registers* ($PR\{ \} []$). A set of pointer registers is assigned to each enabled actor instance when it becomes ready, i.e., during the fast buffer pre-loading phase of an actor instance. For example, the expression $PR\{OP, AP\}[yy]$ represents the pointer register set assigned to actor instance $\langle OP, AP \rangle$. The value in the square brackets (*yy*) can be 'I' for indicating that the register contains a pointer to a block in $FB[]$ which contains the instructions of the actor, or *yy* can be a *b-ptr* '*Bxx*' for indicating that the register contains a pointer to a fast buffer memory block which contains data block *Bxx* of the actor instance.

In the following table, we show the added symbols to various information in memory units. We remind the readers that the other symbols are found on page 85.

Symbol	Points to (locations in actor attribute memory $AA[]$):
$AP.BA.i$	<i>i</i> th block assignment of actor <i>AP</i>
$AP.N_{BA}$	no. of block assignments of actor <i>AP</i>
Points to (locations in pointer registers $PR\{OP, AP\}[]$):	
'IP'	a pointer value locating a fast buffer memory block containing instructions of actor <i>AP</i> (actually, points to first instruction of actor <i>AP</i>)
'B' <i>xx</i>	a pointer value locating a fast buffer memory block containing the <i>xx</i> th data block of actor instance $\langle OP, AP \rangle$ (actually, pointer value points to first data value within the <i>xx</i> th data block)

The Intermediate Abstract Machine Model

The intermediate abstract machine model is shown in figure 5.15, and the associated memory model appears in figure 5.16.

The actions of the deactivation-enabling, readying, and activation agents are listed below.

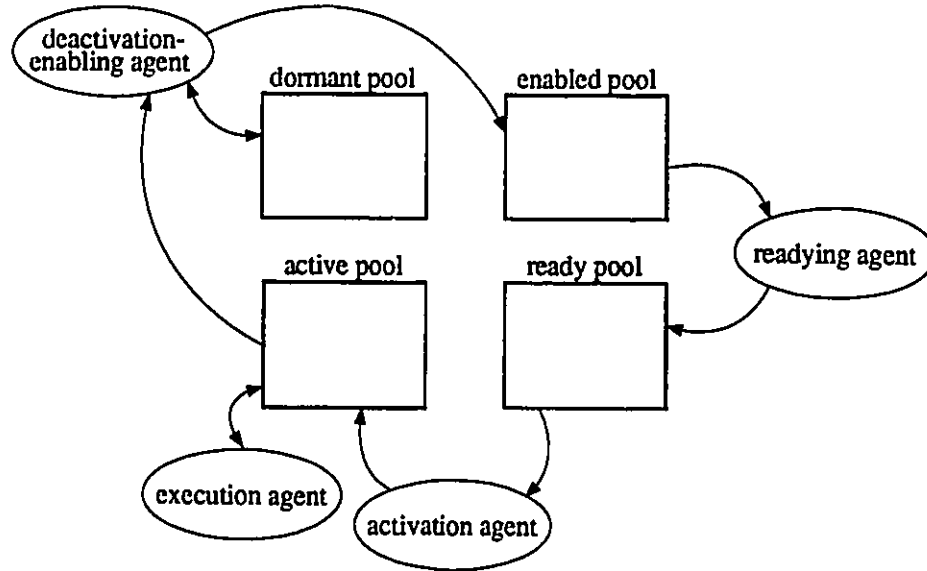


Figure 5.15: The intermediate abstract machine model of the SAM.

```

procedure deactivate-enable () /* deactivation-enabling agent */
  do (forever)
    from Active Pool, pick  $\langle OP, AP \rangle$  with  $D[OP + AA[AP.rs]] = \text{'terminated'}$ 
    deallocate assigned resources, copy data blocks
    back to main memory and free those data blocks
    do ( $i := 0$  to  $(AA[AP.N_{su}] - 1)$ )
      decrement-reset ( $OP, AA[AP.SL_u + i]$ )
    enddo
    if ( $D[OP + AA[AP.cc]] = \text{'true'}$ ) then
      do ( $i := 0$  to  $(AA[AP.N_{st}] - 1)$ )
        decrement-reset ( $OP, AA[AP.SL_t + i]$ )
      enddo
    if ( $D[OP + AA[AP.cc]] = \text{'false'}$ ) then
      do ( $i := 0$  to  $(AA[AP.N_{sf}] - 1)$ )
        decrement-reset ( $OP, AA[AP.SL_f + i]$ )
      enddo
    enddo

```

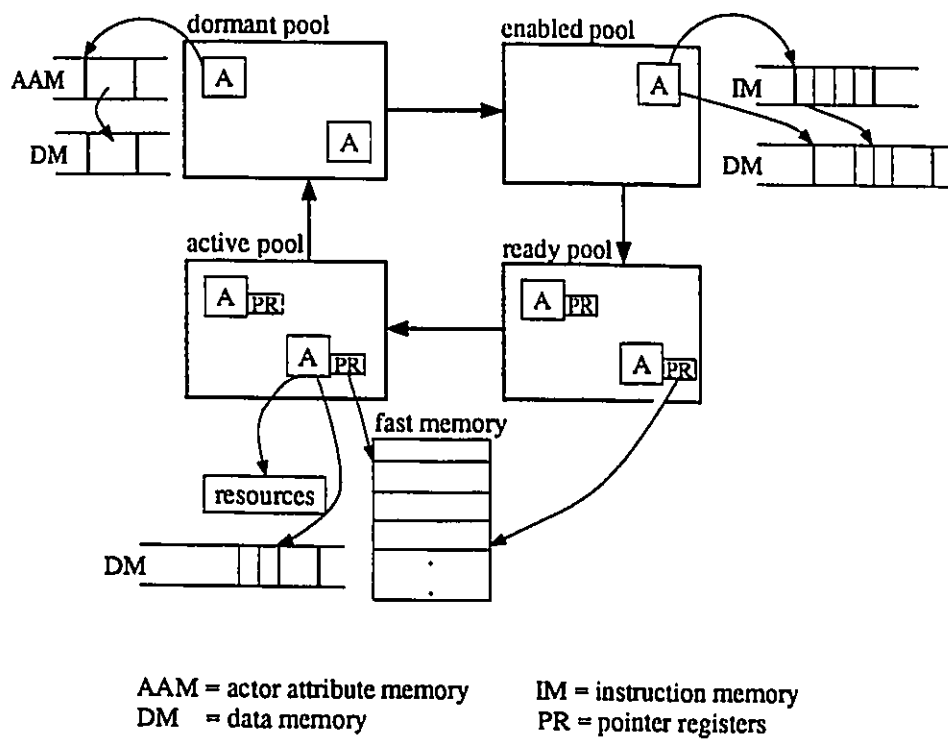


Figure 5.16: The intermediate memory model of the SAM.

The deactivation-enabling agent performs the tasks of the activate and deactivate agents in the base abstract model. That is, it removes terminated actor instances from the active pool and signals the actor instances which require notification. The decrement-reset procedure is responsible for decrementing an actor instance's enable count and if it become zero, it is reset to the actor's reset value and that actor instance is put into the enabled pool.

```

procedure decrement-reset ( $OP, AP'$ )
  decrement  $D[OP + AA[AP'.ec]]$ 
  if ( $D[OP + AA[AP'.ec]] = 0$ ) then
     $D[OP + AA[AP'.ec]] := AA[AP'.C_r]$ 
    put  $\langle OP, AP' \rangle$  into the Enabled Pool

```

The readying agent first checks to see if there are any free blocks in the the fast buffer memory before it can proceed to process an enabled actor instance. Once there is space, the actor's block of instructions are fetched from instruction memory and are stored in the fast buffer memory. A pointer to the assigned $FB[]$ block is stored in the 'I' register of the assigned pointer register set. Then the required data blocks of the actor instance are loaded and the assigned pointer registers updated. Lastly, the actor instance is put into the ready pool.

```

procedure ready () /* readying agent */
  do (forever)
    from Enabled Pool, pick  $\langle OP, AP \rangle$ 
    if (no free block in  $FB[ ]$ ) then wait
    /* fetch instructions of enabled actor instance into fast memory */
     $ptr :=$  label of a free block
    assign a set of pointer registers to  $\langle OP, AP \rangle$ 
    fetch  $I[AA[AP.I]]$  to  $I[AA[AP.I] + AA[AP.N_I] - 1]$  and
      put into  $FB[ptr]$ 
     $PR\{OP, AP\}[IP] := ptr$ 
    /* now fetch data blocks into fast memory */
    do ( $i := 1$  to  $AA[AP.N_{BA}]$ )
      if (no free block in  $FB[ ]$ ) then wait
       $ptr :=$  label of a free block

```

```

        fetch data block at  $D[\text{block-addr}(AA[AP.BA.i])]$  and
        put into  $FB[ptr]$ 
         $PR\{OP, AP\}[AA[AP.BA.i.b-ptr]] := ptr$ 
    enddo
    put  $\langle OP, AP \rangle$  into Ready Pool
enddo

```

In the above procedure and the following function, components of $AP.BA.i$ are accessed with the token names as defined in $\langle \text{block-assgt} \rangle$ (see page 101).

```

function block-addr(b-assgt)
    case (b-assgt.block-info.block-type)
        indir:  $D[OP + b\text{-assgt.block-info.block-value}]$ 
        local:  $OP + b\text{-assgt.block-info.block-value}$ 
    endcase

```

In procedure *deactivate-enable*, the data blocks are copied back to data memory when an actor execution is terminated and in procedure *ready*, enabled actors get their data blocks fetched from data memory. All this copying may lead to significant and excessive memory bandwidth requirements. This is why we introduce a novel architectural concept called the *register-cache* to address this issue. This will be further discussed in chapter 6.

```

procedure activate ()
    do forever
        if (resources available) then
            from ready pool, pick  $\langle OP, AP \rangle$ 
            allocate resources to  $\langle OP, AP \rangle$ 
            put  $\langle OP, AP \rangle$  into activate pool
        enddo
    enddo

```

The actions of the *execution* agent are basically the same as the one in the basic abstract machine model, except that the parameters of the *perform* call are now $(OP, AP, FB[PR\{OP, AP\}[IP] + cnt], AA[AP.N_I], cnt)$.

The instruction set of this model is also the same except that the operand/result field has an added mode, as indicated by macro *:

```
macro * (pxx)
  case (pxx.opr-res-type)
    reg:      reg[pxx.opr-res-val]
    immed:    pxx.opr-res-val
    slot-ptr:  FB[PR{OP,AP}[pxx.opr-res-val.b-ptr]
               + pxx.opr-res-val.offset]
    slot-indir: FB[PR{OP,AP}[pxx.opr-res-val.b-ptr]
                  + reg[pxx.opr-res-val.reg-name]]
  endcase
```

where the mode *slot-indir* is for indirect addressing within a data block (see Indirect Addressing Mode for Instructions on page 100). Note that all operand and result accesses do not have to access slower data memory.

A problem with indirect addressing by instructions is the possibility of a value in register [px.opr-res-val.reg-name] not being within bounds of a data block. In the next chapter, we show how the register value can be enforced to be within bounds by the hardware.

5.2.5 The Advanced Model of the SAM

To further enhance the execution efficiency of the Super-Actor Machine model, we introduce two types of super-actors: sequential super-actors, and parallel super-actors. In a *sequential super-actor*, the data dependencies between instructions require that they be sequentially executed. Note that switch super-actors are classified as sequential super-actors. The second type of super-actor, called a *parallel super-actor*, is a special case of sequential super-actors where the instructions are data independent, i.e., instructions within a parallel super-actor do not depend on any results produced by any other instruction within the super-actor. Thus, instructions in parallel super-actors can be executed in parallel. An example of a parallel super-actor is shown in figure 5.6(c). Actor 1 can be classified as a parallel super-actor since there are no dependencies between the instructions. To differentiate parallel and sequential

super-actors in a SA graph, we label a parallel super-actor with *PSA*. Super-actors without a label are sequential.

In the intermediate abstract machine model, we note that structure memory operations, overlay management instructions, and the send instruction can access slower main memory when they are being processed (in a multiprocessing implementation, they can access remote memory). Processing such instructions along with other instructions which only access fast memory (registers and fast buffer memory) within the same execution unit can adversely affect the throughput rate of the execution unit. Thus we propose that instructions which access main memory during their execution be separated from other instructions which do not, i.e., they are grouped separately into their own actors. These instructions which can have long and unpredictable latencies are called *long-latency instructions* and are excluded from ordinary (sequential and parallel) super-actors. Long-latency instructions are: **SMalloc**, **SMdealloc**, **SMwrite**, **SMread**, **send**, **Oalloc**, and **Odealloc**. An actor containing a long-latency instruction is called a *long-latency actor* ('L-actor' for short) and is labeled *LA* in a super-actor graph. Generally, an L-actor contains only one long-latency instruction, although multiple instructions within an L-actor are allowed, e.g., the 'apply' and 'return' actors as shown in figure 5.9. By introducing a type field for actors, actors can be separated such that a dedicated unit can handle L-actors and the main execution unit can handle ordinary super-actors which generally have short and predictable completion times (the short execution times are aided by the fact that all memory accesses by the instructions are to fast memory).

Furthermore, instructions which modify memory addresses for indirect block addressing (values in locations pointed to by *overlay-slot*, see page 101), should be grouped separately into aggregates called *support-actors*.²⁰ In a super-actor graph, they appear with an *SPPTA* label. The separation of these instructions from super-actors is implementation specific and will be explained when we discuss the proposed architectural implementation of the SAM.

Pre-processing L-actors by pre-loading a fast memory with their operands and results is probably not worthwhile since such actors access slower main memory. Accesses to main

²⁰In chapter 7, we show how support actors can be generated in SA graphs.

memory generally dictate the actor's execution time. Thus, we propose that no pre-loading of fast memory be performed for L-actors and that *block assignments* not be specified in the syntax of L-actors. Pre-processing of support-actors will also not be performed; the reason is solely an economic one. Thus, block assignments are also not part of the support-actor's syntax.

Note that although L-actors and support-actors are differentiated in the advanced abstract machine model, they are still regarded as super-actors in the super-actor execution model.

The Model

We now describe the abstract machine model of the SAM which closely resembles the advocated implementation. This model is represented by five pools containing actor instances in their various states (fig. 5.17): the Dormant Actor Pool, the Enabled Actor Pool, the Ready Super-Actor (SA) Pool, the Active Super-Actor Pool and the Active Other-Actor (OA) Pool. The major difference between this model and the intermediate one is the addition of the other-actor activation and execution agents for handling long-latency and support actors. The associated memory model is sketched out in figure 5.18. As shown in this model, many heterogeneous actions can occur simultaneously.

Operation of this Model

Starting the execution of a super-actor program on this machine model is similar to the routine employed in the basic abstract machine model. The following description applies to the machine model once the agents have been started.

When an actor instance in the Dormant Actor Pool has received all its signals (this is the enabling function of the *deactivation-enabling* agent), the actor instance's enable count will be reset and the actor instance moved to the Enabled Actor Pool. In the process, the type of the actor, the pointer to its memory in the *IM* and the pointer to its overlay in *DM* fetched. Once in the Enabled Actor Pool, the *SA-readying* agent will put super-actor

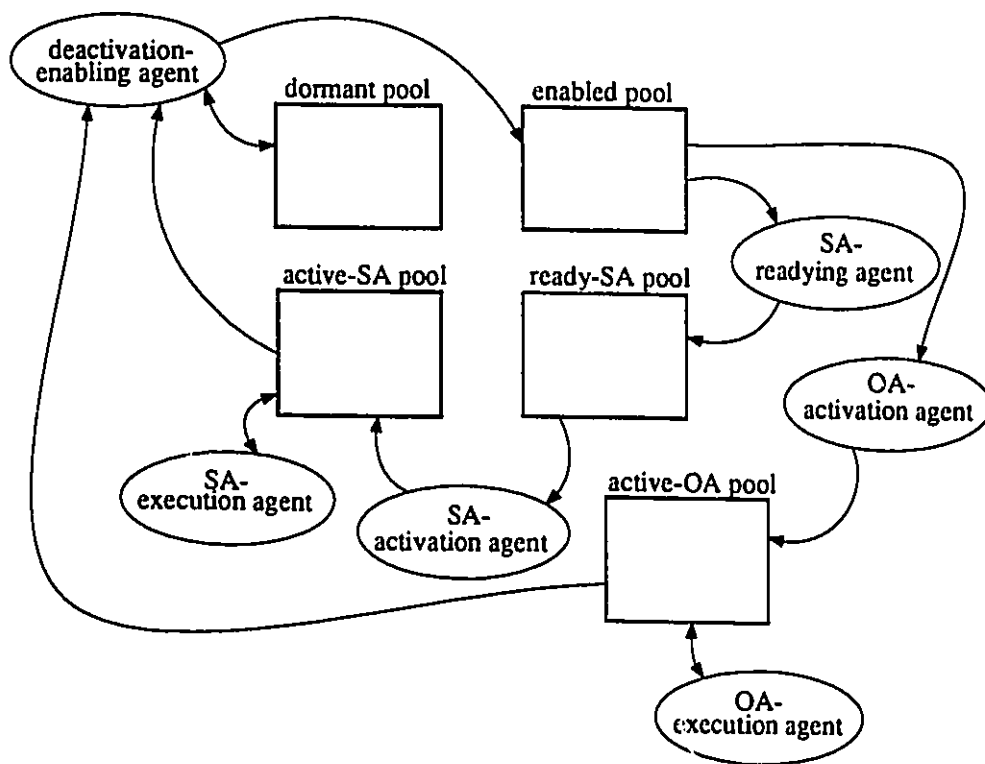


Figure 5.17: The abstract machine model of the SAM.

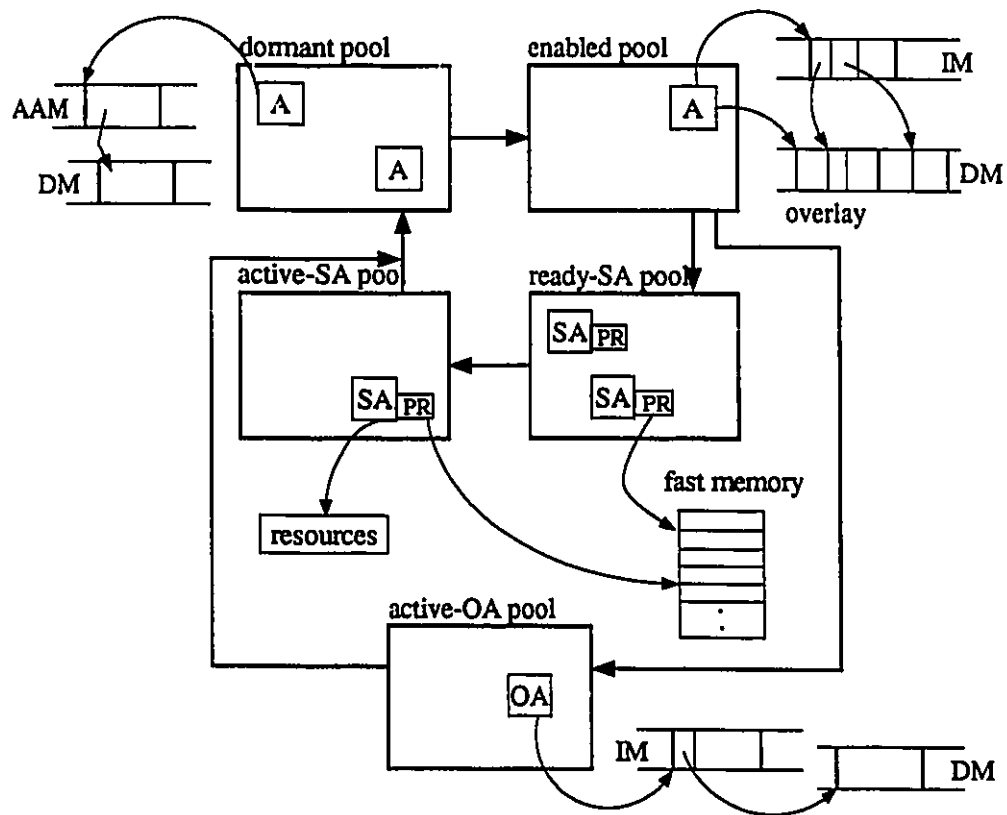


Figure 5.13: The memory model of the abstract machine.

instances into the Ready Super-Actor Pool by fetching the instructions and data from *IM* and *DM* and putting them in the fast memory. Also, a set of pointer registers is assigned to each enabled super-actor instance and the necessary pointers stored. The *OA-activation* agent would put actors other than super-actors (i.e., long-latency actors and support actors) into the Active Other-Actor Pool where they can be processed by the *OA-execution* agent. Once an actor instance has been processed by the *OA-execution* agent, it will be labeled as 'terminated' and the *deactivation-enabling* agent will fetch the corresponding signal list and update the memory in *DM* (the enable counts) of the actor instances which require notification. Finally, the actor instance will be put back into the Dormant Actor Pool. As for super-actor instances in the Ready Super-Actor Pool, the *SA-activation* agent is responsible for obtaining the necessary resources (e.g., a register set, a program counter, etc.) for executing the super-actor instance. Once they are obtained, the super-actor instance is put in the Active-SA Pool. The *SA-execution* agent is responsible for processing the super-actor instances where all required resources and data for processing are local to the agent. Once the actor instance is processed, the resources are deallocated, the super-actor instance labeled 'terminated' and the deactivation-enabling agent takes over. The deactivation-enabling agent treats terminated super-actors the same way it treats terminated L-actors and support-actors.

To formally define the agents, we describe their actions in a similar format as those used in the former two models. Since they are simple extensions of functions performed by agents in the previous two machine models, we will list them in appendix B.

5.3 Summary

In this chapter, we have outlined a new execution model called the *Super-Actor Execution Model* which uses a *super-actor graph* as the program format for the abstract SAM model. We have also formally described the abstract machine models which represent the Super-Actor Machine. The abstract machine model is defined in a manner where proposed features are isolated so that the reader may quickly comprehend the functioning of the Super-Actor Machine.

A super-actor graph is a variant of dataflow graphs where the nodes are aggregations of one or more simple actors, and the arcs between the nodes indicate the signal flow rather than the flow of data. In the past, other program representations like Sarkar's Program Dependence Graph [99], and the Program Dependence Web of Ballance, Maccabe and Ottenstein [18] have been proposed. Since those program representations can generate dataflow graphs, it could be possible to generate SA graphs given the other program representations (we will outline techniques to generate SA graphs from dataflow graphs in chapter 7). The novelty in this work is in the way the nodes get executed, that is, they are executed atomically where all required data must be logically produced and physically residing "close" to the execution mechanism. Once activated, the super-actor executes till termination. The *Macro Dataflow Model* of Sarkar[99] also stipulates that aggregates in his macro dataflow nodes (super-actors) be executed to completion once activated. However, his work is focused only on the compiling aspects and did not propose any architectural model.

With the super-actor execution model, a novel scheme where inherent parallelism of an application can be used to hide the local memory latencies—latencies for fetching data from local main memory to fast buffer memory close to the execution unit—from the execution unit has been introduced. In the next chapter, we will describe one possible implementation of the Super-Actor Machine and the device for tolerating local memory latencies.

Chapter 6

The Architecture of the Super-Actor Machine

The Super-Actor Machine is to be a multi-processor system consisting of multiple processing elements (PEs) linked together by some interconnection network (fig. 6.1—the ICU is the interprocessor communications unit). Memories are distributed to each processor in the machine, and the aggregation of these memories presents a global address space which is shared among all processors; thus, there is no centralized global memory subsystem. In this chapter, we will concentrate our discussions on one processing element and assume that the interconnection network can be a multi-stage interconnect (a tutorial on interconnection networks can be found in [120]). The interprocessor communications are handled by the Inter-PE Communications Unit (ICU).

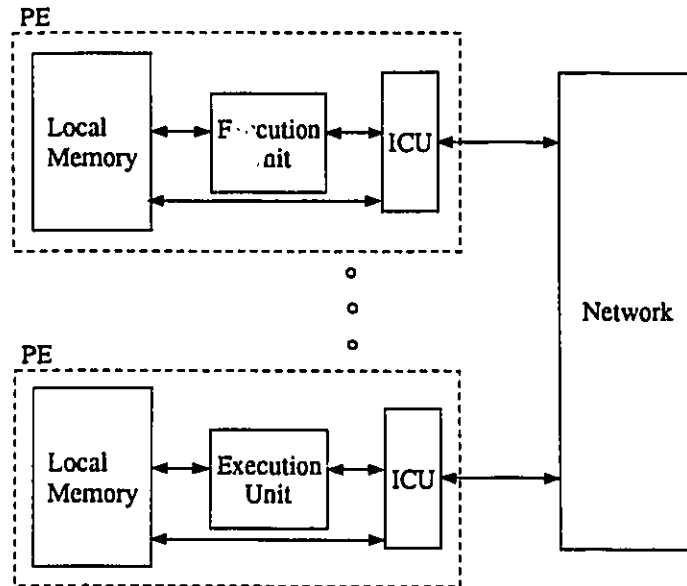


Figure 6.1: The Super-Actor Machine.

6.1 Mechanisms Needed for Super-Actor Processing

To support the execution of a parallel super-actor¹, where its instructions can be entered into an execution pipeline at each pipe beat² without fear of data hazards (the hazard of multiple instructions in the execution pipe which are modifying and accessing the same data concurrently), the machine will require:

- a counter to sequence through the instructions of a super-actor, and
- a high-speed memory which can be pre-loaded with the necessary data so that instructions of parallel super-actors can be issued every pipe beat and proceed through the pipe without impediment.

¹In this chapter, we will use “super-actor” (“actor”) when referring to a super-actor (actor) instance where possible, i.e., when there is no confusion of what we mean.

²Actually, the instructions can be executed in parallel, but since the proposed PE architecture has only one execution pipe for super-actors, we will restrict our discussions to this configuration.

For a sequential super-actor, where an instruction can only enter the execution pipe after knowing that the previous instruction of the super-actor has been executed, the machine will need:

- a counter as mentioned for the execution of parallel super-actors;
- a sequencing mechanism which can issue instructions from multiple active sequential super-actors so that their execution can be interleaved with other sequential/parallel super-actors for maximal utilization of the execution pipe; and
- for each active sequential super-actor in the execution pipe, a register set for storing temporary results and a condition code is required. Thus, multiple register sets are necessary to hold distinct sets of temporary values for several active sequential super-actors.

The main purpose of the sequencing mechanism is to keep a smooth flow of operations in the execution pipe. This is done by “micro context-switching” among active sequential/parallel super-actors. The register-cache (see below) and the multiple register sets will also help in smoothing the instruction flow through the pipe by keeping every memory access to a *low* and *fixed* latency.

As was previously mentioned, long-latency actors have different processing requirements than those of sequential and parallel super-actors (see section 5.2.5), thus a unit for processing only long-latency actors will be incorporated into the machine.

To synchronize all of the different types of actors, a dedicated synchronizer and scheduling unit for handling the signals between actors is required. The reason is that super-actors with one instruction can have a high synchronization requirement to computation ratio. Moreover, performing fine-grain dataflow-like synchronizations can effectively hide the latencies associated with inter-PE operations.

From the requirements specified above, we arrived at a design for the Super-Actor Machine which is based on the McGill Dynamic Dataflow Architecture[47, 71]. The separation of the execution unit from the scheduling unit permitted the easy addition of mechanisms such as the register-cache pre-loader, sequencer, register sets, and the register-cache.

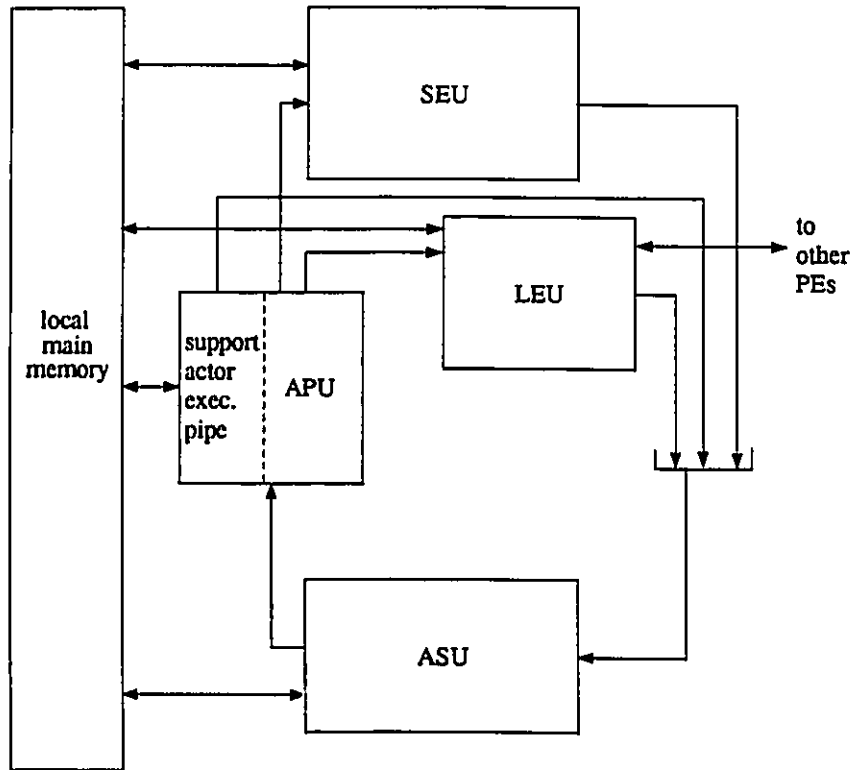


Figure 6.2: A processing element of the Super-Actor Machine.

6.2 A Processing Element of the Super-Actor Machine

A processing element of the Super-Actor Machine has five processing units: the Super-actor Execution Unit (SEU), the Actor Preparation Unit (APU) which has an adjoining support-actor execution pipe, the Actor Scheduling Unit (ASU), the L-actor Execution Unit (LEU), and the local main memory (see fig. 6.2).

The local main memory contains the overlays of function instances which have been assigned to the PE, structure memory objects, and the program code, i.e., all instructions and data for the PE are stored in this local main memory. A *program segment* contains the program code consisting of local constants, function invocation information, actor attributes,

instructions of actors, and the corresponding signal lists. A *data segment* containing a function overlay consists of information for returning values to the caller function, slots (memory locations) for the operands and results, and the enable count values for the corresponding actors. A data segment can also be used to contain structure memory objects instead of function overlays. (We will say more about the local main memory in section 6.2.7.)

The SEU processes *ready* sequential and parallel super-actors and is capable of interleaving instructions from active super-actors for maximal utilization of the execution pipe. When a super-actor has been terminated, a done signal is emitted to the ASU.

The APU utilizes a *register-cache loader* for making an *enabled* super-actor *ready* by ensuring that the requested data and instructions can be found in the register-cache. (The register-cache loader is responsible for processing the block assignments of an enabled super-actor—see section 5.2.4.) For a super-actor to become ready, there is also a short-cut path for super-actors which are labeled as “fast-path candidates” at compile time. The short-cut path eliminates much overhead associated with loading the register-cache with the necessary data when it can be determined that the operands/result space are *most likely already* in the register-cache (this mechanism and the technique to identify the fast-path candidates will be discussed in detail in later sections). The APU is also responsible for routing L-actors to the LEU and support-actors to an attached RISC pipeline. The reasons for introducing a separate execution unit for processing support-actors are two-fold: the first is that the results of support-actors will be accessed by the register-cache loader in the APU, so introducing the attached RISC pipeline to generate and deposit the support-actor results into a separate data cache can decrease the memory bandwidth demands on the register-cache. (Otherwise, the SEU would execute these support-actors and place those results into the register-cache. Now, the register-cache loader simply accesses the data cache of the support-actor execution pipe.) The second is that the additional execution pipe further increases the instruction-level parallelism the architecture can exploit.

The LEU processes long-latency actors and contains the Inter-PE Communications Unit (ICU) for sending and receiving data from the interconnection network.

The purpose of the ASU is to process done signals for super-actors, L-actors and support-actors. In processing the done signals, the ASU identifies other actors which become enabled, and sends these enabled actors to the APU.

Before we describe the processing units in detail, we will define the tuples as used in the algorithms which describe the processing units' functions. Then, the register-cache mechanism will be detailed since it is at the heart of the Super-Actor Machine. Afterwards, the ASU, the APU and the attached support-actor execution unit, the LEU, the SEU, and local main memory (including the memory map) will be described in the following sections.

6.2.1 Tuple Definitions

The reader may skip this section and continue with the next section on page 124. However, the reader will have to refer to this section if he/she wants to understand the full details of the algorithms.

The list of tuples and their components is in alphabetical order. *Tuples* contain two or more components, e.g., $\langle xx, yy, zz \rangle$ is a tuple and is equivalent to $\langle xx \rangle \langle yy \rangle \langle zz \rangle$. Expressions of the form $[\langle xx \rangle]^1$ indicate one or more tuples of the form $\langle xx \rangle$.

$\langle F\text{-off} \rangle ::=$ offset from $\langle \text{sig-list-ptr} \rangle$ locating the false signal list

$\langle F\text{-len} \rangle ::=$ length of false signal list

$\langle La\text{-info} \rangle ::= \langle \text{instr-ptr}, \text{length} \rangle$

$\langle T\text{-off} \rangle ::=$ offset from $\langle \text{sig-list-ptr} \rangle$ locating the true signal list

$\langle T\text{-len} \rangle ::=$ length of true signal list

$\langle R\text{-cache-tag} \rangle ::= \langle \text{mem-block-addr}, \text{reserve-count}, \text{age-count} \rangle$

$\langle U\text{-off} \rangle ::=$ offset from $\langle \text{sig-list-ptr} \rangle$ locating the unconditional signal list

$\langle \text{U-len} \rangle ::= \text{length of unconditional signal list}$
 $\langle \text{active-flag} \rangle ::= \text{'T' | 'F'}$
 $\langle \text{actor-attr} \rangle ::= \langle \text{actor-type, sig-list-ptr, actor-info} \rangle$
 $\langle \text{actor-cache-entry} \rangle ::= \langle \text{sig-list-ptr, start-offset, end-offset, instr-line, op-res-line} \rangle$
 $\langle \text{actor-cache-tag} \rangle ::= \langle \text{base-addr, actor-ptr, active-flag, age-count} \rangle$
 $\langle \text{actor-info} \rangle ::= \langle \text{sa-info} \rangle \mid \langle \text{sppta-info} \rangle \mid \langle \text{La-info} \rangle$
 $\langle \text{actor-ptr} \rangle ::= \text{address of actor, } \langle \text{actor-attr} \rangle$
 $\langle \text{actor-type} \rangle ::= \text{'sa' | 'sppta' | 'La'}$
 $\langle \text{age-count} \rangle ::= \text{integer value indicating the age of a cache line}$
 $\langle \text{alu-out} \rangle ::= \langle \text{ctxt-id, sa-type, last-instr, res-value, res, write-to-mem} \rangle$
 $\langle \text{base-addr} \rangle ::= \text{base address of overlay}$
 $\langle \text{cond-code} \rangle ::= \text{'U' | 'T' | 'F'}$
 $\langle \text{count-signal} \rangle ::= \langle \text{base-addr, actor-ptr, enbl-cnt-ptr, dec-value, fast-path-cand} \rangle$
 $\langle \text{ctxt} \rangle ::= \langle \text{base-addr, actor-ptr, sig-list-ptr, sa-type, instr-line, curr-offset, end-offset, op-res-lines, ready-flag, free-flag} \rangle$
 $\langle \text{ctxt-id} \rangle ::= \text{id of context}$
 $\langle \text{curr-offset} \rangle ::= \text{offset from the beginning of the instruction block locating the currently processed instruction}$
 $\langle \text{dec-rsvd-count} \rangle ::= \langle \text{instr-line, op-res-lines} \rangle$
 $\langle \text{dec-value} \rangle ::= \text{decrement value for the enable count}$
 $\langle \text{done-signal} \rangle ::= \langle \text{base-addr, sig-list-ptr, cond-code} \rangle$

<d-R-cache-req> ::= <mem-block-addr, line-directive, line-type>

<enbl-L-actor> ::= <base-addr, sig-list-ptr, instr-ptr, length>

<enbl-actor> ::= <base-addr, actor-ptr, fast-path-cand>

<enbl-cnt-off> ::= offset from <base-addr> which points to the enable count of the actor pointed to by <actor-addr>

<enbl-cnt-ptr> ::= address of the enable count to be decremented

<enbl-sa> ::= <base-addr, actor-ptr, sig-list-ptr, sa-type, instr-info, op-res-info>

<enbl-sppta> ::= <base-addr, sig-list-ptr, instr-ptr, length>

<end-offset> ::= offset value locating the last instruction in the memory block pointed to by mem-block-addr of instr-info

<fast-path-cand> ::= 'Y' | 'N' indicating if the actor is a fast-path candidate or not

<free-flag> ::= 'T' | 'F'

<i> ::= integer

<instruction> ::= <opcode, op1, op2, res>

<instr-info> ::= <line-directive, mem-block-addr, start-offset, end-offset>

<instr-line> ::= instruction R-cache line no.

<instr-no> ::= address of instruction in i-R-cache

<instr-packet> ::= <ctxt-id, instr-no, sa-type, last-instr, op-res-lines>

<instr-ptr> ::= address of first instruction in actor

<j> ::= integer

$\langle \text{last-instr} \rangle ::= \text{'T'} \mid \text{'F'}$

$\langle \text{length} \rangle ::= \text{number of instructions in actor}$

$\langle \text{line-directive} \rangle ::= \text{'load'} \mid \text{'optional'}$

$\langle \text{line-mode} \rangle ::= \text{'direct'} \mid \text{'local'} \mid \text{'indirect'}$

$\langle \text{line-type} \rangle ::= \text{'oprnd'} \mid \text{'res'}$

$\langle \text{line-value} \rangle ::= \langle \text{mem-block-addr} \rangle \mid \langle \text{offset-value} \rangle \mid \langle \text{pointer} \rangle$

$\langle \text{mem-block-addr} \rangle ::= \text{memory block address}$

$\langle \text{no.-op-res-infos} \rangle ::= \text{number of } \langle \text{op-res-info} \rangle \text{'s}$

$\langle \text{op1} \rangle, \langle \text{op2} \rangle ::= \langle \text{operand-mode, operand-value} \rangle$

$\langle \text{offset-value} \rangle ::= \text{local overlay offset locating memory address of overlay block to be loaded}$

$\langle \text{operand-mode} \rangle ::= \text{'reg'} \mid \text{'Rc'} \mid \text{'indir'} \mid \text{'immed'}$

$\langle \text{operand-value} \rangle ::= i \mid i.j$

$\langle \text{op-res-info} \rangle ::= [\langle \text{line-directive, line-mode, line-type, line-value} \rangle]^1$

$\langle \text{op-res-lines} \rangle ::= \text{list of data R-cache line nos.}$

$\langle \text{pointer} \rangle ::= \text{pointer locating memory address of block to be loaded}$

$\langle \text{ready-flag} \rangle ::= \text{'T'} \mid \text{'F'}$

$\langle \text{ready-sa} \rangle ::= \langle \text{base-addr, actor-ptr, sig-list-ptr, sa-type, instr-line, start-offset, end-offset, op-res-lines} \rangle$

$\langle \text{res} \rangle ::= \langle \text{result-mode, result-loc} \rangle$

$\langle \text{reserve-count} \rangle ::= \text{integer value indicating how many super-actors have}$

reserved the line

$\langle \text{reserve-lines} \rangle ::= \langle \text{op-res-lines} \rangle$
 $\langle \text{result-mode} \rangle ::= \text{'reg'} \mid \text{'Rc'} \mid \text{'indir'}$
 $\langle \text{result-loc} \rangle ::= i \mid i.j$
 $\langle \text{res-packet} \rangle ::= \langle \text{ctxt-id}, \text{sa-type}, \text{last-instr}, \text{res-value} \rangle$
 $\langle \text{res-value} \rangle ::= \text{value of result as generated by an instruction}$
 $\langle \text{sa-info} \rangle ::= \langle \text{sa-type}, \text{instr-info}, \text{op-res-info} \rangle$
 $\langle \text{sa-type} \rangle ::= \text{'seq'} \mid \text{'par'}$
 $\langle \text{sig-list-entry} \rangle ::= \langle \text{actor-ptr}, \text{enbl-cnt-off}, \text{dec-value}, \text{fast-path-cand} \rangle$
 $\langle \text{sig-list-key} \rangle ::= \langle \text{U-off}, \text{U-len} \rangle$
 $\quad \mid \langle \text{U-off}, \text{U-len}, \text{T-off}, \text{T-len}, \text{F-off}, \text{F-len} \rangle$
 $\langle \text{sig-list-ptr} \rangle ::= \text{address of } \langle \text{sig-list-key} \rangle$
 $\langle \text{sppta-info} \rangle ::= \langle \text{instr-ptr}, \text{length} \rangle$
 $\langle \text{start-offset} \rangle ::= \text{offset value locating the first instruction}$
 $\quad \text{in the memory block pointed to by mem-block-addr of instr-info}$
 $\langle \text{write-to-mem} \rangle ::= \text{'sppta-d-cache'} \mid \text{'LEU-d-cache'} \mid \text{'nil'}$

6.2.2 The Register-Cache Architecture

In the previous chapter, we described an algorithm for making an enabled super-actor ready by loading its necessary data into fast memory and then storing the memory blocks back to main memory when a super-actor is terminated (see section 5.2.4). In this section, we

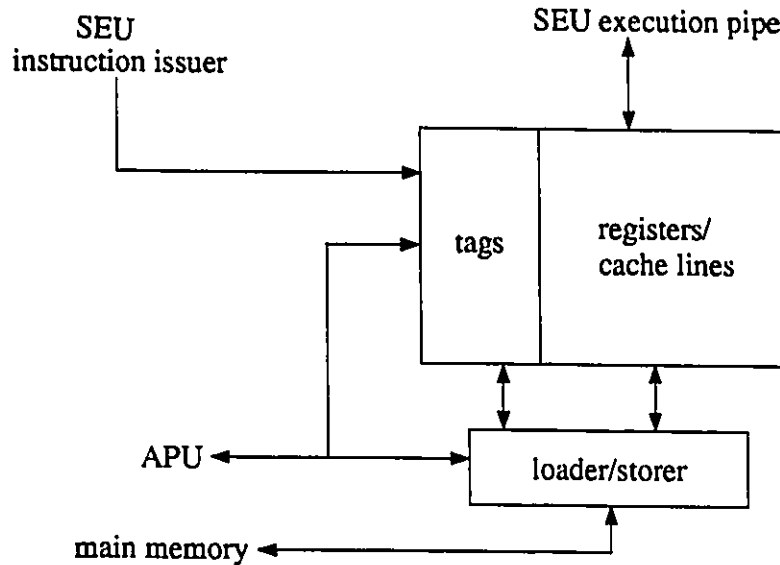


Figure 6.3: The Register-Cache.

introduce the register-cache mechanism which attempts to decrease the traffic of copying data back and forth to main memory.

A register-cache (*R-cache* for short) is organized both as a register file and a cache. Viewed from the SEU, its contents are directly accessible using relatively short addresses; a process similar to the addressing of general registers in conventional processors. The SEU interfaces with two R-caches, an instruction R-cache (*i-R-cache*) and a data R-cache (*d-R-cache*). (The reason for having separate instruction and data R-caches is so that instructions and data do not have to compete for R-cache space. Moreover, it provides an increased memory bandwidth for the SEU.) From the APU's perspective, though, an R-cache is content addressable, i.e., its contents are tagged just as in conventional caches (fig. 6.3). Each R-cache line is tagged with the information as indicated in *R-cache-tag*. To make effective use of all R-cache lines, the APU sees a fully-associative cache. The associativity of the R-cache is important in determining the maximum number of active and ready super-actors a PE can support (described later in section 6.2.4). The R-cache can also have a set-associative organization, however, a small associativity factor will limit the

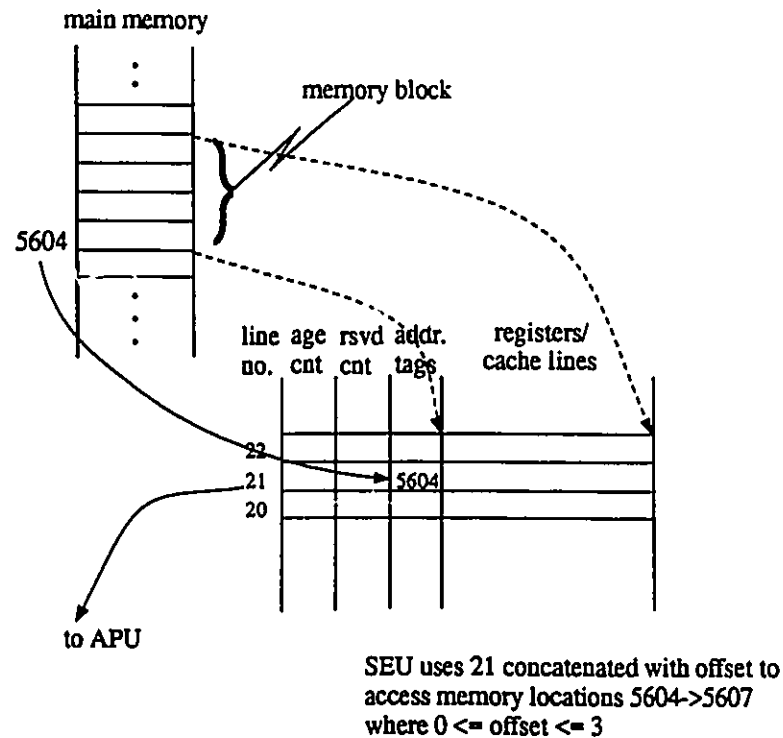


Figure 6.4: The registering process.

number of *live* super-actors (active and ready super-actors).

The binding of a block of memory to an R-cache line is done at run time and is performed using cache update and replacement algorithms which operate on the *reserve-counts* and *age-counts* in the tag section. Once this is done, the R-cache locations within a line can be directly accessed by the SEU using short addresses, just as if they were general registers (i.e., the full address of a local main memory location is not used when the SEU addresses the R-cache). The short addresses are formed by concatenating a compiler generated offset in the code with the line number of the assigned R-cache line. This binding of a memory block to a particular line in the R-cache is called *registering* and is shown in figure 6.4.

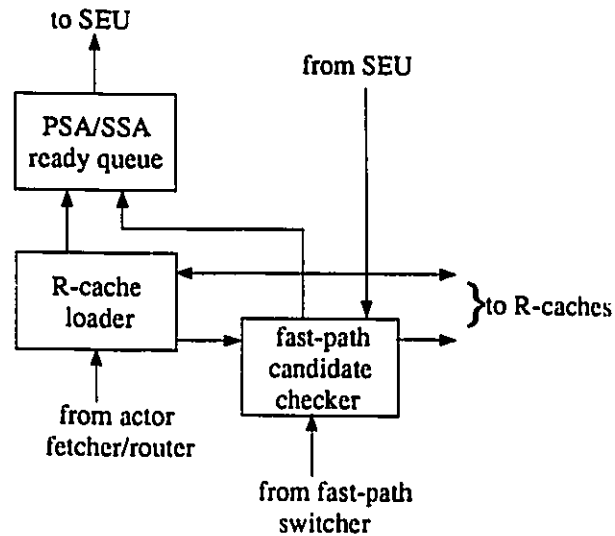


Figure 6.5: APU components which interface with the R-caches.

The Check-In Process for the R-Caches

In this section, the R-cache loader of the APU will be detailed along with the “Check-in” process of a memory block. The rest of the mechanisms in the APU will be described later in section 6.2.4.

The R-cache loader receives an enabled super-actor tuple, *enbl-sa*, and is responsible for *checking-in* the enabled super-actor, i.e., ensuring that all the necessary data for the operation of the super-actor is in the R-cache and that space is reserved in the R-cache for its results. The *enbl-sa* tuple consists of a base address (*base-addr*), the enabled actor’s id (*actor-ptr*), a pointer to the actor’s signal list (*sig-list-ptr*), the super-actor’s type (*sa-type*) and information regarding its block of instructions and operand/result blocks (*instr-info*, *no.-op-res-infos*, and *op-res-info*). When an *enbl-sa* tuple is presented to the R-cache loader, the PSA/SSA ready queue (see figure 6.5, a complete diagram of the APU appears in figure 6.7) is checked to see if it is not full. The number of slots in the ready queue is related to the number of active and ready super-actors the R-cache can contain; thus

checking the ready queue is equivalent to checking for “is memory available in R-cache?” (the size of the ready queue will be discussed in a later section). Once there is space in the ready queue, information for the operand/result lines is sent to the data R-cache and the R-cache loader waits for the assigned d-R-cache line numbers.³ Next, the information for the instruction block is sent to the instruction R-cache and a corresponding R-cache line is received.⁴ Lastly, the *ready-sa* tuple is sent to the PSA/SSA ready queue.⁵ The *ready-sa* tuple consists of a base address, the actor’s id, signal list pointer, super-actor type, instruction R-cache line number (*instr-line*), offsets indicating the first and last instructions in the instruction R-cache line (*start-offset* and *end-offset*), and a list of data R-cache line numbers (*op-res-lines*).

The check-in algorithm used by the R-cache loader is listed below.

```

algorithm check-in
  Input <enbl-sa>
  output <ready-sa> to PSA/SSA ready queue and
                    to fast-path candidate checker
  function
    If (PSA/SSA ready queue is full) then wait
    for (i := 1 to no.-op-res-lines) do
      op-res := ith op-res-info
      d-R-cache-req.mem-block-addr :=
        calculate-mem-block-addr(op-res.line-value, op-res.line-mode,
                                base-addr)
      d-R-cache-req.line-directive := op-res.line-directive
      d-R-cache-req.line-type := op-res.line-type
      send <d-R-cache-req> to d-R-cache
      If (R-cache-line-no. not received from d-R-cache) then wait
      append R-cache-line-no. to ready-sa.op-res-lines

```

³The ordering of operand and result lines which appear in the *op-res-info* component of the input *enbl-sa* tuple must be maintained, such that the corresponding R-cache line numbers will also appear in the same order in *op-res-lines* of the output *ready-sa* tuple. We will see why this is necessary when the operation of the SEU is described.

⁴Actually, the interaction between the i-R-cache and the R-cache loader and those between the d-R-cache and the R-cache loader can be done in parallel. The simulator described in chapter 8 does exactly that.

⁵The *ready-sa* tuple is also sent to the *fast-path candidate checker* unit so that pertinent information can be stored. The stored information is detailed when we explain the fast-path mechanism.

```

endfor
send instr-info.mem-block-addr to i-R-cache
if (R-cache-line-no. not received from i-R-cache) then wait
ready-sa.instr-line := R-cache-line-no.
send <ready-sa> to fast-path candidate checker unit
and to PSA/SSA ready queue

```

Notations of the form *xx.yy* refer to the *yy* component of the *xx* tuple. For the sake of clarity, the components of the output tuples which are not explicitly assigned in the algorithm contain values from the appropriate components of the input tuple. Also, whenever it can be unmistakably determined which tuple a component originates from, the tuple name will be omitted for clarity. For example, in the above algorithm, the *instr-info* component in the statement “send *instr-info.mem-block-addr* ...” can only come from the input *enbl-sa* tuple, thus the tuple name was omitted. The following algorithms will also adopt this policy.

```

function calculate-mem-block-addr (line-value, line-mode, base-addr)
  case (line-mode)
    'direct': line-value
    'local': line-value + base-addr
    'indirect': Mem[line-value + base-addr]
  endcase

```

The *calculate-mem-block-addr* function calculates the memory address of the data block to be pre-loaded into the data R-cache and is a translation of the *block-ptr* function on page 107. The *direct* addressing mode was added so that the machine can access locations which are fixed at compile or load time, e.g., values in a constants table.⁶ The *line-mode* of an operand or result line indicates whether *line-value* is a memory block address (*mem-block-addr*), an offset from the base address of the local overlay indicating the overlay block (*offset-value*), or an indirect value (*pointer*) where the memory block address is found in a local overlay

⁶In a super-actor graph, the direct addressing mode of a block assignment is represented by the '#' sign. For example, 'B1 := #const-block' indicates that the data block labeled as *const-block*—a data memory address in the SAM implementation—should be pre-loaded.

slot ($Mem[line-value + base-addr]$).⁷

An analogy of this check-in process can be found at the ticket counter in an airport. Before boarding the airplane (execution unit), the tour group (super-actor) must first receive their boarding passes indicating that a block of seats are reserved (set of cache lines L it requires is in place). The assignment of seats within the block (relative locations of operands and results) to each member (instruction) of the tour group can be done statically by the tour group manager (at compile time). However, the final row numbers are assigned dynamically prior to the departure (the processing of the super-actor) during the check-in time. Two divergences from the analogy are: the SAM architecture can overlap the check-in process of super-actors with the execution of other ready super-actors, and super-actors can share blocks.

Operations of a Register-Cache

In this section, we will describe the functioning of the data register-cache. The functions of the i-R-cache is similar to the d-R-cache except simpler since it is read only and it only processes one type of requests from the R-cache loader.

The inputs to the data R-cache are: read and write requests to explicit locations in the R-cache; a *dec-rsrd-count* signal from the SEU which contains the R-cache line numbers a terminated super-actor had reserved and no longer needs; a *write-to* tuple containing the R-cache line number which is to be copied to a *write-to-cache* (this cache is either the data cache of the support-actor execution pipe or the data cache of the LEU); loads, mandatory loads and reserve signals from the R-cache loader (*d-R-cache-req* tuple); and a list of R-cache lines to be reserved from the fast-path candidate checker. Note that all these inputs from different processing units (SEU, R-cache loader, and fast-path candidate checker) can come to the R-cache simultaneously, but the tag information must be updated

⁷The programmer can always set line-value past the boundaries of the overlay. The only means for ensuring that the program does not perform out-of-bound accesses is to attach a program id to portions of memory which the program can access and to attach the same program id to the actor attribute. Thus, some hardware logic can perform the boundary checking at run time by comparing the program ids.

atomically (for instance, a reserve signal and dec-rsvd-count signal both access the same line, so the reserve count must reflect the situation correctly). To enforce the atomic updates of the count fields, simultaneous tag-checking and counter incrementing/decrementing is done atomically, and simultaneous requests will be serialized by some arbitration logic.

The *dec-rsvd-count* signal from the SEU is used to decrement the *reserve-counts* of R-cache lines. The *write-to* signal is used to enforce cache coherency between the d-R-cache and the LEU and support-actor execution pipe data caches via software control. Mandatory loads are necessary because operand line(s) of a super-actor which were written by an L-actor must be brought into the d-R-cache since the LEU can only write to main memory. Another scenario which requires a mandatory load is when a function is newly created (function applications are handled by the LEU) and the input parameters must be loaded into the d-R-cache in case an old copy of the memory blocks which contain the input parameters—that is, those memory blocks were allocated to a defunct overlay—is still in the d-R-cache. Mandatorily loading those blocks ensures that the initial super-actors of the function will operate on the correct data.

In the d-R-cache, the registering process begins when a memory block address is sent to the R-cache from the APU. Read-in requests are issued for operand lines and reserve requests are issued for result lines. Lines containing both operand and result values would be identified as an operand line at compile time.⁸ The reason why there is a distinction between operand and result lines is so that the d-k-cache does not have to perform memory block loads for result lines. During the registering process, the Least Recently-Used (LRU) cache replacement policy is used on lines which are no longer needed (i.e., lines with reserve-counts equal to zero) to find a replacement line. The LRU algorithm uses the *age-counts* to decide which line to replace, i.e., a line with the maximum age-count value is a candidate for replacement. Note that the *age-count* values are updated only by requests from the APU, not by accesses from the SEU. After the registering process, we say that the instruction is *checked-in*.

The algorithm describing the operation of the data R-cache is listed below.

⁸The compiler must be certain that the producer actor of the line containing operands and results will not be simultaneously active with a consumer actor of that line.

algorithm *d-R-cache*

```

Input read/write request | <write-to>
        | <dec-rsrvd-count> from SEU
        | <d-R-cache-req> from R-cache loader
        | <reserve-lines> from fast-path candidate checker
output data or acknowledgment-signal to SEU
        R-cache-line-no. to ASU
        acknowledgment-signal to fast-path candidate checker

function
    /* process inputs from SEU */
    if (read/write from SEU) then service it
    if (<dec-rsrvd-count> from SEU) then
        decrement reserve-count of the R-cache-line-nos. in the list
    if (<write-to> from SEU) then
        write R-cache line as indicated by R-cache-line-no.
        to data cache as indicated by write-to-cache and
        send acknowledgment-signal to SEU
    /* process input from fast-path candidate checker */
    if (input from fast-path candidate checker) then
        increment reserve-count of the R-cache-line-nos. in the list
        increment age-count of R-cache lines with reserve-count = 0
        send acknowledgment-signal to fast-path candidate checker
    /* the registering process (process input from R-cache loader) */
    if (input from R-cache loader) then
        if (line-type = 'oprnd') then
            case (line-directive)
                'load': load-line()
                'optional': if (line exists) then
                    increment reserve-count of line
                    send R-cache-line-no. to R-cache loader
                else load-line()
            endcase
        else /* line-type = 'res' */
            if (line exists) then
                increment reserve-count of line
                send R-cache-line-no. to R-cache loader
            else reserve-line()
            increment age-count of lines with reserve-count = 0

```

procedure *load-line* ()

use LRU replacement policy on lines with *reserve-count* = 0 and
 write back dirty line if necessary
 read memory block pointed to by *mem-block-addr* into assigned line
 increment *reserve-count* and zero *age-count* of assigned line
 send assigned *R-cache-line-no.* to R-cache loader

procedure *reserve-line* ()

use LRU replacement policy on lines with *reserve-count* = 0 and
 write back dirty line if necessary
 increment *reserve-count* of assigned line
 send *R-cache-line-no.* to R-cache loader

The Size of the Register-Cache

For the tandem of the APU and R-cache (i-R-cache or d-R-cache) to function correctly, i.e., the SEU is guaranteed that an active super-actor will always find its instructions, operands, and result locations in the register-caches, there is a minimum number of required R-cache lines. The minimum number of lines is $(J + K) \times L$ for J ready super-actors and K active super-actors, assuming that the R-cache is fully associative and L is the maximum number of register-cache lines allocated per super-actor.⁹ That is, J is the number of slots in the PSA/SSA ready queue, and K is the maximum number of allowable super-actors in the SEU. To arrive at this number of R-cache lines, we must assume the worst case scenario where no *live* (active or ready) super-actors share R-cache lines. On average, however, many super-actors do share lines, so $(J + K) \times L$ R-cache lines support more than $(J + K)$ live super-actors. The use of reserve-counts in the R-caches guarantees that a register-cache line which is reserved by one or more super-actors will not be replaced until those super-actor(s) which requested it exits the SEU.

⁹For an m -way set-associative R-cache C organization, then $(J + K)$ must be less than or equal to $\lfloor m/L \rfloor$.

The guarantee of not replacing a line until it is not needed also requires that the PSA/SSA ready queue issue a 'full signal' before the queue is actually filled. The reason is that the R-cache loader and the fast-path candidate checker (discussed in 'The Fast-Path Mechanism' on page 141) operate in parallel—both the loader and checker can access the R-caches and deposit a ready super-actor in the ready queue simultaneously. That is, in order for the guarantee to be upheld, the PSA/SSA ready queue must issue a full signal when the queue still has one slot left; both the fast-path candidate checker and R-cache loader check with the ready queue before an enabled super-actor is processed. In this manner, an enabled super-actor can be processed without the worry of a needed R-cache line being inadvertently replaced.

Features of the Register-Cache

Beneficial features of the register-cache mechanism include:

- R-cache lines are assigned dynamically at run time to active super-actors, thus providing a capability beyond the conventional static register allocation techniques performed at compile time. For instance, a register allocator may not know if a particular value should be kept in registers because some long-latency instruction is to be executed after the production of the value and before the use of that value. With the register-cache, the value is simply left there, and hopefully, when the consumer is activated, the value is still in register-cache (if not, it is simply fetched from main memory).
- Super-actors (threads) can pass data between each other via high-speed memory.
- There is "true" overlapping of memory block loads/stores with computations because the loader/storer of the R-cache can load a line while the SEU accesses an element in the R-cache. This is useful because a memory block load to the R-cache may take multiple cycles. Conventional caches generally have one port, so while a memory block is being loaded, no other unit can access the cache.
- Local main memory bandwidth requirements can be decreased since the R-cache attempts to keep the most recently used R-cache lines in high-speed

memory. This feature is reinforced by the fast-path mechanism (page 141) which allows frequently activated super-actors to share operands/results effectively.

- Keeping active super-actors' computation times bounded by providing a low and fixed memory latency time.
- Avoiding associative searching (tag-matching) when the super-actor execution pipe accesses values in the R-cache. Instead, the task of associative searching is performed by a separate processing unit removed from the critical execution datapath. And,
- an effective pre-fetch mechanism where a pre-fetched line will not replace a line which is needed by another live super-actor. This benefit comes at a cost of an associative R-cache—the associativity basically determines how many super-actors can be live in a PE of the SAM at any given time. If an ordinary cache with a pre-fetch mechanism replaced a R-cache in the SAM, it would run the risk of replacing lines which are needed by other live super-actors. In general, ordinary caches have low associativities, say four or less,¹⁰ thus there can be unwanted line replacements due to multiple memory blocks mapping to the same cache line. However, low associativities in ordinary caches are required so that access times can be kept low—every access must perform an associative search, and the smaller the associativity, the faster the access. Full associativity can be used in the R-caches because the SEU accesses them directly as if they were register files.

First and foremost, these features are beneficial in a uniprocessor context, and second, these benefits are manifested in a multiprocessing context via a more effective processing element.

6.2.3 The Actor Scheduling Unit

The Actor Scheduling Unit consists of two sub-units: a *signal processor* and an *enable controller* (fig. 6.6). Standard set-associative caches are used for the signal list cache (*SL cache*) and the enable count cache. These caches are used to store the most recently

¹⁰In today's newer generation of RISC computers, directed-mapped caches are often employed.

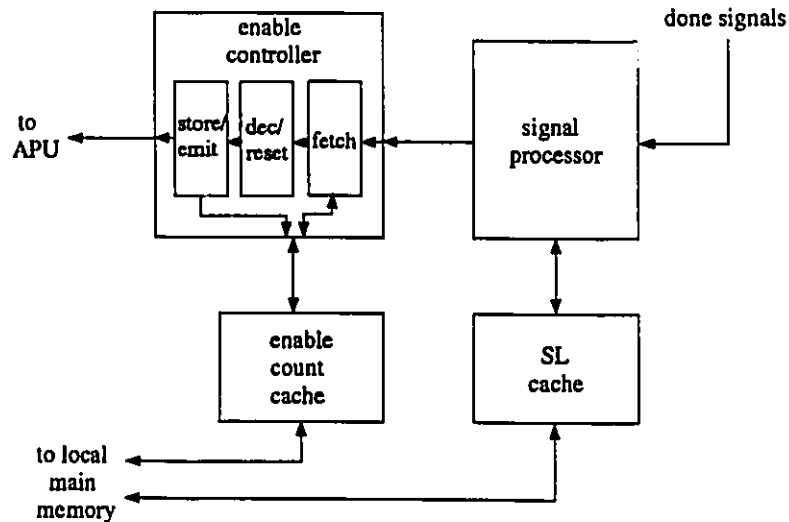


Figure 6.6: The Actor Scheduling Unit.

accessed signal lists and enable counts so that average access times and local main memory bandwidth requirements can be decreased.

The Signal Processor

The signal processor processes *done-signal* tuples and sends *count-signals* to the enable controller stage. A done-signal for an actor is issued by the support-actor execution pipe, the SEU or the LEU and is a tuple containing: a base address, a pointer to the actor's signal list and a condition code (*cond-code*). The signal processor basically takes the done signal and fetches the signal lists according to the condition code and issues a count signal for each entry in the list. An entry in the signal list (*sig-list-entry*) is a 4-tuple and consists of an actor pointer, a local overlay offset value locating the enable count of the actor, a weight for the count signal (*dec-value*), and flag indicating whether the actor is a fast-path candidate or not (*fast-path-cand*). The enable count value will require a few bits (say, four bits) for its representation and is much less than a standard 32-bit word; thus the

enable count offset can be a pair, the first value indicating the offset from *base-addr*, and the second value indicating the offset within the word. Another point to note is that the count signals are weighted by the magnitude of *dec-value*. The reason for weighted count signals will be explained in a later section (see A Uniform Reset Value on page 139). The *count-signal* tuple emitted from the signal processor consists of: a base address, an actor address, a pointer to the enable count value of the actor (*enbl-cnt-ptr*), the decrement value (*dec-value*) and the fast-path candidate flag.

The algorithm describing the operations of the signal processor is listed below. Though not shown, it is straightforward to group the operations into three separate stages, one to fetch the signal list key and calculate the location of the signal lists, the second to sequence through the signal lists, and the last for calculating the enable count pointer and sending off a *count-signal*:

```

algorithm sig-processor
  input <done-signal>
  output <count-signal>
  function
    sig-list-key := Mem[sig-list-ptr]
    /* send signals to actors in uncond. list */
    for (i := 0 to sig-list-key.U-len - 1) do
      entry := Mem[sig-list-key.U-off + i]
      fill-count-signal (entry, base-addr)
      output <count-signal>
    endfor
    /* now send signals to actors in t- or f-sig-list if required */
    if (cond-code = 'T') then
      for (i := 0 to sig-list-key.T-len - 1) do
        entry := Mem[sig-list-key.T-off + i]
        fill-count-signal (entry, base-addr)
        output <count-signal>
      endfor
    if (cond-code = 'F') then
      for (i := 0 to sig-list-key.F-len - 1) do
        entry := Mem[sig-list-key.F-off + i]
        fill-count-signal (entry, base-addr)
        output <count-signal>
      endfor
  endfunction

```


endfor

Note that $Mem[x]$ indicates the value in main memory location x .

```
procedure fill-count-signal (entry, base-addr)
  count-signal.actor-ptr := entry.actor-ptr
  count-signal.enbl-cnt-ptr := entry.enbl-cnt-off + base-addr
  count-signal.dec-value := entry.dec-value
  count-signal.fast-path-cand := entry.fast-path-cand
```

The Enable Controller

The enable controller sub-unit is a 3-stage pipeline consisting of an enable count fetch stage, a decrement/reset stage and a store/emit stage. The enable controller outputs an *enbl-actor* tuple to the APU when an actor has its enable count reach zero. The *enbl-actor* tuple consists of a base address, an actor pointer, and a fast-path candidate flag. The algorithm for the enable controller is:

```
algorithm enable-controller
  input <count-signal>
  output <enbl-actor>
  function
    new-cnt :=  $Mem[enbl-cnt-ptr] - dec-value$ 
    if (new-cnt = 0) then
      output <enbl-actor>
       $Mem[enbl-cnt-ptr] := reset-value$ 
    else
       $Mem[enbl-cnt-ptr] := new-cnt$ 
```

A subtlety in the enable controller has to do with its pipelined structure. Since the fetch stage can function concurrently with the decrement/reset stage, care must be taken such that an enable count value is updated *atomically*. To enforce the atomic updates of count

values, a mechanism akin to bypass latches found in traditional RISC pipelines must be employed between the decrement/reset and fetch stage. In this scheme, the bypass latch is checked to see if the last enable count which was updated is the same as the one entering the decrement/reset stage. If so, then the value in the bypass latch is decremented instead.

A Uniform Reset Value The *reset-value* in the above algorithm is a constant for all enable counts. This uniform reset value is a boon to function applications in that the enable count values of actors in a newly created function do not have to be initialized individually with different values; all enable count values of the new overlay are set to one value. In fact, if the super-actor graph constitutes a *self-cleaning* function body where an actor's state is made ready for the next set of input operands before the function overlay is deleted, then the function creation routine will not have to worry about initializing count values in the enable count cache, though the LEU must still initialize the enable counts in the new overlay, i.e., initialize them in main memory.¹¹ The idea here is simple: if a new overlay uses a memory address for an enable count, and that memory address also happens to be in the enable count cache (that memory address was also used by a defunct function instance for an enable count value), then because of the self-cleaning feature of function bodies, the location in the enable count cache will also contain the correct initial count value (the self-cleaning feature ensures that all enable counts of actors in a terminated function are set to the initial value).

Since the initial enable count values are uniform and the reset values are also uniform, the count signals must be weighted (computed at compile time) so that an actor will receive the correct number of signals before it can be enabled (section 7.5.1).

6.2.4 The Actor Preparation Unit

The APU consists of the fast-path mechanism, an actor attribute fetcher/router, a R-cache loader, an attached support-actor execution unit and various FIFO and LIFO queues

¹¹There is still one more catch before this scheme is realizable, all other high-speed memories in a PE must not contain an enable count value. We will see how this can be enforced at compile time in a later section.

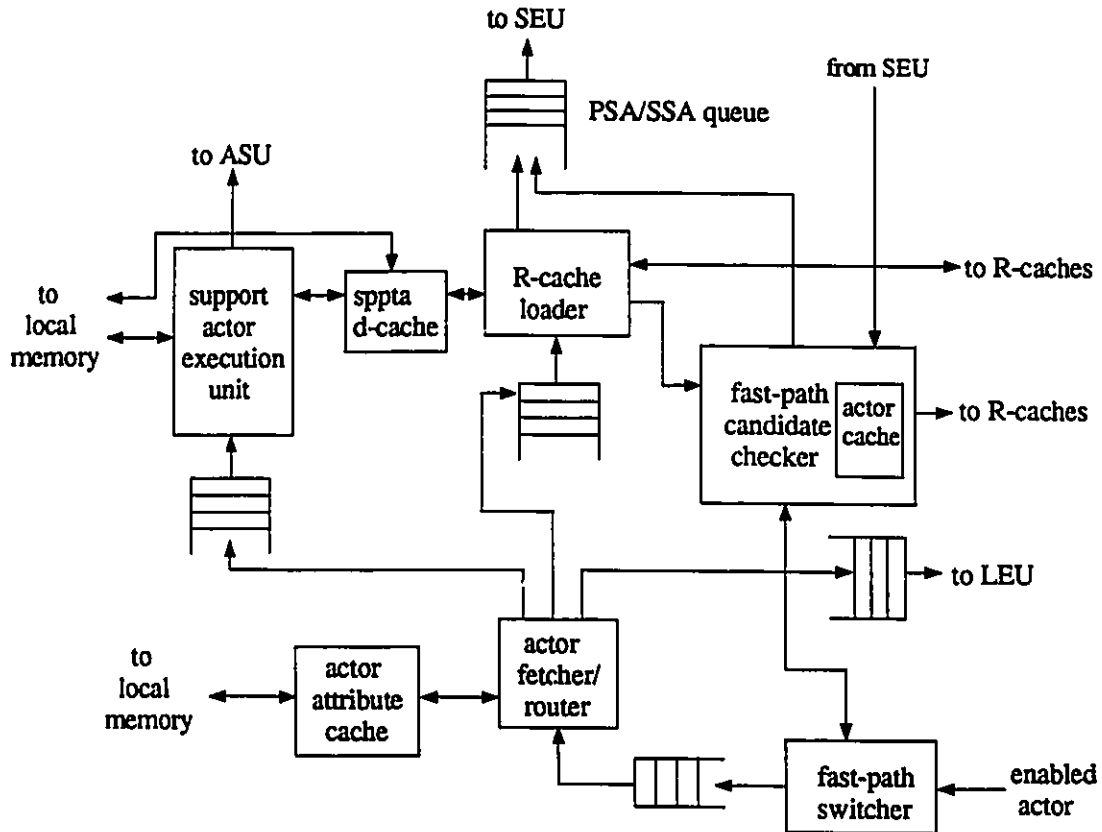


Figure 6.7: The Actor Preparation Unit.

(fig. 6.7).

The *actor fetcher/router* stage is responsible for fetching the actor attributes by examining the word pointed to by *actor_ptr* for the actor type (super-actor = *sa*, support-actor = *sppta*, and L-actor = *La*) and fetching the necessary attributes according to each type. The actor attributes are represented by the *actor-attr* tuple which consists of the actor type (*actor-type*), the pointer to its signal list, and other attributes of the actor (*actor-info*). If the actor is a super-actor, *actor-info* consists of the super-actor type, information indicating where its instructions can be found and information pertaining to the operand/result lines to be pre-loaded into the d-R-cache. If the actor is either a support-actor or a L-actor,

actor-info consists of a pointer to the first instruction of the actor (*instr-ptr*) and the number of instructions of the actor (*length*). Once the router has fetched all the attributes, it can output the information for an enabled support-actor (*enbl-sppta*), an enabled super-actor (*enbl-sa*) or an enabled L-actor (*enbl-L-actor*) to their respective queues.

We believe that the use of a LIFO queue for containing enabled super-actors can better exploit the temporal and spatial locality of data references in the R-caches. A LIFO queue should result in fewer requests by the R-caches to main memory as compared to a FIFO scheduling scheme because a super-actor which was just enabled by a signal of a processed super-actor should have a greater chance of finding an R-cache line containing its operands (the R-cache line for the results of the just processed super-actor) in the data R-cache. As for the FIFO queues of enabled L-actors and support-actors, the scheduling scheme is less important due to their non-deterministic completion times (cache misses, interconnection delays, etc.).

The Fast-Path Mechanism

The fast-path mechanism consists of the *fast-path switcher* and the *fast-path candidate checker*. This path is used to avoid unnecessary loading of super-actor attributes and probing of the R-caches. The idea is simple: for super-actors in loop constructs which are enabled every time the loop iterates, the lines they use might still be in the R-caches when they are enabled the next time around.¹² These super-actors can be tagged by the compiler as possible *fast-path candidates* so that when they are enabled, the fast-path switcher will route them to the fast-path candidate checker (other types of enabled actors are sent directly to the actor fetcher/router) where a small fully associative cache—the *actor-cache*—containing the labels of recently executed super-actors can be checked for its presence. If an enabled super-actor has its entry in the actor-cache, then the R-cache line numbers which it used previously are retrieved, the lines reserved by sending the list of R-cache line numbers to the R-caches and a corresponding *ready-sa* entry is then deposited into the PSA/SSA

¹²Note that if a function overlay can be reused by another instance of the same function, then we may also tag some more super-actors other than the initial super-actors of the function as fast-path candidates.

queue. However, if that instance of a super-actor is not present in the actor-cache, then the *enbl-sa* tuple will be sent back on the regular path (to the actor fetcher/router) where its other attributes can be fetched and the R-caches probed.

The Actor-Cache An entry in the actor-cache (*actor-cache-entry*) consists of a signal list pointer, the start and end offsets locating the start and end instructions in the i-R-cache line, the instruction R-cache line number and the R-cache line numbers of the operand/results. Each entry has a tag containing the following information: a base address, an actor pointer, a flag indicating if the actor instance is active or not (*active-flag*), and an age count. The *active-flag* is used when the *age-counts* of lines are updated. If an entry contains an active super-actor, its *age-count* will not be updated when a new entry is input from the fast-path candidate checker. This is to ensure that the super-actors present in the actor-cache will still have their lines in the R-caches. This will become clearer when the operation of the fast-path candidate checker is detailed next.

The Fast-Path Candidate Checker The fast-path candidate checker takes either a ready super-actor tuple (*ready-sa*) from the R-cache loader, an enabled actor tuple (*enbl-actor*) from the fast-path switcher, or a *terminate-signal* (consisting of a base address and actor pointer) from the SEU as input. A terminate signal from the SEU simply switches off the active flag of the actor-cache entry containing the actor instance as indicated in the terminate signal. This will make the actor-cache entry a candidate for replacement when a new label of a ready super-actor instance comes from the R-cache loader. The new entry is put into the actor-cache and is a replacement for the oldest entry—the one with the largest age-count. Whenever a new entry is put into the actor-cache, age-counts of inactive entries are incremented, not just all entries. The reason is that active super-actors may not terminate in the order they were made ready¹³, and the actor-cache can only replace entries which do not contain active super-actors. When there is an input from the fast-path switcher, the

¹³For example, a sequential super-actor is made active and this sequential super-actor represents a loop (a small loop with, say, less than five instructions in the loop body) which is iterated, say 1000 times. A parallel super-actor is then made active and it only has, say eight instructions. Definitely, the parallel super-actor will terminate before the sequential super-actor.

PSA/SSA ready queue is checked if there is an empty slot or not. Once the checker is allowed to proceed, the actor-cache is associatively searched for the previous activation of the super-actor instance. If its there, the tag of the entry is updated, the data retrieved, and a *ready-sa* tuple is sent out to the PSA/SSA queue. Otherwise, the *enbl-actor* tuple is sent back to the fast-path switcher which sends it along the regular path.

The algorithm for the fast-path candidate checker is listed below:

```

algorithm fast-path candidate checker
  input <ready-sa> from R-cache loader
           <enbl-actor> from fast-path switcher
           <terminate-signal> from SEU
  output <enbl-actor> to fast-path switcher
           <ready-sa> to PSA/SSA ready queue
  function
    if (input from SEU) then
      set active-flag of corresponding entry to 'F'
    if (input from R-cache loader) then
      replace entry with max. age-count and corresponding tag in actor-cache
      with new entry
      increment age-count of entries with active-flag = 'F'
    if (input from fast-path switcher) then
      if (no space in PSA/SSA queue) then wait
      check actor-cache for entry
      if (there) then
        set age-count of that entry to 0 and active-flag to 'T'
        send instr-line to i-R-cache
        send op-res-lines to d-R-cache
        wait for acknowledgement signals from R-caches
        output <ready-sa> to PSA/SSA ready queue
      else
        output same <enbl-actor> to fast-path switcher

```

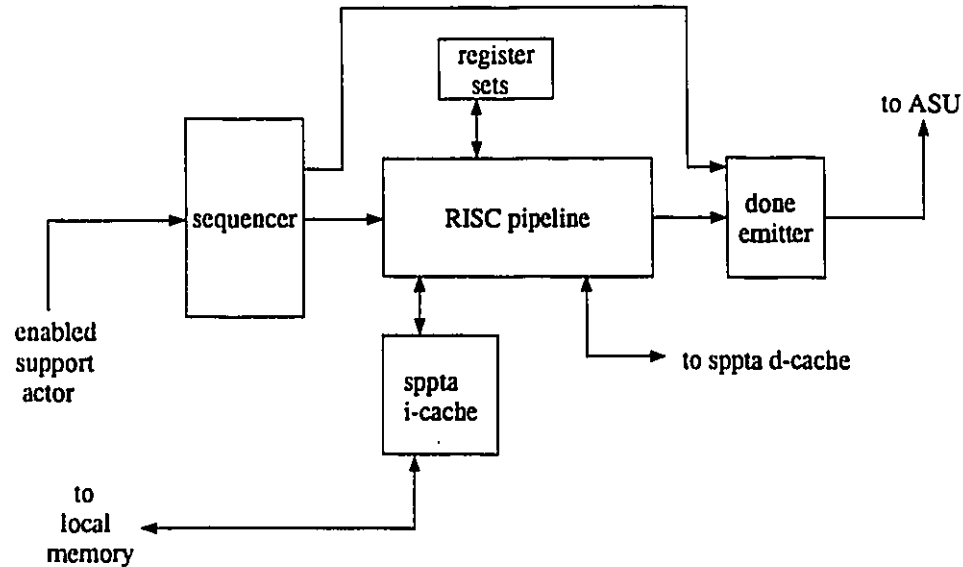


Figure 6.8: The Support-Actor Execution Unit.

The Support-Actor Execution Unit

The support-actor execution unit (fig. 6.8) is responsible for processing support-actors, that is, actors which modify memory block addresses used in indirect block addressing.

The actor sequencer first stores the base address of the actor instance into an assigned register (say, register 0) so that instructions in the support-actor will have access to it. It then issues the first instruction pointed to by *instr_ptr* in the *enbl-sppta* tuple and each successive instruction till all instructions of the support-actor have been issued. At that time, a *done-signal* in which the condition code is set to 'U', is sent to the done emitter stage where it waits till the RISC pipeline triggers it. In order that the RISC pipeline can signal the done emitter stage, an 'end' tag is entered into the RISC pipeline along with the last instruction of the support-actor. When the 'end' tag reaches the result store stage in the RISC pipe, a signal is then sent to the done emitter stage to release the *done-signal*.

To decrease the local main memory bandwidth requirements of the super-actor execution unit, an instruction and data cache (sppta i-cache and sppta d-cache) based on the conventional set-associative organization are used. It is through the sppta d-cache in which a support-actor can communicate its results to a dependent super-actor; the register-cache loader performs indirect data block addressing by accessing the memory address through the sppta d-cache.

The RISC pipeline only executes integer add and multiply instructions, shifts, and the standard load and store instructions since the sole purpose of a support-actor is to perform address calculations. No branching instructions will be supported so the RISC pipe can be much simpler than traditional ones. Another feature of the execution pipeline is that the pipe does not have to be drained of instructions from a previous support-actor before another can enter the pipe. To facilitate this feature, two register sets are used. A bit indicating which set to use is sent with each instruction so that while one support-actor is finishing, another can follow immediately. The bit is flipped everytime a new support-actor is processed by the sequencer. Care must be taken in that the minimum number of instructions in a support-actor must be greater than the number of stages through the longest path in the RISC pipe (in the RISC pipe which we used for the simulations, there are six stages through the path which performs the integer multiply) so that the first and third actors in a string of three simultaneously enabled support-actors will not access the same register set.

6.2.5 The Long-Latency Actor Execution Unit

The Long-latency actor Execution Unit (LEU) is responsible for processing L-actors, i.e., inter-PE operations, function applications, and structure memory accesses. The LEU is also very simple and consists of a sequencer, an execution pipe, an instruction and data cache, and done emitter stage (fig. 6.9). The operations of the LEU components are very similar to the support-actor execution unit but simpler. The sequencer and done emitter perform the same duties, but since there is only one register file, only one L-actor can be in the pipe at any time. Furthermore, the instructions of an L-actor are not simple instructions as found in the support-actor, but complex instructions involving more than two operand values. This

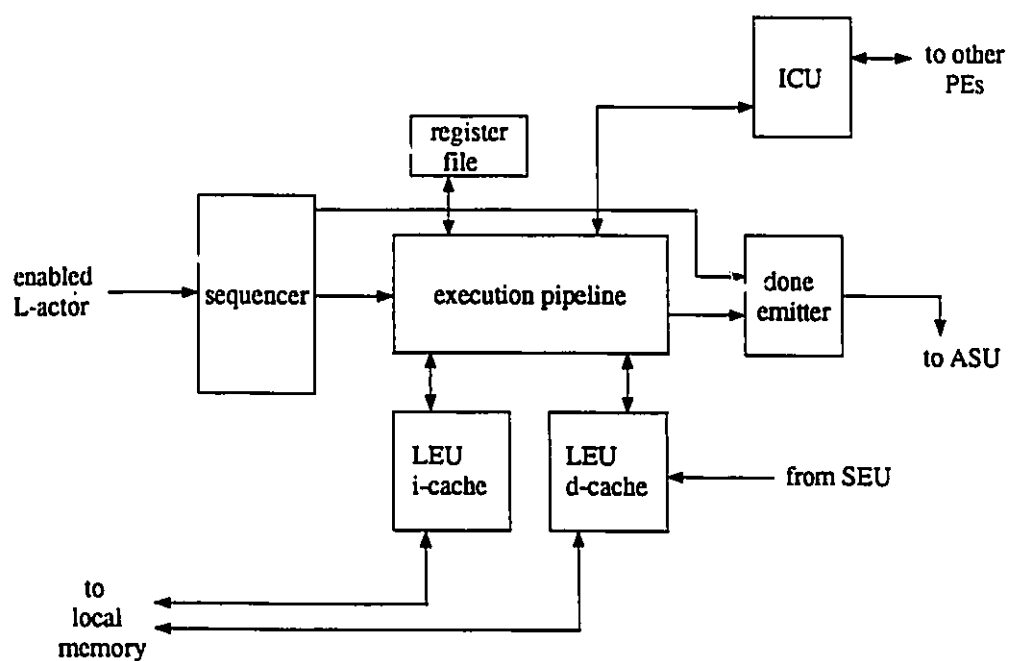


Figure 6.9: The L-Actor Execution Unit.

implies that if a simple RISC pipe is used for the execution pipe, then each L-actor would have to invoke some stored routine (similar to the microprogram store in CISC processors, to process the long-latency instructions. The data cache of the LEU must be a write-through cache since the data an LEU produces may be consumed by super-actors.¹⁴ The only way the SEU can fetch the data which was written by an LEU actor is via the mandatory load from local main memory (not via the LEU d-cache). In figure 6.9, the input path from the SEU to the LEU d-cache is used when the SEU encounters a **wrtto** instruction which copies a d-R-cache line to the LEU d-cache; it is not used for data accesses by the SEU (the 'wrtto' instruction is explained in more detail when we describe the SEU).

The connected interprocessor communications unit (ICU) is responsible for routing packets to and from the PE. It accepts packets from the LEU execution pipe and sends them onto the net. Data from the net are stored directly into local main memory and the ICU notifies the LEU execution pipe to send done signals to the ASU so that the appropriate actors can be notified.

As we have mentioned, the LEU is responsible for handling interprocessor communications, function applications, and structure memory operations. For communications, the execution unit of the LEU creates a packet containing the data, the destination and actor instance ids—actors to be notified—and sends it to the ICU. Creation of a structure memory object involves the searching of a free memory list for the appropriate chunk of memory space and allocating it. Deletion of structure memory objects entails the return of the pointer to the free memory list. The process for function applications is similar to the actions performed by the apply and return super-actors as outlined in the SAM abstract machine model (see Function Applications on page 72).

From these descriptions, it is quite evident that LEU and support-actor instructions are different in terms of their format and processing needs. Thus, it was natural for us to introduce *heterogeneous* processing units into a PE of the SAM.

¹⁴The data cache in the support-actor execution pipe does not have to be write-through since the reg-cache loader is the only other unit which accesses the data produced by support-actors.

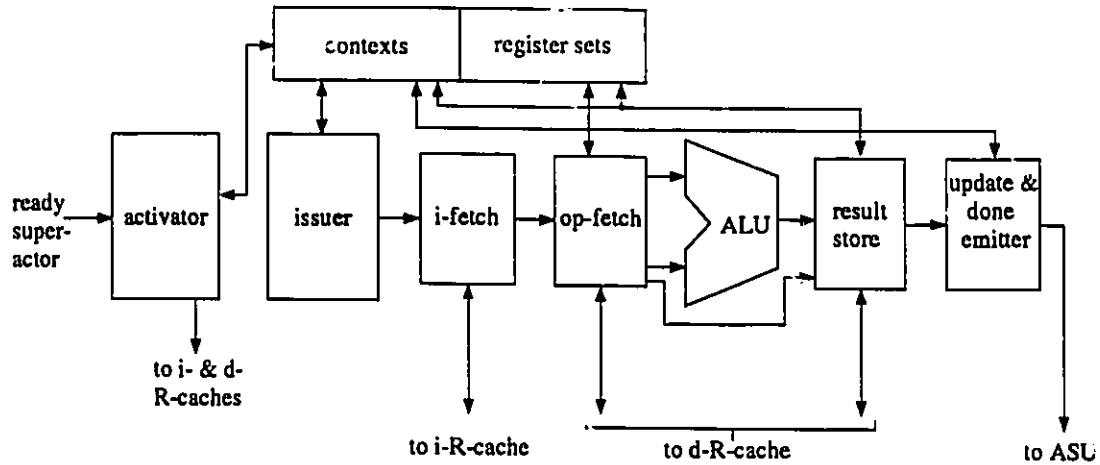


Figure 6.10: The Super-Actor Execution Unit.

6.2.6 The Super-Actor Execution Unit

The structure of the execution unit is shown in figure 6.10. The *contexts* contain sets of registers (we call a set of registers a '*context slots*' or '*context*' for short) for keeping track of the states of active super-actors. The *activator* is responsible for monitoring which context slot is available and loading the available slot with the information of a ready super-actor. The task of the *issuer* is to take the next context which is ready to issue an instruction, say x , and send the necessary information into the execution pipe to initiate instruction x . The *execution pipeline* consists of an *i-fetch*, an *op-fetch*, an *ALU* and a *result store* stage. The *i-fetch stage* only accesses the *i-R-cache* for retrieving instructions and the *op-fetch* and *result store stages* access either the registers in the *register sets* or the *d-R-cache*. The *update & done emitter stage* is responsible for signaling a context containing a sequential super-actor if one of its instructions has exited the *result store stage*. It will also send a done-signal for a super-actor when it encounters its last instruction. The condition-code of the done signal will be indicated by the *result store stage*.

The Contexts

A context constitutes the state of an active super-actor which is currently being processed by the SEU. Each context (*ctxt* tuple) is identified with a unique identifier and the information contained in a context consists of a base address, an actor pointer, pointer to the actor's signal lists, the super-actor's type, the current and end offsets locating the currently processed instruction and last instruction in the instruction line (an i-R-cache line), an ordered list of d-R-cache line numbers, a *ready-flag* and a *free-flag*.

Associated with each context is a register set for storing temporary values and an id number identifying the context (*ctxt-id*). The expression *rs[ctxt-id, i]* identifies the *i*th register in the register set associated with context *ctxt-id*. The register sets cannot be used for passing values between super-actors; the advantage is that state saving is not necessary. We recommend that the number of contexts in the SEU be greater than the maximum number of stages through the execution pipe. The reason is that if all active super-actors in the SEU are sequential, then the execution pipe can still be kept fully busy by issuing instructions from the contexts in a round-robin fashion.

The Activator

The operation of the activator is as follows: when the *free-flag* of a context becomes 'T', the activator will simultaneously issue a *dec-rsrvd-count* signal to the R-caches and a *terminate-signal* to the fast-path candidate checker. (The *dec-rsrvd-count* signal consists of the instruction R-cache line number (sent to the i-R-cache) and a list of operand and result R-cache line numbers (sent to the d-R-cache). The *terminate* signal consists of the *base-addr* and *actor-ptr* of the just freed context.) Then the activator will take the next ready super-actor and store its *base-addr*, *actor-ptr*, *sig-list-ptr*, *sa-type*, *instr-line*, *end-offset*, and *op-res-lines* in their respective locations in the free context. The *start-offset* value from the *ready-sa* tuple will be put in *curr-offset*, the *ready-flag* set to 'T', and the *free-flag* set to 'F'.

The Issuer

The issuer is responsible for monitoring the *ready-flags* of the contexts and picking the first available context. For a ready context, it issues an instruction packet (*instr-packet*) to the execution pipe. The packet consists of: the id of the context (*ctxt-id*), the address of the instruction in i-R-cache (*instr-no*), the super-actor type, a flag indicating whether it is the last instruction or not (*last-instr*), and the operand/result R-cache line numbers. The algorithm describing the issuer operations is shown below:

```

algorithm issuer
  output <instr-packet>
  function
    ctxt-id := id of ctxt with ready-flag = 'T'
    ctxt.ready-flag := 'F'
    instr-no := concatenate (ctxt.instr-line, ctxt.curr-offset)
    increment ctxt.curr-offset
    if (ctxt.curr-offset = ctxt.end-offset) then
      last-instr := 'T'
    else
      last-instr := 'F'
      if (ctxt.sa-type = 'par') then
        /* for parallel sa, set ready-flag back to true */
        ctxt.ready-flag := 'T'
    endif
  output <instr-packet>

```

The Execution Pipe

In designing the execution pipe, a major goal is to *initiate an independent instruction every pipe beat*, thus the *smooth* pipeline should have the following features:

1. it is *clean* or free of *structural hazards*, and
2. all stages in the pipeline have a uniform and fixed processing time.

The last feature is aided by the fact that the memory accessing stages only access registers or the R-caches.

The I-Fetch Stage The i-fetch stage retrieves the corresponding instruction pointed to by *instr-no* from the i-R-cache and sends the tuple $\langle \text{ctxt-id}, \text{sa-type}, \text{instruction}, \text{last-instr}, \text{op-res-lines} \rangle$ to the op-fetch stage. The *instruction* which is fetched consists of: an *opcode*, two values for the operands (*op1* and *op2*) and another value for the result (*res*).

The Op-Fetch Stage The op-fetch stage fetches the operands according to the following four modes: register (*reg*), location in d-R-cache (*Rc*), indirect addressing within a d-R-cache line (*indir*), and immediate (*immed*). These four modes are identical to those in the intermediate abstract machine model (section 5.2.4).

```

case (operand-mode)
  reg:  operand := rs[ctxt-id, i]
  Rc:   operand := d-R-cache[concatenate (ctxt.op-res-lines[i] j)]
  indir: temp-line := int(rs[ctxt-id, i] / line-size)
        temp-off := mod(rs[ctxt-id, i] / line-size)
        operand :=
            d-R-cache[concatenate (ctxt.op-res-lines[temp-line], temp-off)]
  immed: operand := i
endcase

```

The register mode simply indicates that the value is found in the assigned register set of the actor. Indirect addressing within a data block in the SAM (see An Indirect Addressing Mode for Instructions on page 100) is implemented by taking the integer value (the *int* function) of $rs[\text{ctxt-id}, i]$ divided by the R-cache line size as the index into the *op-res-lines* array and using the modulus (the *mod* function) of $rs[\text{ctxt-id}, i]$ divided by the R-cache line size as the offset value. This scheme is more flexible than the one proposed in the intermediate abstract machine model where the indirection can only occur within a pre-specified R-cache line. To enforce the rule that each active super-actor can access only the d-R-cache lines it requested, the lower bound of *i* is set to zero and the upper bound to the length of the *op-res-lines* list minus one. Any value which does not point to an entry in the *op-res-lines* list will be flagged as a run time error.

The *Rc* and *indir* modes involve some calculation and indirections before the operand values are fetched. To aid the execution pipe in having a *smooth* throughput, this op-fetch

stage can be divided into multiple sub-stages. A direct path to bypass some stages for immediate and register type operands can be incorporated. The beauty of this execution model is that there is no ordering specified for instructions within the pipe, thus instructions can complete out-of-order and the execution pipe can be simplified in the sense that no interlocking logic is required.

Once the operands have been fetched, the op-fetch stage sends the tuple *<ctx-id, sa-type, last-instr, opcode, operand1, operand2, res>* into the ALU.

The ALU

The ALU consists of multiple sub-pipes for processing integer operations, floating-point additions, multiplications, etc. Since some pipes have fewer stages than others, e.g., the integer pipe might have two stages, whereas a floating-point add pipe might be six stages long, adding FIFO queues at the end of the shorter pipes and having a prioritized merge logic which favours the longer pipes can allow the ALU to accept an instruction every pipe beat. With multiple sub-pipes, instructions entering the ALU stage may leave out of order, thus some pertinent information from the tuple *<ctx-id, sa-type, last-instr, and res>* which was input from the op-fetch stage must accompany the operation and operands through the sub-pipes in the ALU stage.¹⁵

The ALU outputs a tuple, *alu-out*, containing: the context id, super-actor type, last instruction flag (*lst-instr*), value of the result (*res-value*), information where the result is to be stored (*res*), and a flag indicating whether the d-R-cache line should be copied to other data caches or not (*write-to-mem*). The d-R-cache line would be the line as specified in *res-value*. The *write-to-mem* field contains either 'sppta-d-cache', 'LEU-d-cache' or 'nil', and is set by the operation, *wrtto*, which the ALU would have just processed. (Operation *wrtto* will be detailed in the next section.)

¹⁵Some optimizations can be performed here. For instance, shorter sub-pipes with the same pipe length can share the same FIFO buffers. And latches in each stage which contain the pertinent information of the input tuple can be shared by pipes of equal length.

The Result Store Stage

The modes of the result location are similar to the modes for operands except that there is no 'immediate' mode. The result store stage takes the *alu-out* tuple, stores the result according to information in *alu-out.res* and checks if a d-R-cache line has to be copied to another cache or not. The algorithm for the result store stage is shown below:

```

algorithm result-store
  input <alu-out>
  output <res-packet> to update&done emitter
           <write-to> to d-R-cache
  function
    store res-value according to information in res
    if (write-to-mem = 'sppta-d-cache' or 'LEU-d-cache') then
      send <write-to> to d-R-cache
      wait for acknowledgement signal fro d-R-cache
    output <res-packet>

```

res-packet is a tuple containing the context id, the last instruction flag, the super-actor type and the result value. *write-to* is a tuple containing *write-to-mem* and the d-R-cache line number which is to be copied.

The **wrtto** instruction is used to enforce the consistency between the d-R-cache and the data caches of the support-actor execution pipe and the L-actor execution pipe. It is assumed that at compile time, certain data which are shared amongst the SEU and the other two execution pipes will be determined and **wrtto** instructions inserted into the appropriate super-actors. A **wrtto** operation requires the copying of d-R-cache lines to the other data caches and its *res-packet* tuple can be passed to a sub-unit responsible for issuing the *write-to* tuple to the d-R-cache. There, it can wait for a signal from the d-R-cache to indicate that the line has been copied (copying a d-R-cache will also clear its dirty bit) before the *res-packet* is sent to the update & done emitter stage. This will prevent the blockage of other instructions which can go straight to the update & done emitter stage.

The Update & Done Emitter

The last stage of the SEU takes a *res-packet* tuple as input and may issue a done signal to the ASU. The update & done emitter is responsible for setting the *ready-flag* of the context labeled *ctx-id* if the *sa-type* is 'seq'. Also, if *last-instr* is 'T', the context labeled *ctx-id* will have its *free-flag* set to 'T' and a done signal is emitted to the ASU. To form the *done-signal*, the *base-addr* and *sig-list-ptr* are retrieved from the context labeled *ctx-id*, and the condition code is set to *res-value* if it is either 'T' or 'F', otherwise it is set to 'U'.

6.2.7 Local Main Memory

The local memory is to be a banked memory subsystem where blocks of memory (a contiguous memory space of, say, sixteen words) are interleaved amongst the memory banks. The accessing of a block of memory from a particular bank can be performed in 'burst' mode¹⁶ where devices like *Static-Column RAMs* are used [56]. The banks themselves can be accessed concurrently under the control of a Memory Switch which interfaces the memory banks to the accessing units. As we have described in the previous sections, high-speed memories¹⁷ exist between the local main memory and the processing units which need to access memory so that the average access times and the main memory bandwidth requirements can be decreased.

The Memory Map

The memory map of the local main memory is shown in figure 6.11.

At the bottom of memory sits the working area for administrative instructions, i.e., data structures for overlay management instructions, structure memory operations, inter-PE operations, etc. The rest of the memory is used for storing program segments (PS_x) and

¹⁶The block of data is streamed a word or multiple words every cycle to or from the unit which is accessing it.

¹⁷We did not want to use the word 'cache' since not all of the high-speed memories are caches.

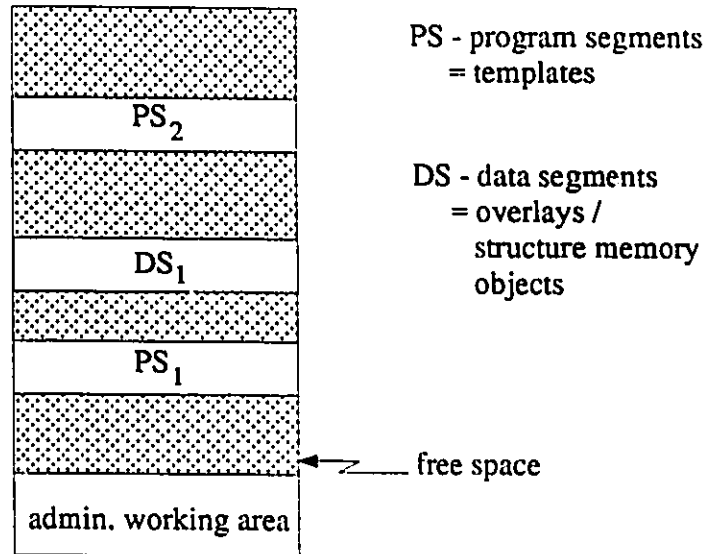


Figure 6.11: The map of local main memory.

data segments (DS_x). The actor attributes, their signal lists, and their instructions are stored in *program segments*. Whereas *data segments* contain overlays of function instances or structure memory objects.

The program segments are divided into contiguous spaces each containing certain types of information. Figure 6.12 shows a typical program segment. The bottom is reserved for local constants and information for function applications such as the overlay size for a function, an offset locating the first enable count value in a function overlay, etc. Since local constants are accessed by the data R-cache, the area must start and end on a data R-cache line boundary ($address \bmod line-size[d-R-cache] = 0$). Instructions of actors are placed in the next upper contiguous area.¹⁸ The instruction area is further divided into sub-areas. Instructions for sequential and parallel super-actors are grouped together and are separate from the group of support-actor instructions, which in turn are separate from the long-latency instructions. The boundaries between these groups of instructions are dependent on

¹⁸This was also alluded to when we mentioned in section 6.2.2 that the memory block address (*mem-block-addr*) is an address to a memory block the size of an R-cache line.

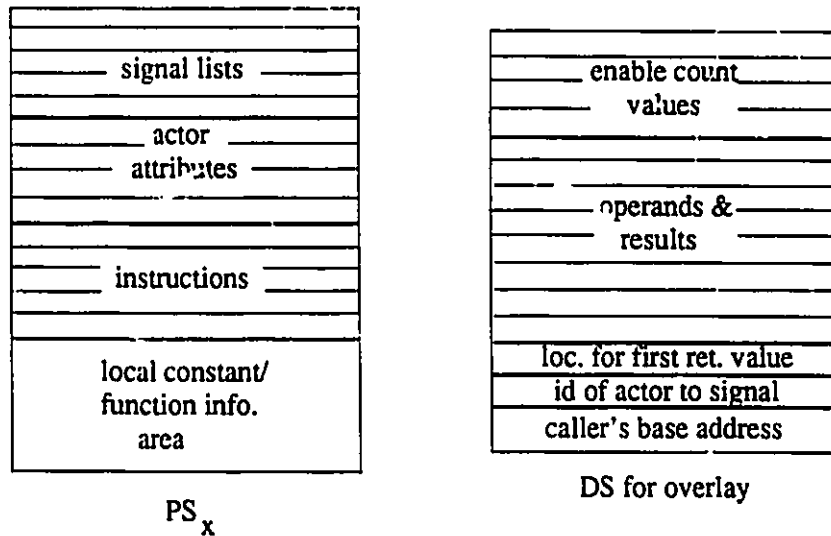


Figure 6.12: The program segment and data segment maps.

the cache line sizes of the processing units which must access them. For example, the LEU must access long-latency instructions and the LEU instruction cache sitting between main memory and the LEU has a cache line size of four words. This information is known by the compiler, thus it would arrange the long-latency instructions to start on a 4-word boundary ($address \bmod 4 = 0$) and assure that the group of long-latency instructions would end on a 4-word boundary.

The actor attribute area of PS_x contains information like pointers to the signal lists, addresses of operand lines, etc. (see section 6.2.4), and the signal list region contains the signal lists. Again, the boundaries of the actor attribute and signal list regions must take into account the line sizes of their respective caches.

The mapping of an overlay in a DS segment is also illustrated in figure 6.12.¹⁹ The caller's base address is located in the first slot, the id of the actor in the caller to notify when after return values are sent back is in the second slot, and the third slot contains the offset

¹⁹We did not illustrate the mapping of structure memory since the data segment usage would be straightforward where the first element would start at the base address of the segment.

from the base of the caller's overlay to store the first return value. in the next. Operand and result values are stored contiguously in the next memory area and the last area contains the enable count values. The boundary between the instruction result area and enable count area is determined by the d-R-cache line size and the line size of the enable count cache.

The "Blocking" of Typed Information

The "blocking" of different types of information, that is, putting one type of information into a contiguous memory space where the memory space starts and ends on a k-word boundary, will allow the multiple caches in the PE to retain the types of data which are accessed by the processing unit the cache is attached to so that:

- prefetching (the loading of a cache line with multiple words) can be more effective, i.e., information which has nothing to do with the processing unit will not be fetched,
- no hardware cache coherency mechanisms are necessary to keep the data consistent in the various caches of one PE, and
- there may be less contentions for a particular bank in local main memory.

The second point is not obvious, so let us examine the various caches in a PE, namely, the instruction caches, the actor attribute cache, the R-caches, the signal list cache and enable count cache. All the instruction caches, the actor attribute cache, the i-R-cache, and the signal list cache are read-only (let us call these *read-only caches*), and since they do not contain any data common to the data caches, d-R-cache, and the enable count cache, then the read-only caches do not affect the consistency. Data which are written by the SEU and are accessed by the support-actor execution pipe or LEU will be kept consistent in the data caches by the **wrtto** instruction, i.e., by software control. Data written by the LEU and are accessed by the SEU are kept consistent by the write-through data cache in the LEU and the mandatory load mechanism in the APU. Data written by the support-actor execution pipe may be accessed by only the R-cache loader and no other execution pipe requires access to that data. Data written by the LEU and accessed by the support-actor

pipe are rare and the SEU can be used as a go-between, i.e., the compiler can introduce a super-actor which performs a mandatory load and then a **wrtto** instruction to copy the data to the sppta d-cache. Lastly, the enable count cache contains data not found in the other data caches, so there are no cache consistency problems. However, a problem may develop when an overlay is reused by another function invocation. Values in the sppta data cache and LEU data cache from a function instance which used the same overlay may still exist. (Old values in the d-R-cache are of no concern if the compiler adheres to the rule which stipulates that all initial super-actors of a function use mandatory loads for their operand lines.) This problem can be resolved if some cache line invalidation mechanism is employed when an overlay is deallocated.

6.3 Summary

In this chapter, we have presented an implementation of the advanced abstract machine model (section 5.2.5). The mapping of the machine model to this implementation is as follows: a major portion of the deactivation-enabling agent corresponds to the ASU; the remaining portion of the deactivation-enabling agent which fetches the actor attributes, the SA-readying agent, OA-activation agent and the SA-activation agent maps to the APU; the SA-execution agent is the SEU, and the OA-execution agent is represented by the attached support-actor execution pipeline and the LEU. Lastly, the memory units in the memory model are represented by the respective caches and the main memory.

We have detailed one possible implementation of the Super-Actor Machine which uses the register-cache mechanism to hide local memory latencies. Heterogeneous processing units are employed within one PE of the SAM to exploit the parallelism between different types of actors. Variations of the basic configuration are possible, for example, the merging of the support-actor execution unit and the LEU, an instruction issuer in the SEU which emits two or more instructions into the execution pipe, allowing the support-actor execution pipe to process conditionals, utilizing a register-cache for the support-actor execution pipe, etc. Such variations are subject to further investigation where trade-off issues must be

considered. Other issues to be examined in the future include: the design of the inter-processor communications unit and how multiprocessing can be efficiently supported, the implementation of virtual memory support, and a formal analysis of the cache consistencies in one processing element of the SAM. In the next chapter, we will look at some compiling techniques for generating machine code for the Super-Actor Machine.

Chapter 7

Generating Code for the Super-Actor Machine

A compiler which generates machine code for the SAM might contain four stages: a front end, a partitioner, a translator and an assembler (fig. 7.1). The front end is responsible for parsing a program written in a high-level language and generating a well-formed dataflow graph. The high-level language can be a functional language such as SISAL[80], VAL[2], Id[85], etc. or an imperative language, e.g., FORTRAN. Generating a well-formed dataflow graph from a functional language is a well understood process [1, 114], and only recently, has there been work [18] which outlines a method for generating dataflow graphs from a subset of FORTRAN. Machine independent optimizations like loop invariant removal, constant folding, dead code removal, etc.[5] can also be performed on the dataflow graph and is beyond the scope of this thesis. The generated dataflow graph from the front-end will be attributed with information to aid the partitioning phase in aggregating dataflow actors into super-actors. Once a well-formed super-actor graph (section 5.1.3) is generated, the translator will produce the corresponding Super-Actor Machine Assembly Language (SAMAL) representation (appendix C outlines a preliminary version of SAMAL). In the last stage, the assembler takes the SAMAL code, generates the executable format for the SAM and at the same time, packs that executable format (actor attributes, instructions, etc.) into memory areas with computed boundaries as stipulated in section 6.2.7.

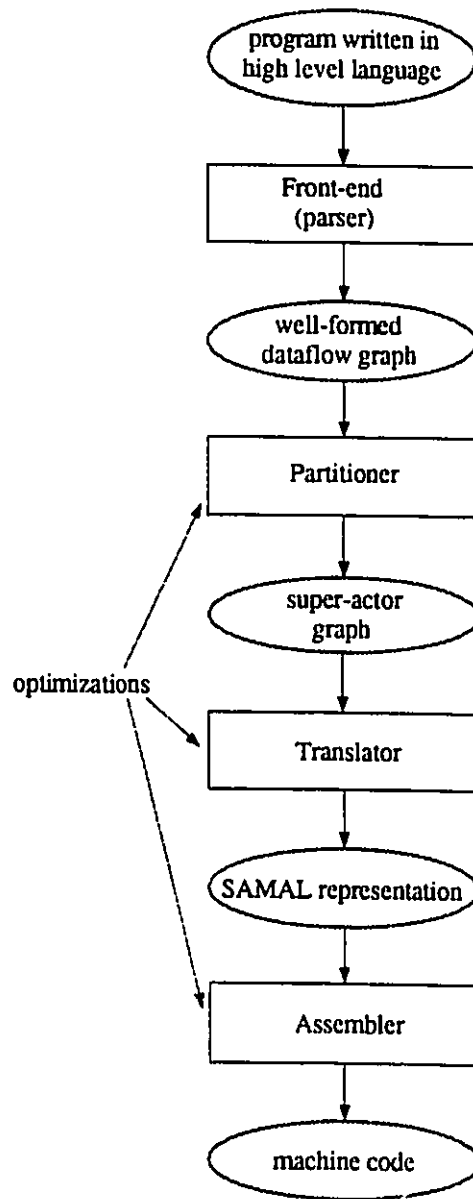


Figure 7.1: A possible organization of a compiler for the SAM.

Much work has been done on the front end of compilers and most of it can be adapted here to generate dataflow graphs; thus we do not dwell on the issues in the front-end stage. Instead, this chapter focuses on the partitioning techniques for generating super-actors (sect. 7.2 and 7.3), the translation (sect. 7.4), and the assembly processes (sect. 7.5). But first, well-formed dataflow graphs—inputs to the partitioning phase—are reviewed in the next section. The reason why we suggest that the partitioning phase operate on well-formed dataflow graphs is that there is an existing and readily available compiler (the Id compiler [114]) which generates a well-formed dataflow graph as an intermediate form. (Perhaps a more efficient method of generating super-actor graphs can have the partitioner operate upon some form of an abstract syntax tree as generated by the front-end parser, but we will leave this to future work.)

7.1 Well-Formed Dataflow Graphs

In the review of dataflow computing (section 1.2.1), we have shown examples of well-formed dataflow graphs in the form of a simple arithmetic and logic computation block, a conditional construct and a loop construct. The partitioning algorithm as outlined in this chapter takes a well-formed dataflow graph as input and generates well-formed super-actor graphs. The determinacy property of dataflow computing can be retained in the super-actor execution model if the structures of well-formed constructs—the structures of conditional and loop constructs in a dataflow graph—are retained in the super-actor graph (SA graph). In this chapter, we illustrate the structures of well-formed constructs by borrowing the notion of *encapsulators* from Traub[114].¹ (In section 5.1.1, we borrowed the notion of encapsulators to explain the syntax and semantics of function definitions, and conditional and loop constructs in a SA graph.)

¹Traub used encapsulators for grouping dynamic dataflow actors. Here, we use them to group static dataflow actors.

7.1.1 Encapsulators in a Dataflow Graph

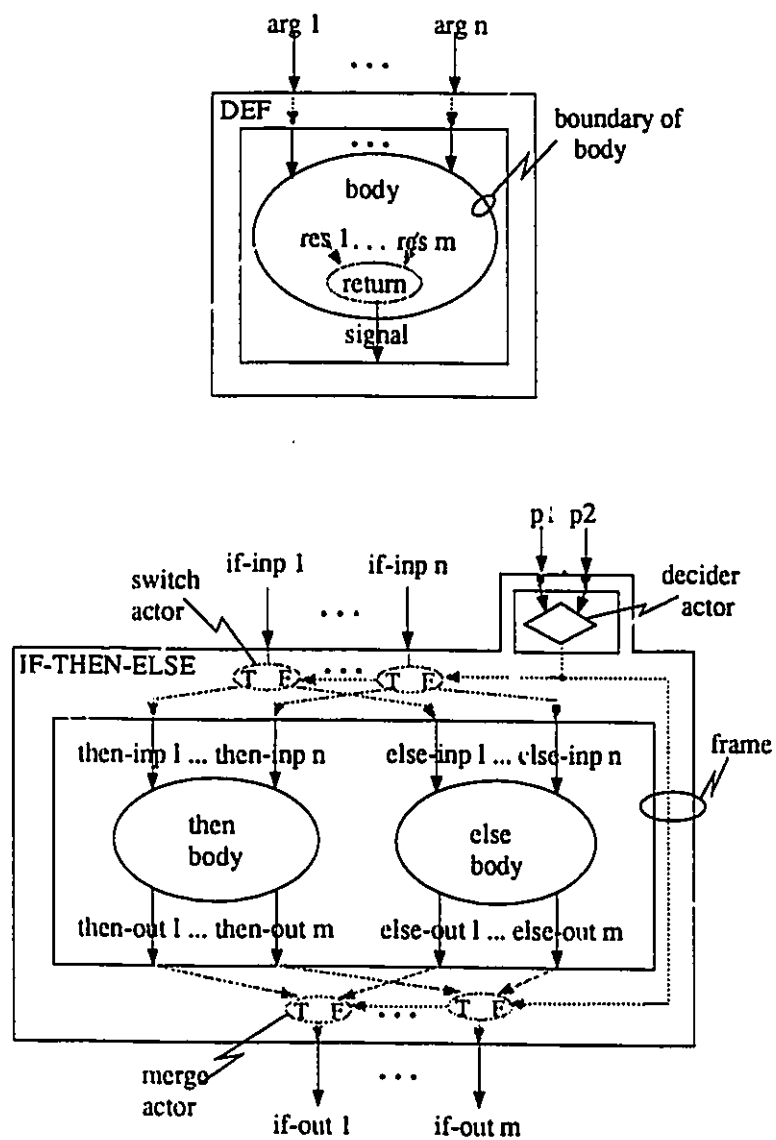
Three major encapsulators are used: a *def*, an *if-then-else*, and a *loop* encapsulator (figure 7.2 and 7.3).

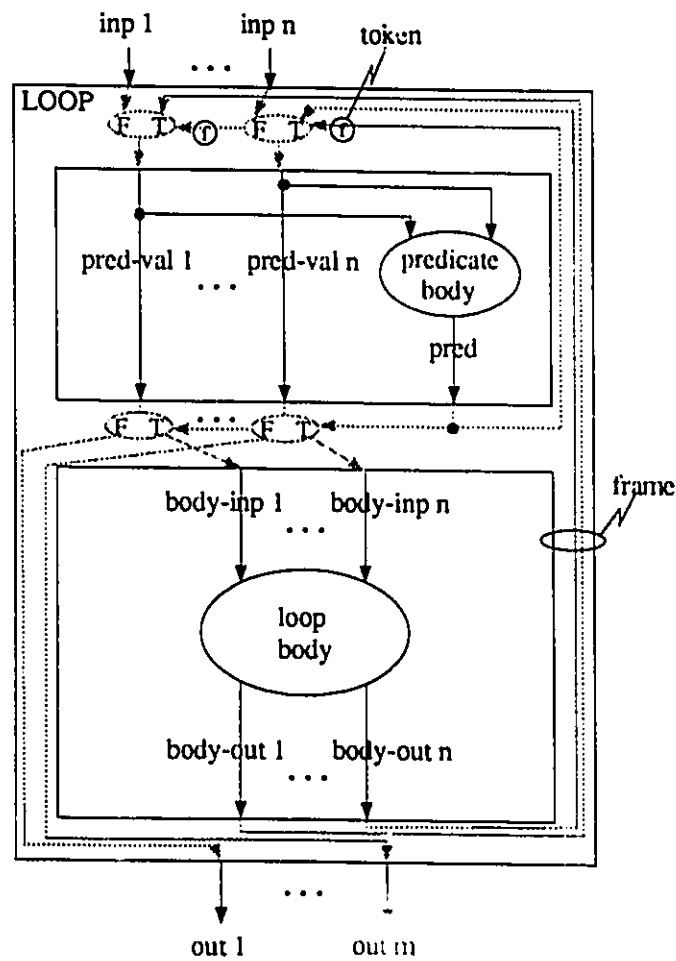
The *def* encapsulator corresponds to a function codeblock of a program and contains a *body* (acyclic graph) of one or more nodes. A *node* is either a dataflow actor, an *if-then-else* encapsulator or *loop* encapsulator, where the bodies within *if-then-else* and *loop* encapsulators also contain nodes (as defined below). When a function is invoked, the input arguments (tokens on *arg* arcs) trigger dataflow actors in the body and eventually, tokens are produced on the result arcs (*res* arcs) which are then routed back to the caller—performed by a ‘return’ actor. Lastly, a token is sent out of the body (along the *signal arc*) to the *def* encapsulator which triggers the termination of the function. For all *def* encapsulators except the *main* *def* encapsulator—the main function definition of the program—a return actor will send results back to the caller.

The *if-then-else* encapsulator represents an *if-then-else* expression and contains switch actors, merge actors, a decider actor (see sect. 1.2.1 for the operations of different dataflow actors), and a *then-* and an *else-body*.² The *then-body* and *else-body* can each contain an acyclic graph of one or more nodes. Merge actors and switch actors (shown in hashed lines in the figure) appear in the *frame* of the encapsulator and are used in the actual dataflow graph—the one to be executed by a dataflow machine—to implement the *if-then-else* expression. The *p* inputs are routed to the decider actor which generates a condition code to route the inputs (tokens coming in on *if-inp* arcs) to the *then-inp* or *else-inp* arcs. Either the nodes in the *then-body* or *else-body* will be triggered and the outputs of the ‘triggered’ body sent to the merge actors. There, they are routed to the output arcs (*if-out*’s). The encapsulator cannot process another set of inputs until the merge actors have executed—firing rules for a static dataflow merge actor—that is, until one of the bodies have produced the outputs. Thus the construct can exhibit a determinate behaviour.

The *loop* encapsulator is used to represent a loop expression and consists of merge actors, switch actors (switch gates), a *predicate-body* (*pred-body*) and a *loop-body*. Again,

²In Traub’s work, the decider actor is not included in the encapsulator.

Figure 7.2: The *def* and *if-then-else* encapsulators.

Figure 7.3: The *loop* encapsulator.

the merge actors and switch gates used in the actual dataflow graph are shown in hashed lines in the frame of the encapsulator, and each body (an acyclic graph) can contain one or more nodes. The encapsulator takes its inputs (from the *inp* arcs) and sends them onto the *pred-val* arcs via the merge actors. The tokens with an 'f' value (for false) indicate that the merge actors are ready to route the input tokens from the *inp* arcs to the *pred-val* arcs. The *pred-val* arcs supply the data to the *pred*-body and the switch actors. The predicate-body is thus triggered and produces a token on the *pred* arc. If the token on the *pred* arc is true, the tokens on the *pred-val* arcs are fed to loop-body (through the switch gates) and the loop body is triggered. When the output of the loop-body is produced, they are routed to the *pred-val* arcs and the cycle continues once more. When the token on the *pred* arc is false, the values on the *pred-val*'s get routed to the *out* arcs by the switch actors and the outputs of the loop generated. The token with a false label from the *pred* arc is also sent to the merge actors and the loop is ready to process the next set of inputs. In this manner, the loop construct processes only one set of inputs at a time and can exhibit a deterministic behaviour.

With this description of encapsulators, it is evident how a well-formed static dataflow graph can be mapped to a dataflow graph containing actors and encapsulators. We call the latter an *encapsulated dataflow graph*. A program can be represented by a set of *def* encapsulators (function definitions) and is called an *encapsulated program*.

7.2 The Partitioner

In this section, we define two terms to differentiate encapsulators in a dataflow graph and those in a super-actor graph. A *df-encapsulator* is an encapsulator containing dataflow actors, and a *sa-encapsulator* is an encapsulator of super-actors.

The input to the partitioner is an encapsulated program, and for each *def* *df-encapsulator* in the program, a well-formed SA graph is produced, i.e., a *def sa-encapsulator*. There are two phases in this process: first, each *def df-encapsulator* in the encapsulated program is partitioned, that is super-actors and if-then-else and loop *sa-encapsulators* created; and

second, location assignments for the results of super-actors are performed along with the generation of *block assignments* (sect. 5.2.4) for each super-actor. In the following sections, we will first look at how super-actors and well-formed constructs in a SA graph can be generated (sect. 7.2.1 to 7.2.2). In section 7.2.3, we discuss the deadlock-free property of the partitioned graph. Lastly, section 7.2.4 illustrates the partitioning process with an example.

7.2.1 The Partitioning Phase

The partitioning strategy for generating a well-formed SA graph is as follows:

1. from the encapsulated dataflow graph, dataflow actors in the same body can be grouped into a super-actor such that the structure of the if-then-else and loop constructs can be maintained in the SA graph;³ and
2. assure that the super-actors do not create cycles which can lead to deadlock.

The reason for possible deadlocks is that the SAM executes a super-actor instance in an *atomic* manner where it must execute till completion once activated (see operational semantics of a super-actor instance on page 60). In other words, an instruction within a super-actor must either get its input from another instruction within the aggregate or its input *must already have been produced* by another super-actor instance before it can be activated. Therefore, erroneous groupings which can lead to deadlock must be avoided in the aggregation phase. An example of an improper grouping which results in an unpermitted cycle between super-actors A_i and A_j is shown in figure 7.4; permitted cycles are the looping structures found in loop sa-encapsulators since they are retained from the loop df-encapsulators during the transformation process.

The Method of Dependence Sets

To create a SA graph which has no static cycles that can cause deadlock, we base our algorithm on an algorithm described by Iannucci in his dissertation[75] to perform the

³We will see later how an if-then-else or loop construct can be grouped within a super-actor.

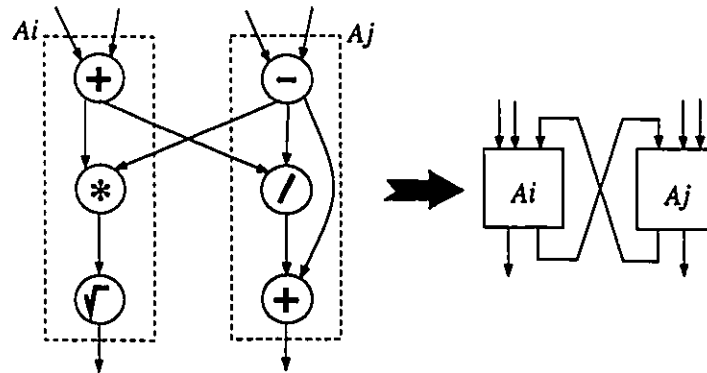


Figure 7.4: An example of an improper aggregation of actors which can lead to deadlock.

partitioning of dataflow actors in an acyclic subgraph of the encapsulated dataflow graph. The *Method of Dependence Sets* (MDS) takes an acyclic dataflow graph (the acyclic portions of a dataflow graph) and generates aggregates called *scheduling quanta* (SQ). This algorithm deals with the avoidance of static cycles and *dynamic cycles*—a dynamic cycle is a cyclic dependence arising from *I-Structure*[16] accesses. However, we do not have the problem of dynamic cycles since we do not intend to support I-Structures.

In Iannucci's MDS algorithm, each node (dataflow actor, an if-then-else, or a loop df-encapsulator) is assigned an input dependence set and an output dependence set. A *dependence set* contains a set of node identifiers. The algorithm traverses the graph in topological order, assigns the output dependence set of a node based on its input dependence set and its type; the input dependence set is dependent on the output dependence set of the node's input nodes. (The input dependence set of nodes with no input and nodes which are immediately executable when the function they belong to is invoked is the empty set.) If a node is a long-latency actor—in our work, a long-latency actor is one which takes an unknown amount of time for completion such as a structure memory operation, a function application operation or an inter-PE instruction⁴—then its output dependence set is the union of its input dependence set and a set containing its id, otherwise, it is simply its input dependence set. As input and output dependence sets are assigned, SQs are formed

⁴In Iannucci's work, a long-latency instruction (actor) is generally characterized by an I-structure access.

from nodes having the same input dependence set. In effect, new SQs are created for the successor actors of a long-latency actor. Note that an SQ can contain multiple if-then-else and/or loop constructs if there are no long-latency actors within those constructs.

The Super-Actor Generating Algorithm

In our work, long-latency actors cannot be grouped with other actors, so we must modify the MDS algorithm. Also, we do not need to create super-actors with multiple if-then-else and/or loop constructs (in Iannucci's work, the larger the SQ, the better), thus we can retain the structures of if-then-else and loop constructs (structures which are the basis for well-formed SA graphs). Lastly, we must introduce signal arcs which are not a one-to-one correspondence with the arcs in the original dataflow graph.

Below, are the definitions as used in our partitioning algorithm called the Super-Actor Generating Algorithm (SA-gen):

Definition 7.1 *A node is either a dataflow actor, an if-then-else df-encapsulator, or a loop df-encapsulator.*

Definition 7.2 *The Input Dependence Set of a node i , $IDS(i)$, is the union of the output dependence sets of all nodes from which i receives input.*

Definition 7.3 *The Output Dependence Set of a node i , $ODS(i)$, is the union of $IDS(i)$ and the singleton set containing the id of i if i is a long-latency actor or a df-encapsulator, otherwise, $ODS(i)$ is simply $IDS(i)$.*

Given the set of *def* df-encapsulators representing the encapsulated program,

$$P = \{D_1, D_2, \dots, D_n\}$$

where D_i is an encapsulated dataflow graph, the SA-gen will generate an encapsulated graph of "pseudo super-actors", that is, a graph of sa-encapsulators containing pseudo

super-actors. A pseudo super-actor is similar to a super-actor, the exceptions are that it contains one or more dataflow actors (instead of super-actor instructions) and it has no block assignments (R-cache loading instructions) yet. Each pseudo super-actor will eventually become a super-actor after the dataflow actors are converted and block assignments inserted. The term *sa-node* implies either a (pseudo) super-actor—includes a (pseudo) L-actor—or an if-then-else or loop sa-encapsulator of (pseudo) super-actors.

A *sink node* of node i is a node on which an output arc of node i terminates, thus function *sink-nodes* (x) returns a list of sink nodes of node x . Also, there is no sink node at the end of an output arc which crosses a body boundary (a boundary of either a body of the def encapsulator, a then-, else-, pred- or loop-body) so that the *partition-graph* procedure may terminate properly, i.e., instructions exterior of a body cannot be grouped with instructions within the body). Function *id*(x) returns a unique label corresponding to x , where x can be a node, a body (then-, else-, pred-, or loop-body), or a def df-encapsulator. And function *append* (x,y) takes the list of elements in y and appends it to the end of the list of elements in x .

algorithm *SA-gen*

Input P

output encapsulated graphs of pseudo super-actors

function

for (each D_i of P) **do**

 insert a top node, T , which signals

 all nodes which receive all their inputs from the incoming arguments

$ODS(T) := id(D_i)$

 partition-graph (sink-nodes(T))

 insert-arcs (T)

endfor

procedure *partition-graph* (*to-do-list*)

while (*to-do-list* is not empty) **do**

$I := \text{pop}(\text{to-do-list})$

if (I has not been processed and

$ODS(\text{source of input arc of } I)$ is defined for all input arcs of I) **then**

$IDS(I) := \text{union of } ODS()'s \text{ of all nodes from which } I \text{ receives input}$

```

case (type of  $I$ )
  encapsulator:
    partition-encapsulator ( $I$ )
     $ODS(I) := IDS(I) \cup id(I)$ 
    append (to-do-list, sink-nodes ( $I$ ))
  long-latency actor:
    create  $SA(id(I))$ 
    put  $I$  in  $SA(id(I))$ 
    label  $SA(id(I))$  as 'LA'
     $ODS(I) := IDS(I) \cup id(I)$ 
    append (to-do-list, sink-nodes ( $I$ ))
  ordinary dataflow actor:
    create  $SA(IDS(I))$  if it does not exist
    put  $I$  at end of  $SA(IDS(I))$ 
     $ODS(I) := IDS(I)$ 
    append (to-do-list, sink-nodes ( $I$ ))
endcase
endwhile

```

```

procedure partition-encapsulator ( $I$ )
  if ( $I$  is an if-then-else df-encapsulator) then
    create  $SA(id(x))$  for decider actor  $x$ , and put  $x$  in  $SA(id(x))$ 
    create pred-body in  $I$  and put  $SA(id(x))$  into pred-body of  $I$ 
    label  $SA(id(x))$  the switch super-actor of the pred-body of  $I$ 
    for ( $body := \{then-body, else-body\}$ ) do
      insert top node,  $T$ , which signals all nodes of  $body$  which receive
        all their inputs from the incoming arguments
       $ODS(T) := id(body)$ 
      partition-graph (sink-nodes ( $T$ ))
      insert-arcs ( $T$ )
    endfor
    create-well-formed-cond ( $I$ )
  else /* df-encapsulator is a loop */
    for ( $body := \{pred-body, loop-body\}$ ) do
      insert top node,  $T$ , which signals all nodes of  $body$  which receive
        all their inputs from the incoming arguments
       $ODS(T) := id(body)$ 
      partition-graph (sink-nodes ( $T$ ))
      insert-arcs ( $T$ )
    endfor
  end

```

```

endfor
create-well-formed-loop (I)

```

Procedure *insert-arcs* (T) performs the following steps on a body of nodes (body of a def encapsulator, then-, else-, pred-, or loop-body):

1. *Insert signal arcs.* Traverse acyclic dataflow graph rooted at T and for each output arc of node x which does not exit the body do (that is, a node which is an encapsulator is treated as if it is a dataflow actor and its internal structure is not traversed):

```

y := node on which output arc of x terminates
case (type of x)
  dataflow actor:
    case (type of y)
      dataflow actor: create signal arc from super-actor containing x to
                      super-actor containing y, if it does not exist and they are
                      not in the same super-actor
      if-then-else encap.: create signal arc from super-actor containing x
                          to switch super-actor of pred-body of y, if it does not exist
      loop encap.: create signal arc from super-actor containing x
                   to corresponding merge node in y5
    endcase
  if-then-else encap.:
    case (type of y)
      dataflow actor: create signal arc from corresponding merge node
                      in x to super-actor containing y, if it does not exist
      if-then-else encap.: create signal arc from corresponding merge node
                          in x to switch super-actor of pred-body of y
      loop encap.: create signal arc from corresponding merge node
                   in x6 to corresponding merge node in y
    endcase
  loop encap.:
    bval := negation of the label of a labeled arc terminating in
           loop-body of x
    case (type of y)
      dataflow actor: create a bval labeled signal arc from switch
                      super-actor in pred-body of x to super-actor containing y,

```

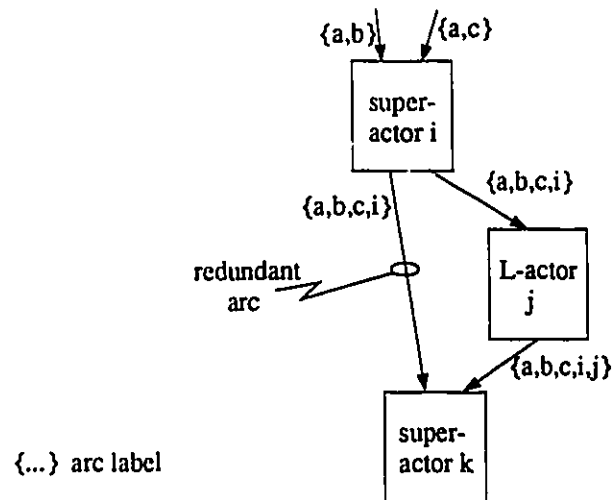


Figure 7.5: Example of a redundant arc.

if it does not exist
if-then-else encaps.: create a *bval* labeled signal arc from switch
 super-actor in pred-body of x to switch super-actor of
 pred-body of y
loop encaps.: create a *bval* labeled signal arc from switch
 super-actor in pred-body of x to corresponding merge node in y
 endcase
 endcase

2. *Removal of redundant arcs.* Redundant signal arcs between sa-nodes in the body should be removed (an example of a redundant signal arc is shown in figure 7.5). This can be accomplished by first labeling the output arcs of the associated top virtual node with the set containing the id of the body. Then each outgoing signal arc of an sa-node in the body is labeled with the union of the singleton set containing the id of the sa-node and the labels of the input signal arcs of the sa-node; thus a label of an arc is a set of ids (see figure 7.5). An input arc of sa-node x is deemed redundant and can be removed if its label

⁵Note that an incoming arc in a loop df-encapsulator terminates on a merge actor, thus there is a corresponding merge node for each incoming arc.

⁶Note that every outgoing arc of an if-then-else df-encapsulator originates from a merge actor, thus there is a corresponding merge node for each outgoing arc.

is a subset of a label of another of α 's input arcs.

3. *Removal of redundant merge nodes in loop encapsulators.* Redundant virtual merge nodes in loop encapsulators⁷, identified by having the same two predecessor sa-nodes, can be combined into a new merge node which has as input the same two sa-nodes, and as output, the union of output arcs of the redundant virtual merge nodes.

Procedure *create-well-formed-cond* (I) transforms an if-then-else df-encapsulator, I , to an if-then-else encapsulator containing pseudo super-actors (fig. 5.4 shows an if-then-else sa-encapsulator). The only things missing are the input and output arcs of the sa-encapsulator; they are added in procedure *insert-arcs*. The following steps are executed to produce an if-then-else sa-encapsulator:

1. Add labeled signal arcs. Add true output arc(s) from the switch super-actor in the pred-body to the sa-node(s) containing dataflow actor(s) pointed to by the top node (added in partition-encapsulator) in the then-body. And similarly, add false output arc(s) to the else-body. Remove the two top nodes in the then- and else-bodies.
2. *Create virtual merge nodes.* For each merge actor in the frame of the original encapsulator, create a virtual merge node. For each input arc to the merge actor, add an input arc from the corresponding sa-node of the source of that dataflow arc to the virtual merge node.⁸ Redundant virtual merge nodes, identified by having the same two predecessor sa-nodes, can be combined into a new merge node which has as input the same two sa-nodes.

Procedure *create-well-formed-loop* is similar to *create-well-formed-cond* except that it transforms a loop df-encapsulator to a loop sa-encapsulator (the reader should refer to fig. 5.5 for the structure of the loop encapsulator in a SA graph). Just as with *create-well-formed-cond*, the input and output arcs of the sa-encapsulator will be added by *insert-arcs*. *Create-well-formed-loop* performs the following steps on encapsulator I :

⁷Redundant merge nodes in if-then-else encapsulators are removed in function *create-well-formed-cond*.

⁸Note that the virtual merge nodes are non-deterministic, so its input arcs do not have to be destined for a specific port as does the input arcs of a merge actor in an encapsulated dataflow graph.



1. *Insert labeled signal arcs.* Locate the switch super-actor in the pred-body⁹ and add true (false) output arc(s) from the switch super-actor to the sa-node(s) containing instructions pointed to by the top node (added in partition-encapsulator) in the loop-body, that is those sa-nodes which are initial sa-nodes. (True output arc(s) are added if dataflow actors in the loop-body received tokens from the true arcs of the switch gates, otherwise false output arc(s) are added.) Remove the two top nodes in the pred- and loop-bodies.
2. *Create virtual merge nodes.* For each merge actor in the frame of the original encapsulator, create a virtual merge node. For an input arc from a node in the loop-body to the merge actor, add an input arc from the sa-node of the source to the virtual merge node. And for each output dataflow arc of the merge actor, add an output arc from the virtual merge node to the corresponding sa-node or switch super-actor which contains the dataflow actor the merge actor signals.

When signals are added in the steps of procedure create-well-formed-cond and create-well-formed-loop, some signals may originate or terminate from sa-nodes which are not super-actors. In those cases, the rules as outlined in the first step of procedure insert-arcs should be followed.

Algorithm *SA-gen* basically calls procedure partition-graph for each def df-encapsulator. The top node, T , is a virtual node (see section 5.1.1) introduced by the partitioner; i.e., it does not appear in the final machine code.¹⁰ A top node of a def sa-encapsulator is used to indicate the actors to notify when the function is invoked. Moreover, its output dependence set contains the id of the def df-encapsulator so that dataflow actors within the df-encapsulator can only be grouped with other dataflow actors within the same df-encapsulator; the input dependence sets of the actors which the virtual top node signals will contain the id of the def df-encapsulator, thus the input dependence sets of all actors in the df-encapsulator will contain that id.

The *partition-graph* procedure traverses a dataflow graph and groups dataflow actors with the same input dependence set together, i.e., if $IDS(i) = IDS(j)$ then dataflow actors

⁹The condition code generating instruction may not be the last instruction in the switch super-actor so some reordering may be necessary.

¹⁰Bottom nodes as described in section 5.1.1 are not necessary since the 'return' instruction already performs a function instance termination.

i and j belong in the same super-actor, $SA(IDS(i))$. For long-latency dataflow actors, an individual super-actor (an L-actor to be exact) is created for each one. Assigning a long-latency instruction to an individual L-actor ensures that the successor dataflow actors are put into a new super-actor. This is very important since we do not want instructions waiting for results from a long-latency operation to be waiting in the SEU. Note that the algorithm processes a node only when all of its predecessor nodes' ODSs have been defined, thus the instructions in a super-actor would be in a topological ordering since instructions are appended to the end of a super-actor's instruction list.

When an if-then-else or loop df-encapsulator is processed, a body within a df-encapsulator is treated like the body in a def encapsulator. That is, a top node is inserted which has output signal arcs to the nodes which can be fired once the body is activated, and the output dependence set of the top node contains the id of the body. The latter action ensures that dataflow actors in the body can only be grouped with other dataflow actors within the same body. When a body of a df-encapsulator has been partitioned, *insert-arcs* is invoked just as is done for the body of a def df-encapsulator. Top nodes in the then-, else-, pred- and loop-bodies are only introduced to facilitate the *create-well-formed-xx* procedures—called after the encapsulator has been partitioned—unlike the role of the top node in a def sa-encapsulator.

As the reader may note, procedure partition-graph operates on acyclic subgraphs of the well-formed dataflow graph; cycles of loop constructs are 'hidden' by procedure partition-encapsulator. Also, the switch actors and merge actors are hidden from partition-graph, thus they are not grouped into any super-actor.

When the SA-gen terminates, the list of dataflow actor(s) within a pseudo super-actor constitutes the list of instructions of that super-actor and the signal arcs created in insert-arcs, create-well-formed-cond and create-well-formed-loop are the arcs in the SA graph. A restriction of this algorithm is that only L-actors and sequential super-actors are produced. Moreover, this algorithm can produce arbitrarily large super-actors. We will see later (page 183) how support-actors and parallel super-actors can be formed, and how we can limit the number of instructions within a super-actor.

In the SA-gen algorithm, the structure of if-then-else and loop constructs are preserved; the if-then-else and loop df-encapsulators in the encapsulated dataflow graph are simply transformed to the if-then-else and loop sa-encapsulators in a SA graph. Original data flow arcs between ordinary dataflow actors (e.g., 'add', 'mul') are either transformed to signal arcs between super-actors (a signal arc between super-actors may replace two or more dataflow arcs), or are deemed redundant due to the topological ordering of dataflow actors (as represented by the list of instructions) within a super-actor. Therefore, data dependencies between instructions are implicitly maintained, and the well-formedness of the original dataflow graph can be retained in the SA graph.

7.2.2 The Location Assignment Phase

The second and last phase in the transformation process involves the assignment of locations in overlay blocks to the outputs of super-actors and the creation of *block assignment entries*—load instructions for the R-cache loader—for parallel and sequential super-actors. Dataflow actors which produce results only for consumption by dataflow actors within the same super-actor can use temporary registers, while other dataflow actors producing results to be consumed by dataflow actors external of the super-actor must be assigned a location within an overlay block. The original dataflow arcs in the encapsulated dataflow graph should be used as a guide for reserving a location within an overlay for results of instructions of super-actors and also for indicating where an instruction in a super-actor can find its operands.

This phase must be aware of where input arguments to a function are placed (in the AAA overlay block—see fig. 5.3), and for constant values which cannot be expressed as an immediate operand of an instruction, those values are put into a table containing constants. Results of a super-actor should be kept within one overlay block, and if results of a super-actor do not fill an entire block (the size is determined by the d-R-cache line size in the underlying machine implementation) then that block may be shared with other actors.

Caution must be exercised in that memory blocks which are modified by a particular execution unit must contain like-wise data such that cache-consistency problems do not

arise (see The Blocking of Typed Information on page 157). For instance, if a block contains data written by the sppta execution pipe, then all other locations in that block must contain data written by that execution pipe because if it contains, say data written by the LEU, then the sppta d-cache and the LEU d-cache can be inconsistent if that block appears in both caches simultaneously with differing values. This can lead to erroneous values being stored in main memory when dirty cache lines are copied back.

Lastly, block assignment entries for super-actors are created and dataflow actors within super-actors are converted to the instruction formats similar to those found in the instruction set of the abstract machine model (section 5.2.3).

7.2.3 Deadlock-Free Super-Actor Graphs

We claim that for any acyclic subgraph of dataflow actors, the application of SA-gen produces an acyclic subgraph of super-actors. This can be proven by contradiction, and the formal proof can be given along the line as used by Iannucci [75]. An informal argument is as follows: a cycle involving one AE-node is not possible. The reasons are that in procedure *insert-arc*, step one prohibits the creation of a loop involving one super-actor (the case where x and y are both dataflow actors), and that procedure *insert-arc* itself only traverses an acyclic graph while inserting arcs. A cycle involving two or more AE-nodes is not possible because from the partitioning algorithm, this would imply that the input dependence set of one of the AE-nodes in the cycle is a *proper subset* of itself, which is impossible (from the partition-graph procedure, the input dependence set of a AE-node contains the union of its input nodes' output dependence set, thus its input dependence set can never be a proper subset of that of its predecessor).

One last question to address is that, does the same inputs to a dataflow graph and its generated SA graph produce the same outputs? Since the list of instructions in a super-actor is in a topological ordering of the corresponding dataflow actors (procedure partition-graph), the SA graph is deadlock-free, and the well-formed SA graph retains the well-behavedness and determinacy properties of the dataflow graph, then the answer is yes.

7.2.4 An Example Partitioning

Let us now step through the algorithm with an example. Figure 7.6(a) shows an encapsulated dataflow graph which computes the expression:

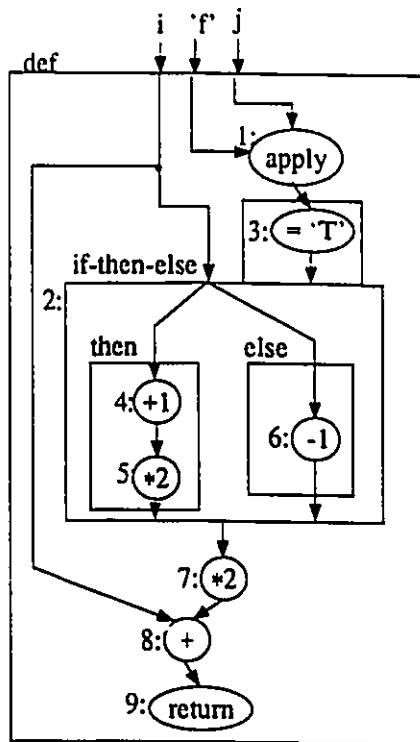
```

function  $g(i, j)$ : integer returns integer
   $k :=$  let
     $b := f(j)$ 
     $h :=$  if  $b$  then
       $(i + 1) * 2$ 
    else
       $i - 1$ 
    in
       $(h * 2) + i$ 
    end let
  returns  $k$ 
end function;

```

When the encapsulated dataflow graph is processed by algorithm *SA-gen*, a *top* node is introduced such that its outgoing arc would point to the ‘apply’ node and partition-graph is called with the id of the apply actor in the *to-do-list*. The partition-graph procedure processes the ‘apply’ actor and since the apply node is a long-latency actor, a super-actor, $SA(<1>)$, is created with the single instruction (long-latency actor a in fig. 7.6(b)). (Note that we used 0 as the id of the def df-encapsulator. Also, the original dataflow arcs are represented by dashed directed arcs in figure 7.6(b), and the solid arcs represent signal arcs.) The if-then-else node is processed next and procedure partition-encapsulator takes over. The decider actor (actor 3) is put into $SA(<3>)$ (super-actor b), and the then- and else-bodies are each processed like a body of a def df-encapsulator resulting in nodes 4 and 5 being grouped into super-actor c and node 6 in a separate super-actor (d). Since there is only one super-actor in each body, no signal arcs are added by function *insert-arc*. In procedure create-well-formed-cond, the pred-body of the encapsulator is created with switch super-actor b , the labeled signal arcs inserted and the top nodes in the then- and else-bodies removed. The virtual merge node (node m) is created with signal arcs from super-actors c and d . Popping up to the def df-encapsulator level, node 7 is processed next.

a) dataflow graph



b) graph with pseudo super-actors

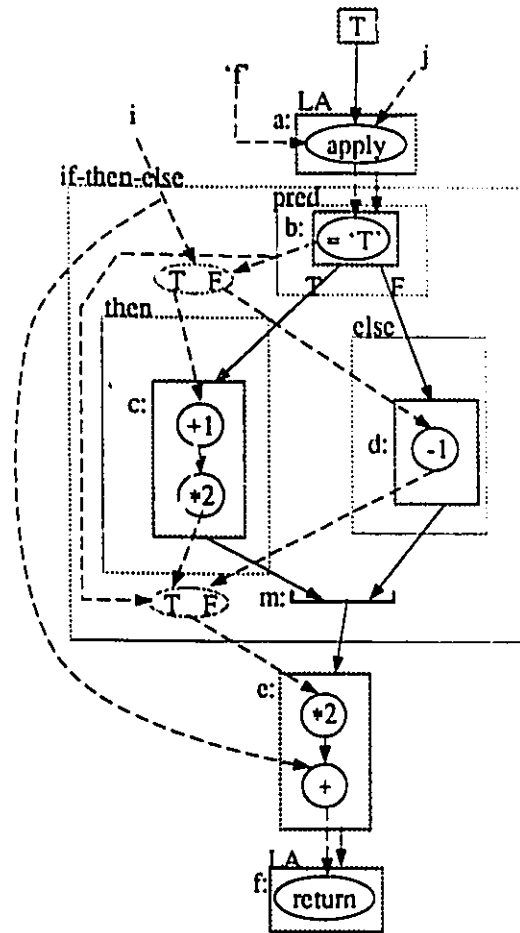


Figure 7.6: A partitioning example.

Since $IDS(7) = (0, 1, 2)$, it will be put in a new super-actor, $SA(<0, 1, 2>)$ (super-actor e). Node 8 is next and because $IDS(8) = (0, 1, 2)$, it will be added to super-actor e . Finally, node 9 is processed and since it is a long-latency actor, it is grouped by itself into long-latency actor f , $SA(<9>)$. Procedure *insert-arcs* is then invoked for the body of the def sa-encapsulator. An arc from a to the if-then-else sa-encapsulator is created and it is directly routed to the switch actor in the pred-body of the if-then-else sa-encapsulator—actor b . An output arc from the if-then-else to actor e is created, and the source of that signal arc is from virtual merge node m , the (only) corresponding merge node of the merge actor. Next, an arc is added from actor e to actor f , and the *insert-arcs* procedure is terminated. The partition-graph procedure is then terminated since the *to-do-list* has become empty and control is returned to algorithm SA-gen.

Next, allocating overlay locations to super-actor results is performed. Block $s0$ is introduced to hold the output values of actor a , and c or d (c and d write to the same location). Actor b 's output is merely discarded—stored in a temporary register—so it needs no space in the overlay. Actor e accesses its operands from $s0$ and stores its output in $s0.1$; the location where actor f can find its operand. The input values to the function, i , j and the function name ' f ' are stored in the AAA block along with the caller's base address, the *ret-sig* actor and the offset value from the caller's base address locating the position to store the return value (see Function Applications on page 72). Next, block assignment entries are generated for actors b , c , d , and e , and dataflow actors are converted to the proper instruction formats of super-actors. The resulting SA graph is shown in figure 7.7.

'apply' and 'return' instructions are shown for the first time in a SA graph (fig. 7.7), so let us detail their syntax (they are similar to the syntax of apply and return macro-instructions in fig. 5.9). For the apply instruction, there are three operands and an optional fourth one. The first operand is a local overlay offset (*block-ptr-id.offset*) indicating the function pointer, the second operand is also a local overlay offset, but it indicates the location of the first argument of the function application. The third operand is yet another local overlay offset and it indicates the location where the first return value is to be stored. Subsequent return values are stored in the next locations. The optional fourth argument—an immediate value—indicates the number of arguments to be sent to the callee function; the absence of



Figure 7.7: Part 2 of the partitioning example.

the operand implies that only one argument is to be passed. The return instruction has one operand and an optional second one. The first operand is a local overlay offset indicating the location of the first return value. The optional second operand indicates—an immediate value—the number of return values to be sent back to the caller; the absence of the operand signifies that there is only one return value.

7.3 Considerations in Partitioning

7.3.1 Machine Specific Constraints for Partitioning

The partitioner must be aware of a few architecture and implementation specific features and constraints: partitions for support-actors, the length limit of parallel and sequential super-actors, making values written by the SEU visible to the other units, and ensuring that all instructions within a parallel super-actor would be executed before a done signal is sent to the ASU.

Instructions Within Parallel Super-Actors Instructions can have varying completion times and if a parallel super-actor consists of, say a mix of integer and floating-point instructions, then the partitioner must order the instructions such that the one with the longest completion time be the last in the list. Since the SEU turns on the ‘last instruction’ flag on the last instruction packet of an active context and the SEU only issues a done signal when the result-store stage encounters such a packet, then it can be guaranteed that all previous instructions within the parallel super-actor would have been executed if the last instruction in the list takes the longest time for completion.

Making SEU Written Values Visible to other Units If a sequential or parallel super-actor produces a value which is to be accessed by a long-latency actor or support actor, then the partitioner must automatically place a **wrtto** instruction at the end of that super-actor. The **wrtto** instruction would specify that the d-R-cache line containing the results be made

visible—to the LEU (if the consumer is an L-actor) or the support-actor execution pipe (if the the consumer is a support-actor). That is, the line is copied to the appropriate data cache (LEU d-cache or SPPTA d-cache).

The Length Limit The number of instructions within a parallel and sequential super-actor is limited by the maximum number of i-R-cache lines allocated per active super-actor multiplied by the number of instruction words per i-R-cache line. For instance, if the SAMi implementation allows only one i-R-cache line of sixteen words per super-actor, then the partitioner can only create parallel or sequential super-actors each having a maximum of sixteen instructions. This can be easily enforced in the algorithm by keeping a running count of instructions in regular (non support and non long-latency) super-actors; once the count is equal to the maximum, the super-actor is deemed filled. This length limiting restriction requires a slight modification in the partitioning algorithm, where after a regular super-actor has been completely filled, the ODSs of each instruction within the super-actor are augmented with a unique identifier, say the id of the first instruction in the super-actor. In this manner, the just filled super-actor will be treated like an encapsulator so that consumer instructions of the super-actor can be properly aggregated and the deadlock-free property can be maintained.

An equal concern is the number of d-R-cache lines allocated per active parallel or sequential super-actor. If a super-actor requires data from too many different super-actors, then memory block sharing between super-actors should be considered. This scenario basically occurs for parallel super-actors and not for sequential super-actors. The reason is that inputs to sequential super-actors would be generated by two or less super-actors within the same body since the partition-graph procedure starts a new super-actor when the output dependence set of the source nodes of a dyadic node are different. If a super-actor requires data from too many different memory blocks, then in the worst case, id nodes for moving data will have to be introduced.

Partitions for Support-Actors As was mentioned in section 5.2.5, support-actors are used to alter the address of the memory block a parallel or sequential super-actor can

access. This indirect data block addressing mode can be used if the structure memory object (or a portion of the structure memory object) which the super-actor is supposed to access is known to reside in local main memory. This high-level information can be specified by the programmer or determined by some data partitioning algorithm. (To ensure that a structure memory object and the code which accesses it reside on the same processing element, an extra input must be added to the **SMalloc** and **apply** instructions for informing the long-latency actor execution unit to create a structure memory object locally (**SMalloc**) and to invoke a particular function locally (**apply**). The intent was for the LEU to perform dynamic load balancing via automatic distribution of function instances to less busy processing elements. However, letting the programmer or compiler have control over where data and function instances are created can lead to more efficient processing.) Figure 7.8 shows a SA graph which uses a support-actor. This SA graph is similar to the one shown in figure 5.8. The differences are that: the **SMread** long-latency actor (actor 4 in fig. 5.8) is replaced by a support-actor (actor 5), and an extra super-actor (actor 1) is introduced to pre-process the inputs to the loop. We call actor 1 a *loop pre-processing actor* and its function is to decrement the address of array 'A' by four, and perform a **wrtto** of that block to the data cache of the support-actor execution pipe.¹¹ The reason for decrementing the address of array 'A' by four in the loop pre-processing actor is that the support-actor increments it by four before actor 6 is fired. In super-actor 6, the '@' is used to indicate that the block assignment to *B2* is an indirect data block address.

7.3.2 Some Optimizations in Partitioning

In this section, we examine some possible optimization techniques which can be employed before, during, or after the partitioning phase. Some transformations which might be performed are: the grouping of an entire if-then-else or loop construct within a sequential super-actor, the enlargement of the switch super-actor in an if-then-else construct, the parallelization of loop bodies, software pipelining of loop bodies, and the identification of

¹¹The syntax of 'wrtto' instruction is as follows: it contains two operands, the first operand specifies which data cache the data block is to be copied to, the d-cache of support-actor execution pipe ('sppta) or that of the LEU ('leu). The second operand specifies which block is to be copied via a *b-pir* (*Bxx*).

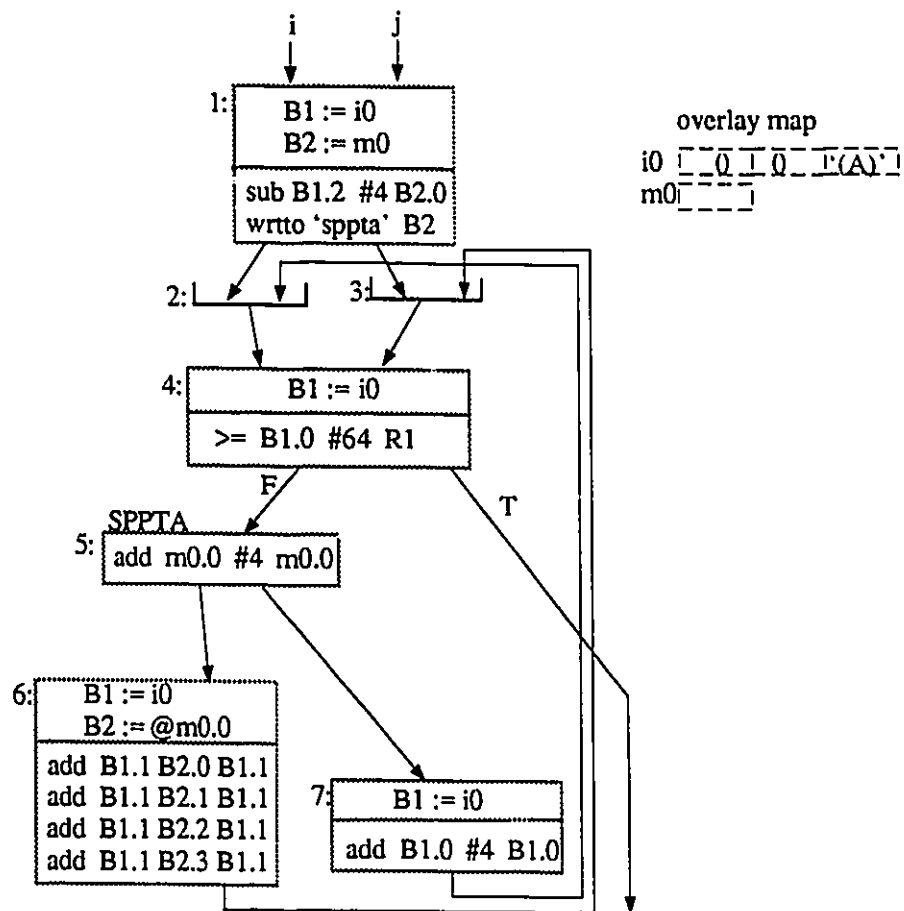


Figure 7.8: An example where support-actors are used.

fast-path candidate super-actors.

Small If-Then-Else and Loop Constructs An if-then-else construct can be classified as 'small' enough if: it contains less than x instructions where x is determined by the machine, and there are no long-latency operations within its then- and else-bodies. If a small if-then-else construct is identified, then it would be advantageous to group the entire construct within one sequential super-actor. For instance, the if-then-else in figure 7.6 can be flagged as a small one such that the partitioner would put the whole expression into one sequential super-actor. As for a small loop construct, an added stipulation is that all the data which the loop body operates upon must be contained in the input memory blocks of the sequential super-actor. This implies that if a structure memory accessing actor or a super-actor which performs indirect block addressing appears in a loop body, then that loop is not considered a 'small' loop.

Enlarging the Switch Super-Actor in a Conditional The switch super-actor in the if-then-else construct, as generated by the partitioner, contains only one instruction. The switch super-actor can be enlarged if the if-then-else df-encapsulator includes a body of one or more dataflow actors for generating the boolean value for routing the inputs. For example, in a high-level language, the construct:

If p then x else y

can signify that the expression p should be the body of dataflow actors to generate the boolean value for the if-then-else df-encapsulator. Then this new *pred-body* can be partitioned in a similar fashion as the *pred-body* of a loop df-encapsulator, possibly producing a switch super-actor with more than one instruction.

Parallelizable Loops When a loop is identified as parallelizable, e.g., a 'forall' in VAL terms [2], the instructions in the body can be replicated to form a parallel super-actor which

consumes a block or two of data and produces another block of data. That is, these parallel super-actors can exploit the data parallelism within the loop and are thus akin to vector instructions—a vector instruction processes blocks of data (chap. 4 of [72]). In each parallel super-actor, the same operation appears multiple times and the number of operations would be equal to the maximum number of instructions per super-actor. The operand and result fields of each instruction in a parallel super-actor would be different since each operates on its own set of inputs and produces its own result(s). The blocks of data which are operated upon basically correspond to vector registers in a vector processor since they will be loaded into the d-R-cache before the parallel super-actor (vector instruction) can be executed.¹²

For example, the loop:

$$D := \text{for } k \text{ in } 0, n \text{ returns array of} \\ (A[k] + B[k]) * C[k]$$

can be flagged as parallelizable. Figure 7.9 shows the SA graph for the above example when it is not parallelized. Array D is the output of this loop and since it is allocated by some predecessor of the loop, i.e., its address is stored in memory, the loop can simply send a signal to its successor when it is done. The loop pre-processing actor (actor 1) decrements all of the array addresses by one¹³ such that the increment of the induction variable k in i0.4 can be performed in the switch super-actor (actor 2) instead of another super-actor in the loop-body. This results in a more efficient execution because an extra super-actor in the loop-body (the one which would have to increment the induction variable in loop-body) can be eliminated.

When the compiler encounters the loop construct above,¹⁴ it can automatically generate the parallel super-actors, alter the structure memory reads and writes to work in block mode (actors 3, 4, and 5), alter the subtraction instructions in the loop pre-processing actor (actor

¹²We are assuming that the number of words in a d-R-cache line is equal to that of the i-R-cache line.

¹³Note that we cheated and let the pre-processing actor deposit its results in the locations of its inputs; the partitioner explicitly indicates that the actor should have its own result location. As long as this loop is not presented with a new set of inputs while it is executing, then it does not really matter.

¹⁴The compiler should be able to detect that this is a parallelizable loop either by automatic inspection or as told by the programmer.

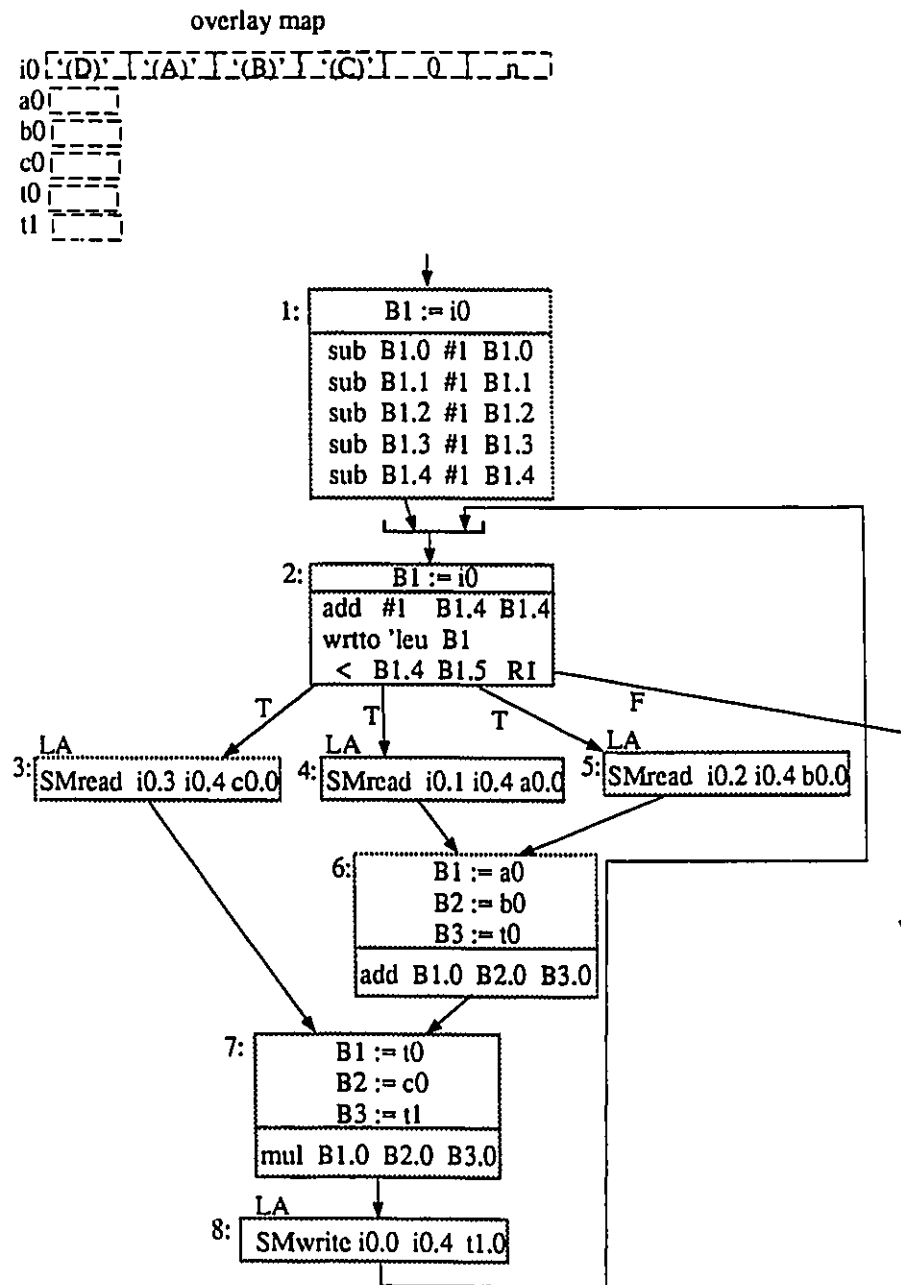


Figure 7.9: A super-actor graph for the parallelizing example.

1), and alter the add instruction in the switch super-actor (actor 2). Figure 7.10 shows the resulting parallelization. For the example in figure 7.10, we assumed that there are sixteen words per R-cache line so that the loop was unrolled sixteen times. Parallel super-actors (labeled *PSA*) perform the “vector add” (actor 6) and “vector multiply” (actor 7).

The above example shows that the Super-Actor Machine can be used effectively to exploit data parallelism just as vector computers can with operations like vector-add, vector-multiply, etc. Other more complicated vector operations are also possible, for example, vector reduction instructions such as vector-sum (a vector instruction which takes a vector and adds its elements to produce a sum), can be performed with sequential super-actors. The question is: will such sequential super-actors be as effective as vector-sum instructions in vector computers? This can be answered indirectly by realizing this fact: if multiple sequential super-actors performing the vector-sum operation are simultaneously enabled and each one is executed in round-robin fashion in the SEU, then the SEU can be fully utilized as is the case with parallel super-actors.

A major difference between the SAM’s “vector instructions” and those of a vector computer is that multiple operators must be repeated within a parallel super-actor, whereas a vector instruction in a vector computer only specifies the instruction once. This seems like memory space is wasted to repeat the instruction multiple times in a ‘vector’ super-actor, but the trade-off is for simpler logic in the issuer of the super-actor execution pipe. Furthermore, the copy of instructions in a ‘vector’ super-actor can be shared by multiple actors. For instance, if two or more parallel super-actors were to perform a vector-add on different vectors, then the actor attributes of those super-actors can point to the same instruction code block. The MultiTitan architecture[77] also uses multiple scalar instructions to replace vector instructions, but the translation is done at run time. The advantage cited for the MultiTitan is that vector and scalar registers can be treated uniformly, thus our architecture also has the advantage of that property.

Dataflow Software Pipelinable Loops As the reader may have noticed, even though the above example loop was parallelized (‘vectorized’), i.e., arrays *A*, *B*, and *C* were operated on in block mode, *vector chaining* [72] where vector registers act as buffers between

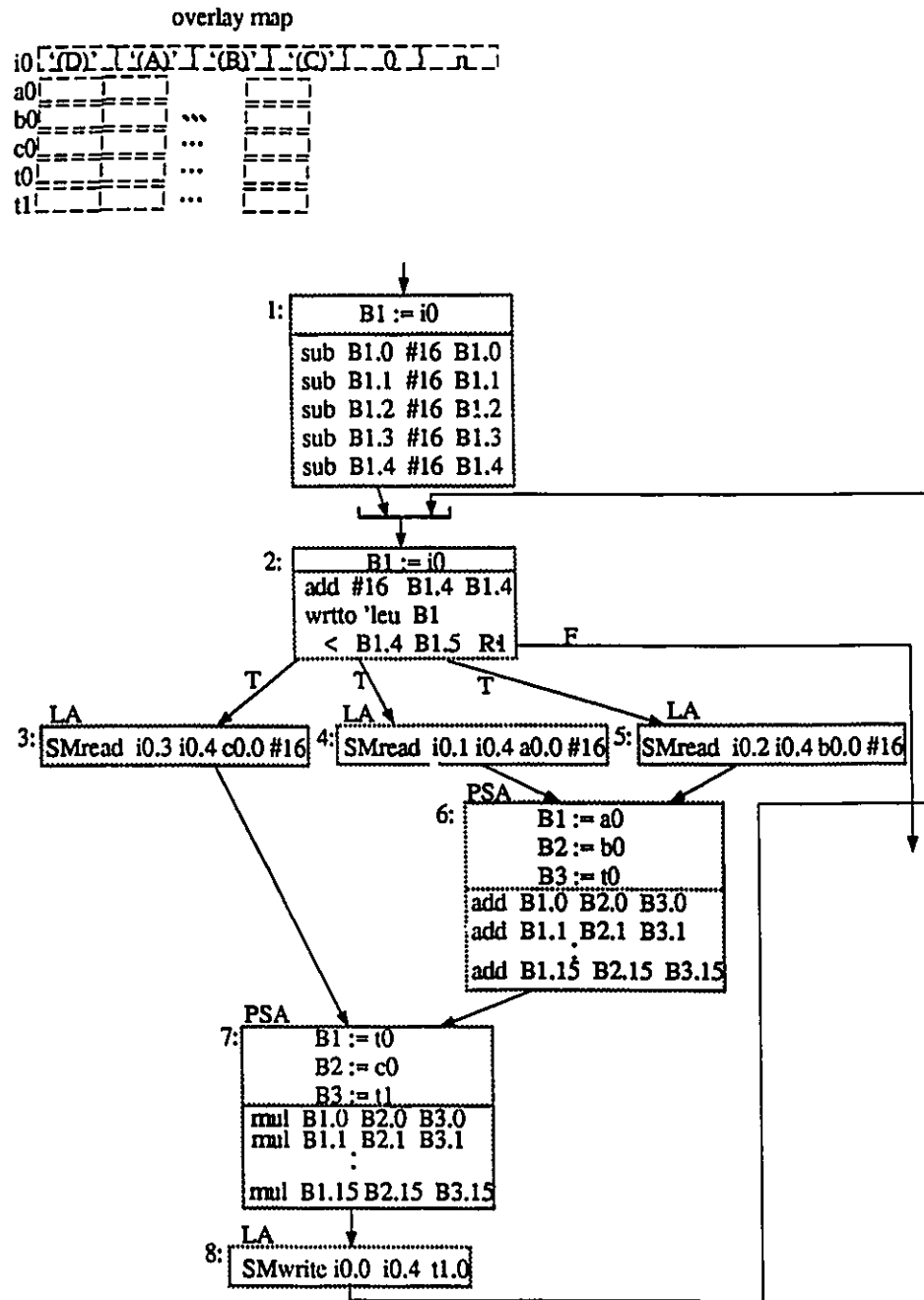


Figure 7.10: The parallelized ('vectorized') super-actor graph.

concurrently executing vector operations (parallel super-actors) did not seem possible. The loop body can only be re-activated when the **SMwrite** L-actor (actor 8) has issued a signal to the switch super-actor (actor 2) through the virtual merge node. To get the 'chaining' effect, where blocks of data can be loaded and stored while other parallel super-actors can execute simultaneously, we can apply dataflow software pipelining techniques (appendix A) to the parallelized loops.

To pipeline the above example, id nodes are first introduced to balance the software pipeline (actors 9, 10, and 11 in fig. 7.11; fig. 7.12 shows the overlay map). The three id nodes are used to propagate the offset value for arrays *C* and *D*. The task of sending an acknowledgement arc from the loop-body has been removed from the last actor (the **SMwrite** long-latency actor—actor 8) to the initial actors of the loop-body, that is actors which receive signals from the switch super-actor in the pred-body (actors 9, 4, and 5). Note that two extra merge nodes are introduced to handle the acknowledgement signals from the three initial super-actors in the loop-body. Also, the loop pre-processing actor (actor 1) must now signal three merge nodes instead of one. As for acknowledgement arcs, each super-actor within the loop-body must acknowledge its predecessor(s) to maintain the pipelining effect. Also, the switch super-actor must acknowledge the loop pre-processing actor upon termination of the loop.

By software pipelining and balancing the loop, the loop body now contains four stages (excluding the switch super-actor since it belongs in the pred-body). That is to say, this code structure can support an average of two loop iterations at any given time. This translates to a little more than 20 (41 instructions in the loop—including the instructions in the switch super-actor—divided by the two phases of a dataflow software pipe) simultaneously active instructions for concurrent execution. If further parallelism need be exposed, this software pipelined loop can be multiply instantiated such that each loop instance handles a block of elements of the array computation. For example, if the above example was to compute a 1000 elements of array *D*, then four loop instances can be created where the first instance handles the first 250 elements, the next instance, the next 250 elements, and so on.

The astute reader may note that a done signal can be sent to the successors of the loop before the **SMwrite** actually writes the last sixteen elements into array *D*. To correct this

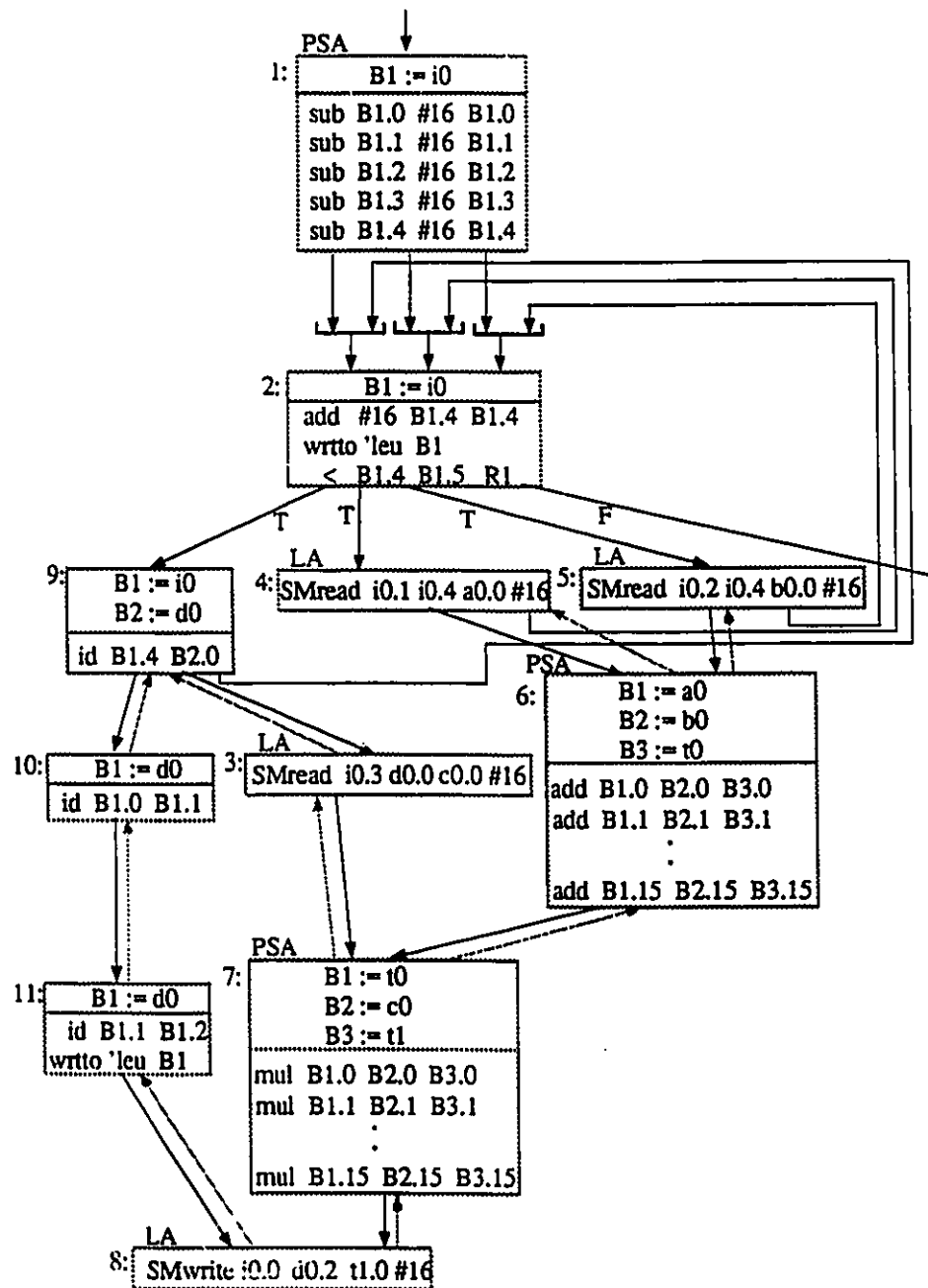


Figure 7.11: A super-actor graph representing a 'vectorized' and software-pipelined loop.

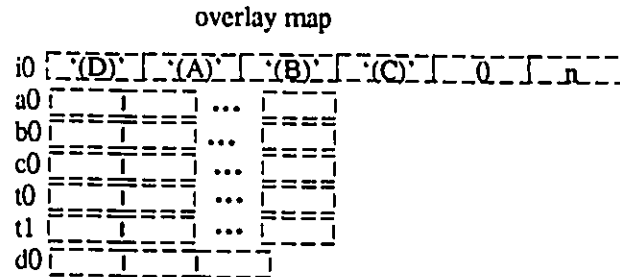


Figure 7.12: The overlay map of the previous SA graph.

situation, the **SMwrite** actor can trigger an added switch super-actor (similar to the one in the pred-body) which increments its own index variable (not the one in *i0.4*) and checks if it is greater than *n*. If so, it will be responsible for triggering the successors of the loop. However, we have left the loop as it currently appears in figure 7.11 because if another software pipelined loop is the consumer of array *D*, then the loop in the figure can be readily integrated with the consumer loop, i.e., collective loop fusion [119] can be performed.

Identifying Fast-Path Candidate Super-Actors This optimization process requires little or no aid from the programmer. Basically, the strategy for identifying fast-path candidate actors is: for each parallel or sequential super-actor within loop encapsulators, check if the required memory blocks containing the operands have to be loaded mandatorily (e.g., if a block was written by an L-actor, then the block must be loaded before a super-actor can use it), if not, then label the super-actor as a fast-path candidate. For example, let us look at the above software-pipelined loop body. The switch super-actor and the three id nodes (actors 2, 9, 10, and 11) can all be labeled as fast-path candidates because for every iteration, they access the same data blocks. The parallel super-actors performing the addition of elements of *A* to *B*, and the one performing the multiplication of elements of *C* to the results of the addition (actors 6 and 7) cannot be labeled as fast-path candidates since the producers of the operands for those super-actors are long-latency actors. Even if the **SMread** actors

(actors 3, 4 and 5) were converted to support-actors, the addition and multiplication parallel super-actors cannot be considered as fast-path candidates because the support-actors will logically change the operand lines the super-actors access (by changing the pointer values which indicate which memory blocks the parallel super-actors should be operating upon), and that change can only be reflected physically in the d-R-cache if an activation of the super-actor is pre-processed by the R-cache loader. What the fast-path mechanism allows is the reuse of the same d-R-cache lines which the last activation of the super-actor used. A super-actor which has its operand lines changed for each activation cannot use the fast-path mechanism since the d-R-cache lines containing the operands from the previous activation is not valid for the current activation.

7.4 The Translator

The translation phase, i.e., creating a SAMAL (Super-Actor Machine Assembly Language, see appendix C) program,¹⁵ is a straightforward process once the super-actor program is produced. The first task of the translator is to assign the constants in the constants table to data blocks. (To minimize the copying of constants from one data block to another—this can arise when a super-actor requires constants residing in multiple data blocks—constants accessed by a particular function should be grouped together.) Next, the translator visits each SA graph in the super-actor program and collects information for function applications. For each function, the size of the overlay must be determined along with the list of actors to notify when the function is invoked (from the signal arcs of top nodes). The size of the overlay is determined by the number data blocks used in the function, excluding the data blocks containing the constants. The list of actors to notify contains the ids of actors pointed to by the virtual *top* node of the function definition. Furthermore, for each apply instruction in the super-actor graph, information is collected concerning the placement of return values in the caller's overlay. A single offset value from the caller's overlay base address can be used for the location of return values (the return values are stored in a contiguous area). The list of actor(s) to notify once values are returned from a function

¹⁵The reader may want to briefly glance at appendix C before proceeding.

call are indicated by the signal arcs emanating from the L-actor. If there are more than one actor to notify, then one is chosen at random to be signaled by the apply L-actor, and the other actors must be signaled by the chosen one. Once the function invocation information is gathered, the generation of the super-actor program can be performed by traversing the super-actor graphs and producing the corresponding SAMAL code for each actor and its contained instructions. (The actual ids of actors to signal for apply and return L-actors are determined at the assembler stage. Also, the location of the first enable count value in a function overlay—required when initializing an overlay—is also determined in the assembler.)

For a sequential or parallel super-actor, the translator must perform the following tasks:

- Blocks of operand data which must be loaded into the d-R-cache are those produced by long-latency actors. Thus, those data block descriptors must have an 'L' prefix to indicate a mandatory load.
- The positions of descriptors (block assignments—load instructions for the R-cache loader) within the data block descriptor array are fixed so that an operand/result field of an instruction can specify which block to access by specifying an integer value which indicates the element in the data block descriptor array.
- For a sequential super-actors, results of instructions which are not visible outside of the actor will be deposited in registers. Subsequent instructions which use those results will access the appropriate register. If the number of temporary registers in the register set is not enough, then a reordering of instructions within the actor is necessary. If that is not sufficient, then extra space in the block which stores the results of the super-actor must be reserved; these extra spaces will act as temporary registers since the access times to temporary registers or to data R-cache are the same.

For support-actors, the translator must perform the following tasks:

1. insert load instructions to bring the necessary values into registers. The values operated by the support-actor will be specified by an offset from the overlay's base address (register 0);

2. insert instructions to perform the required operations while being aware of the delayed loads and the latency of the integer multiply instruction (e.g., the support-actor execution pipe in our simulations assumes a three cycle latency); and
3. copy back the necessary values from registers to memory.

Note that the minimum number of instructions in a support-actor must be greater than the number of stages in the support-actor execution pipe (see section 6.2.4), thus the support-actor might have to be padded with **nops**.

For long-latency actors, the translator performs a direct translation where the operands can be specified as offsets from the overlay's base address (register 0).

7.4.1 An Example

Let us look at the example in figure 7.6 and generate the corresponding SAMAL code representation. Let's assume that during the location assignment phase of the partitioner, it was decided that the data block containing the arguments (including the caller's base address and the id of the apply actor which invoked the function) should be grouped with the result block of L-actor *a* because those values are modified by L-actors, and that the output of *c* or *d* will share the same block as the output of actor *e*. And let us say that a data block contains sixteen words. From the data block optimizer, an overlay map for function 'g' is laid out and is shown in figure 7.13. (This example shows that a lot of memory space is wasted, but in an actual compilation, function 'g' would most likely be in-lined for efficiency purposes.) The SAMAL code for the example is listed below; line numbers are not part of the SAMAL syntax.

```

1  Func g #2
2      (TN (U: >a)
3      LA a (U: >b) Apply #'f &4 &6
4      SA S b (U: ) (T: >c) (F: >d) (OL&0)
5      (= rcl.6 #'T reg1)
```

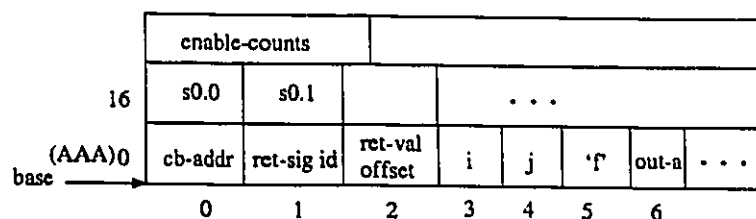


Figure 7.13: The overlay map for the partitioning example in figure 7.6.

```

6      SA S c (U: >m) (OL&0 R&16)
7      (add #1 rc1.3 reg1
8      mul #2 reg1 rc2.0)
9      SA S d (U: >m) (OL&0 R&16)
10     (sub rc1.3 #1 rc2.0)
11     MN m (U: >e)
12     SA S e (U: >h) (O&16 O&0)
13     (mul rc1.0 #2 reg1
14     add reg1 rc2.2 rc1.1
15     wrtto 'leu #1)
16     LA f (U: ) Return &17)

```

Line 1 is a directive stating that function 'g' with two arguments are to be defined. The brackets enclosing lines 2 to 16 lists the actors belonging to function 'g'. Line 2 describes the virtual top node of the function (directive 'TN') with an unconditional signal list '(U: >a)' specifying that a forward signal arc points to actor 'a'. The '>' signifies a forward signal arc. The actor on line 3 describes the L-actor *a* (the directive 'LA' indicates a long-latency actor) and the information '(U: >b)' indicates that a forward unlabeled signal arc emanates from the L-actor. The following information on that line specifies that it is an 'apply' L-actor which invokes function 'f' (specified by the operand, #f—in immediate mode), the next operand '&4' specifies the location of the argument—an offset value from the local overlay—and the third operand is another local overlay offset specifying the location for the return value. Line 4 specifies the sequential super-actor *b* ('SA' stands for super-actor and the following 'S' indicates sequential). From the specifications of the signal lists (the

information in brackets with either a 'U:', 'T:' or 'F:' as the first element), the reader may note that b is a switch super-actor. The next set of brackets contains the description for the data blocks to be loaded. An 'O' prefix specifies that it is an operand block, the 'L' indicates that it is mandatory to load it, and the value '&0' says that the overlay offset of the block is zero (block 'AAA'). On line 5 is the instruction of super-actor b —a relational instruction comparing the value in the sixth location of the first memory block¹⁶ which is loaded for the super-actor ('rc1.6') to the 'T' value and depositing the result into register 1. The other super-actors are similarly defined. The virtual merge node m is listed on line 11. Readers may refer to appendix C for the syntax of the actors, virtual nodes, etc.

7.5 The Assembler

The task of the assembler is to compute the weights of count signals; remove the merge nodes (the other type of virtual nodes, the top nodes, are simply deleted); assign locations for the constants, actor attributes, the actors' instructions, the actors' signal lists, and the enable count values of the actors; translate the SAMAL code to machine readable form; and fill in all of the remaining information once the locations of actor attributes, signal lists, etc. are known.

7.5.1 Calculating the Count Signal Weights

Conceptually, an enable count of an actor consists of two parts: an integer value, s_f , indicating how many (forward) signals are required to activate the actor and another integer value showing how many (backward) acknowledgement signals, s_a , are required to set the actor back to its initial state where it can be triggered for activation for the next set of input values. Notifications of incoming input values are indicated by forward signals, thus the enable count of an actor in its initial state should be equal to s_f . When an actor has

¹⁶Its descriptor is first in the list of data block descriptors, thus it is identified as the *first* memory block of the super-actor.

been activated, it can only be reactivated when it has received all of its acknowledgment signals (s_a) plus the signals indicating that a new set of input values have been computed (s_f). Therefore, the value used to reset the enable count, s_r of an actor should be equal to $s_f + s_a$. In our implementation of the SAM, we would like the reset values of all actors to be uniform so that function applications can be efficient (see Uniform Reset Value on page 139). Let us label this uniform reset value, S_r , and $S_r = S_f + S_a$ where S_f is the uniform initial count value and S_a is the uniform acknowledgement count value. For this to happen, the count signals (the forward and acknowledgement signals) must be weighted so that the machine code representation of the signal graph can mirror the actual structure of the super-actor graph. That is, for a super-actor with s_f input arcs,

$$\sum_{i=1}^{s_f} w_i = S_f$$

where w_i is the weight of input arc i . Similarly, the weights of acknowledgement arcs can be computed.

From the super-actor graphs we have shown thus far, we note that in general, (switch) super-actors do not have any acknowledgement arcs (except when dataflow software pipelining is employed). However, S_a is a (non-zero) positive integer value, so an actor must receive an acknowledgement signal in order that a uniform reset value can be used. To alleviate this problem, we can simply add a self-acknowledgement arc for an actor which does not have an acknowledgement arc. This was done for the benchmark programs we used in our simulations. In retrospect, we could have used two reset values instead of one: a *full reset value* (equal to $S_f + S_a$), and an *initial reset value* (equal to the initial count value, S_f). We can flag an actor as requiring a full reset value or an initial reset value whenever it needs to be reset, that is, an actor without any acknowledgement arc is flagged as requiring an initial reset value. The trade-off is less signal requirements at run time versus the expense of increased hardware logic and increased memory space for differentiating the reset value requirements of actors.

Let us now look at an example. Assume that S_f of all actors is eight, S_a equals seven and that the number of input edges incident on actor x is three, where actors a , b and c ¹⁷

¹⁷Actors a , b and c can be merge nodes, but we will see how weights to the output signals of the predecessors

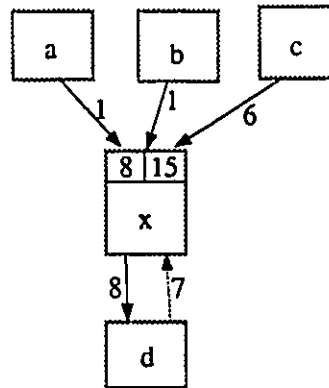


Figure 7.14: An example of weighted count signals.

the actors sending the forward signals. Lastly, actor d is the sole consumer of the result of x . If actor a 's count signal to x has a weight one, and b 's count signal to x is also weighted one, then c 's count signal to x should have a weight of six. Actor d 's acknowledgement count signal to x would be weighted seven. Figure 7.14 illustrates the example. The top left corner of actor x shows the enable count when the actor is in its initial state and the top right corner indicates the reset value. The signal arcs are weighted accordingly.

7.5.2 Removing Merge Nodes

Merge nodes are removed before the actual machine code is generated, i.e., the merge nodes will be encoded in the actors' signal lists and their reset counts. First the weights of output arcs of merge nodes are computed just as if they were actors. An output arc of an actor (merge node) which terminates on a merge node m 'inherits' the weights of the output arcs of m . For example, in figure 7.15, actor a signals actor x and y through m , thus the output arc of a inherits the weights of the signals to x (weight of four) and y (weight of three). In x 's signal list will appear two entries, an id of actor x with a weight of four and an id of actor of merge nodes can be assigned in the next section.

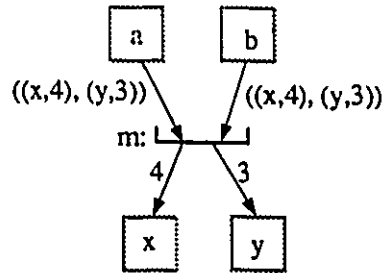


Figure 7.15: Encoding merge nodes into signal lists.

y with weight three. The same goes for actor b . This scheme can be applied recursively to cases where merge nodes output to other merge nodes.

An optimization technique to reduce signals through merge nodes can be applied for cases where an actor (actor x) signals multiple merge nodes and those merge nodes send signals to one actor y . Instead of having multiple entries of actor y in the signal list of x , they can be combined, i.e., only one entry to y need exist with a weight equal to the sum of the weights of the former signals to y .

7.5.3 Packing the Attributes, Instructions, etc.

As we have mentioned in chapter 6, actor attributes, instructions for sequential and parallel super-actors, instructions for support-actors, instructions for L-actors, and the actors' signal keys and signal lists should be put into separate contiguous memory space. This can be simply performed by collecting the machine code for all the actor attributes from all actors, the machine code for all the instructions of sequential and parallel super-actors, etc. together, determining the space required for each type of information, and then organizing them according to the program segment map (shown in figure 6.12) while respecting the boundaries for each type of information (as specified by the line sizes of various caches).

7.6 Summary

In this chapter, we have outlined a scheme for transforming an encapsulated dataflow graph into a well-formed SA graph. We have shown how super-actors can be generated and have brought forward many optimization issues which are to be investigated in future research. In particular, we have discussed some machine specific constraints on the partitioning algorithm, such as making values visible to the LEU and support-actor execution pipe, limiting the number of instructions per super-actor, and creating support-actors. We also examined some partitioning optimizations such as producing parallelized loops, and dataflow software-pipelined code. Moreover, we have described how a SA graph can be expressed as SAMAL code and how machine code can be generated which adheres to the rules as specified by the architecture.

Issues for future research include: having multiple long-latency instructions within an L-actor, sharing of memory blocks (R-cache lines) between super-actors (when a super-actor does not fully utilize all the locations within a block), the placement of results in memory blocks such that a super-actor can request the minimum number of memory blocks containing its operands and results, the sharing of the same instruction code by different super-actors (for example, having a sequence of instructions for representing a 'vector-add' operation), and acknowledgement arc insertion rules for dataflow software pipelining.

Chapter 8

Simulations

A major goal of this dissertation is to examine the performance of one processing element of the Super-Actor Machine, and a parallel objective is to determine whether the SAM can address the issues of tolerating local memory latencies and fine-grain synchronization costs. To achieve these objectives, we demonstrate the power of the SAM through simulations with a detailed simulator of one PE of the SAM.

The successes of simulation-guided system development, like that of the IBM RS/6000 [91], have added weight to the notion of experimentation and validation via architectural simulations. This has reinforced our commitment to using this methodology in researching novel architectures. For our research of the Super-Actor Machine, a detailed simulator was written in Common Lisp with Flavors. With a detailed simulator,

- the desired functionality of the SAM can be verified,
- the SAM's performance can be gauged,
- the innerworkings of the SAM better understood, and
- further enhancements of the architecture can be guided via simulation experiments.

To gauge the performance of the SAM, a set of small loop kernels which are common in scientific numerical applications were selected for use as benchmark programs. The major reason is that a compiler is not yet available and the only feasible solution at this time is to simulate small hand-coded benchmarks; other efforts in designing novel architectures also face this reality (see page 48 of [62]). However, these loop kernels can be used to simulate real scientific applications by chaining multiple loop kernels into producer-consumer relations. By restricting ourselves to small loop kernels, and chaining these kernels to form synthetic benchmark programs, the performance of the SAM can be better understood due to the simple nature of these programs.

In the simulation studies, we concentrate on the efficiency aspects of one PE. We compare the performance of a 1-PE SAM to a state-of-the-art superscalar uniprocessor machine, the IBM RS/6000. The investigation of the SAM's multiprocessing capabilities will be left to future research.

8.1 The Simulated Architecture

In the simulations, some of the design parameters of the SAM architecture were arrived at by examining similar parameters in existing machines, while others were based on rough calculations of the requirements for effective processing. This section lists the default parameters of the simulated SAM architecture. The changes to these parameters will be indicated when different experiments are performed on variants of the basic SAM configuration.

The local main memory is made up of sixteen banks with access times of six machine cycles. A memory controller is used to regulate accesses to them. Addresses are interleaved amongst the banks and each bank can service a request independently from the others.

The signal processor and the enable controller of the ASU are pipelined functional blocks with a pipe beat of one machine cycle. The signal processor is a three stage pipe and so is the enable controller. A 1K word 4-way set-associative cache with a line size of

eight words is used to buffer requests for signal lists from the signal processor to the main memory. The enable controller accesses memory through a 512 word 4-way set-associative cache with a line size of one word. An enable count is only four bits, so one word contains the enable counts of eight actors.

In the APU, the fast-path switcher (see fig. 6.7) takes one cycle. The fetching of actor attributes in the APU goes through a 1K word 4-way set-associative cache with a line size of eight words. The fetcher/router can take one to three cycles to process an enabled actor—depending on the number of attributes fetched. The R-cache loader can load an R-cache line in a maximum of eight cycles (one to form the address, six for the access and one to load) if no write-back is required. A write-back of a dirty line would require an extra seven cycles. Reserving a line takes only one cycle. Requests from the R-cache loader to the i-R-cache and d-R-cache are performed in parallel. The fast-path candidate checker requires two cycles to make an enabled fast-path candidate super-actor ready. The actor-cache in the checker has space for 64 entries, as calculated from the d-R-cache and i-R-cache sizes, see below. The PSA/SSA ready queue has space for $(64 - 16)$ ready super-actors.

The support-actor execution pipe is a basic RISC pipe with a pipe beat of one cycle, i.e., there are an instruction fetch, operand fetch, execution and result store stages. In the execution stage, there are two sub-pipes, one to handle add-type instructions and the other, multiply instructions. An integer add-type instruction takes one cycle while an integer multiply goes through a three stage pipe. The i-cache and d-cache are both 1K words 4-way set-associative with a line size of four words.

There are sixteen available physical contexts in the SEU, so there are sixteen sets of registers where each set contains eight 32-bit registers (eight temporary registers is overkill, but for the following experiments, a large number of registers made our task of manually allocating registers within a sequential super-actor easier). We chose the value of sixteen physical contexts because the execution pipe has eleven stages through its longest path; if the contexts contained only sequential super-actors, then we would require a minimum of eleven active super-actors to keep the pipe fully busy. The execution pipeline has a pipe beat of one machine cycle and the instruction issuer is also pipelined with a cycle time of

one machine cycle. The floating point add, multiply and approximate reciprocal pipes are six stages long with a pipe beat of one cycle, and the integer pipe is one stage long. Fetch and stores from the register set or register-cache each take one cycle. The i-R-cache is 1K words with sixteen words per line. This implies that a super-actor can only contain a maximum of sixteen instructions. Arvind and his associates [9] have found that the average grouping of dataflow actors are of size four, so sixteen should be plenty.¹ Moreover, as we shall see via the experiments, sixteen instructions per super-actor is very beneficial when super-actors must contain enough instructions to keep the execution unit busy. The memory portion (registers/cache lines) of the i-R-cache is two-ported, one for the read access by the execution pipeline and the other, for the write access by the loader/storer (see fig. 6.3). However, the tag portion is only one-ported so that tags cannot be simultaneously altered. The d-R-cache is also 1K words with four words per line and the path to main memory is four words wide. Each super-actor is allowed a maximum of four lines. The memory portion of the d-R-cache is four-ported, two for operand reads, one for result store, and one for the loader/storer. Again, the tag portion is only one-ported.

The LEU was not modeled since the preliminary experiments were only used to investigate the impact of the register-caches in a PE of the Super-Actor Machine.

8.2 The Test Programs

For this study, we have hand-coded four small benchmark programs. The four benchmark programs are: SAXPY, SAXPBYP, SAXPY3, and Lawrence Livermore Loop 1 [81]. SAXPBYP is the same FOR-loop construct as SAXPY except the expression is $a * X[i] + b * Y[i] + c$ instead of $a * X[i] + Y[i]$. The SA graph for SAXPY appears in figure 8.1 in which the loop is unrolled four times and is dataflow software pipelined (n is assumed to be a multiple of four). In the graph, the loop pre-processing actor (actor 1) prepares the inputs to the loop and copies the array addresses to the d-cache of the support-actor execution

¹This does not mean that the rest of the cache line goes to waste. In fact, other super-actors can share the same i-cache-line, the only requirement being that an assembler or compiler must handle the arrangement of instructions into contiguous blocks which are aligned on 16-word boundaries.

pipe. Two support-actors (actors 4 and 5) are responsible for calculating the 4-element block offsets of arrays X , Y and Z for actors 6 and 7. Actor 6 performs the multiply operation on four elements of X and actor 7 takes those results and adds them to four elements of Y and deposits the results in Z . SAXPY3 is a synthesized loop kernel which consists of two SAXPYs producing the input arrays for the third SAXPY—this simulates a producer consumer relation (fig. 8.2). The SAXPYs were inlined and loop fusion [119] was applied so that the loop overhead (the processing of the induction variables) can be reduced to one third of the original amount. This is not the only benefit of loop fusion, in our case, fusing loops also yields increased parallelism to be exploited by the underlying machine.

For all four benchmarks, the loops were unrolled four times so that parallel super-actors can be formed by aggregating four identical operators in the loop.² They were unrolled four times because the base design of the SAM which was simulated has a d-R-cache line size of four words. The index sequencing is handled by a switch super-actor which can either trigger super-actors in the loop body or exit when it is finished. The arrays which the loops process are stored locally in main memory and support-actors are used to perform address calculations for the super-actors in the loop body. The switch super-actor has six instructions (the extra instructions are for calculating the remaining elements to be computed when n is not a multiple of four), while the parallel super-actors in the loop body have four instructions each. Support-actors have an average of seven instructions each. Appendix D contains the machine code for the four benchmark programs.

Dataflow software pipelining[48] was utilized to increase the amount of exposed parallelism in the programs. With dataflow software pipelining, a code body for SAXPBYPYC was reconstructed with five stages, thus handling 2.5 simultaneous iterations and exposing more parallelism, while a code body for Loop1 was reconstructed with six stages. However, SAXPY could not really benefit from software pipelining due to its small loop body—only three stages in the loop body were produced when it was software pipelined. For the SAXPY3 benchmark, software pipelining was only applied to the individual SAXPYs, and was not applied to the whole SAXPY3, i.e., id nodes were not used to buffer the input of

²This is the vectorization technique as mentioned in section 7.3.2.

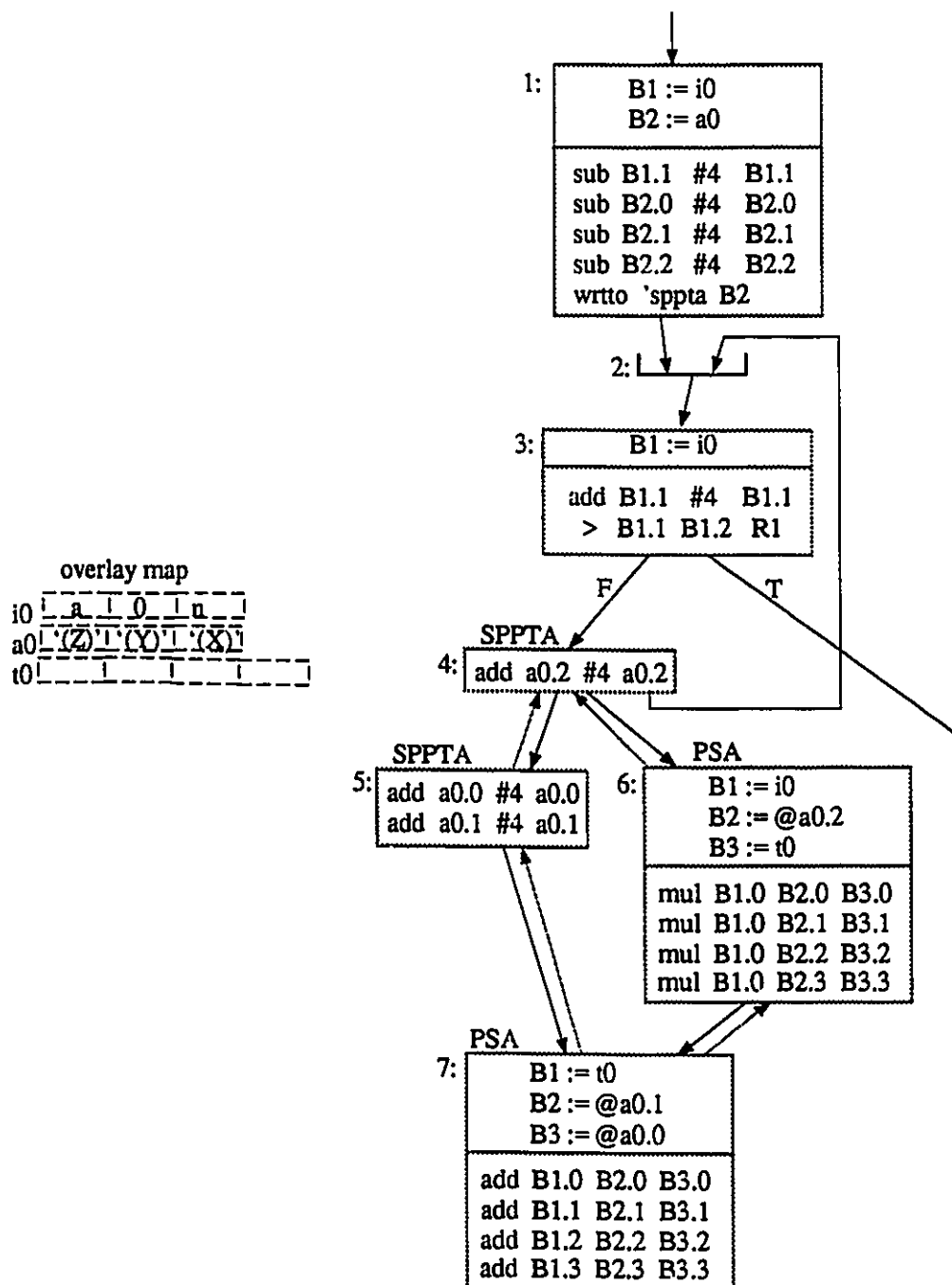


Figure 8.1: SA graph of unrolled-4 version of SAXPY.


```

let
  A := SAXPY (a, X, Y)
  B := SAXPY (a1, X1, Y1)
in
  SAXPY (a2, A, B)

```

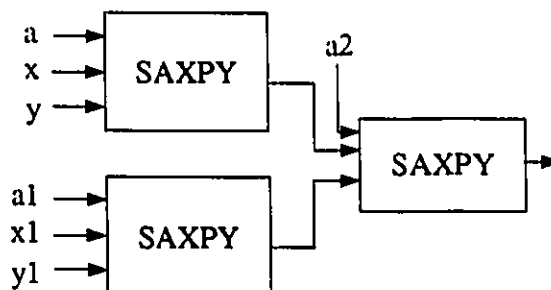


Figure 8.2: SAXPY3.

the B elements to the consumer SAXPY. The reason is that this benchmark is supposed to simulate producer consumer relations in scientific code where, in general, functions like SAXPY are optimized separately and then later inlined.

8.2.1 Simulation Results

Two versions of each program were written: one which only had one loop that iterated from element 1 to 1200, and the other version with four simultaneous loops, i.e., each loop was invoked in parallel where a loop iterated through 300 elements. The results are shown in table 8.1. The reader should note that several operations can be issued each cycle besides an ALU operation in the SEU, and the utilization rate shown in table 8.1 does not reflect the processing of those other operations. The speedup factor was calculated by dividing the execution time for the 1-loop version by that of the 4-loop version. From the table, one can conclude that more parallelism results in a higher utilization of the SEU and the more parallelism the compiler exposes in a program, the faster the execution due to the opportunity of overlapping memory loads to the R-cache with the processing in the SEU (well, at least until the SEU is 100% utilized). The varying and non-linear speedups where the SEU is not 100% utilized indicate that either not enough parallelism was exposed and/or the exposed parallelism was not structured properly. In figure 8.3, we show the execution pipe utilization for varying numbers of parallel loops and unrolling factors for

Benchmark	1 Loop		4 Loops		
	execution time (cycles)	SEU utilization	execution time (cycles)	SEU utilization	Speedup
SAXPY	30988	12%	9231	42%	3.4
SAXPBYPC	45490	14%	11348	56%	4.0
Loop1	33648	22%	13354	56%	2.5
SAXPY3	41819	20%	14376	58%	2.9

Table 8.1: Results for SAXPY, SAXPBYPC, SAXPY3, and Loop1.

the SAXPY benchmark.³ For the unrolling factor of eight and sixteen, a parallel super-actor can execute eight and sixteen instructions respectively. Accordingly, the d-R-cache line size was increased to eight and sixteen words for this experiment. The SAXPY version with eight parallel loops and unrolling factor of sixteen actually sustains 99% execution pipe utilization when start-up and wind-down phases are discounted (fig. 8.4).

Tolerating Increased Local Memory Latencies

One advantage of processor architectures which support multiple threads of computations is their ability to tolerate increases in global memory latencies [73]. In this section, we investigate the SAM's ability to tolerate local memory latencies. Figure 8.5 plots the sustained execution pipe utilization and the completion time for the 8-loop unrolled-16 version of SAXPY3 and SAXPY. Since SAXPY3 has more exposed parallelism, it can sustain the 100% SEU utilization until the delay for an R-cache load is sixteen cycles, whereas SAXPY can only sustain it till an R-cache load of ten cycles. For von Neumann machines which rely on a conventional cache, the utilization of the execution pipe would diminish proportionately with respect to the increase in average memory access times; instruction level parallelism cannot be exploited to hide the local memory latencies.

³Currently, a simple performance model of the SAM is being formalized and preliminary analysis correlates well with the curves as shown.

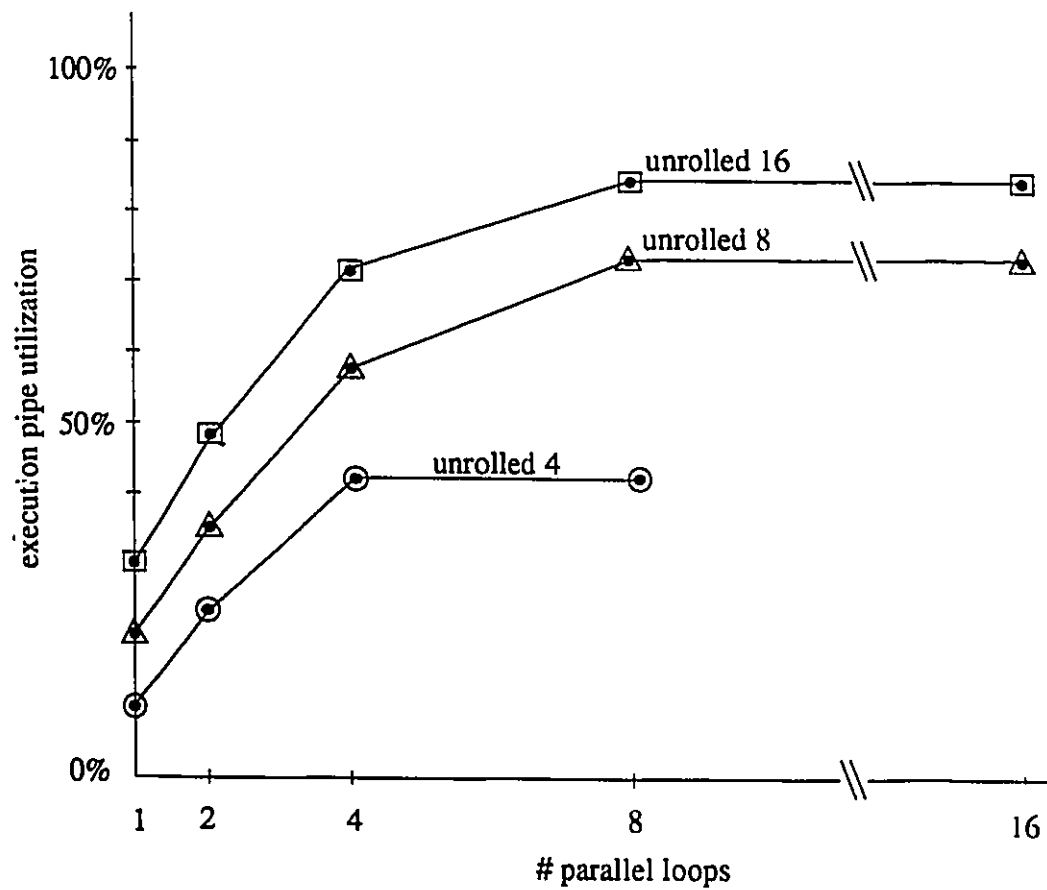


Figure 8.3: Execution pipe utilization rate for different versions of SAXPY.

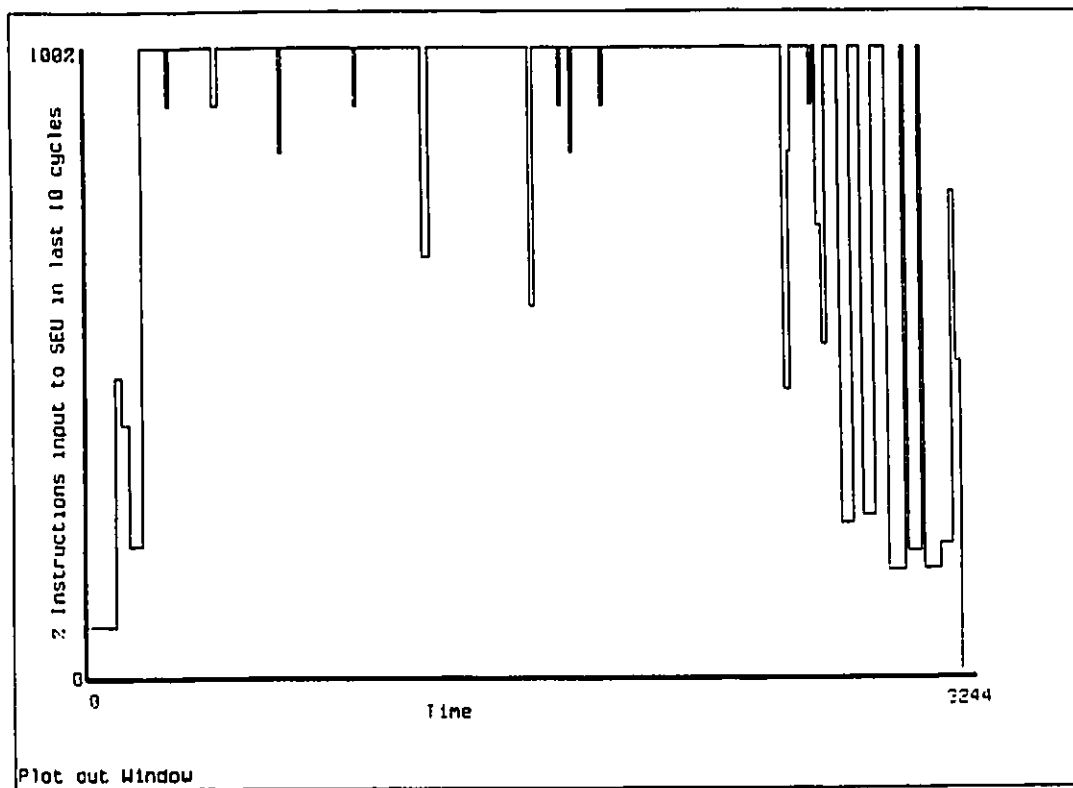


Figure 8.4: Execution profile for the 8-loop unrolled-16 version of SAXPY.

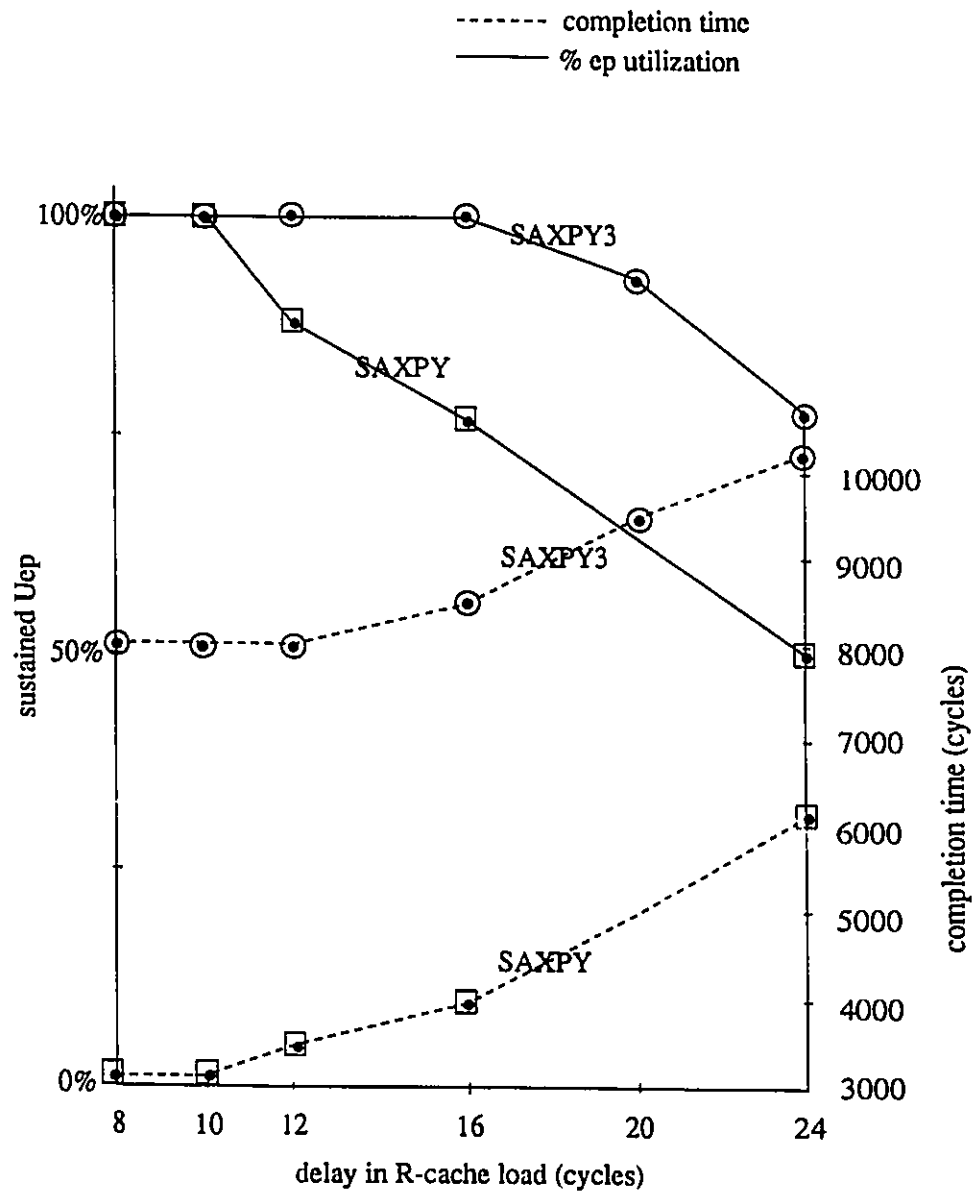


Figure 8.5: Varying memory access times for the 8-loop unrolled-16 version of SAXPY3.

max. number active super-actors	size of d-R-cache (Words)	sustained U_{ep}
64	4K	99%
32	2K	99%
16	1K	99%

Table 8.2: Varying the maximum number of active super-actors.

Efficient Utilization of High-Speed Memory

Another experiment was performed to examine the effects of restricted resources on the performance of the SAM, namely, the size of the R-caches. The original size of the R-caches allowed for 64 active super-actors, and in this experiment, the maximum number of active super-actors was varied from 16 to 64,⁴ i.e., the size of the d-R-cache was varied from 1K words to 4K words for a d-R-cache line size of sixteen. The results of this experiment are shown in table 8.2 for the SAXPY version of eight loops and unrolling factor of sixteen. In each case, SAXPY was used to calculate 1152 elements. If a set of resources (registers, etc.) must be reserved for every possibly active super-actor, then this version of SAXPY would require 24 sets, but as we can see, the SAM can make do with sixteen. This result clearly demonstrates that though a super-actor may be enabled for execution (their data are logically generated), but not ready (because its required data are not in fast memory), it will not hinder the execution throughput by ‘hogging’ some resource in the SEU and preventing other super-actors which are truly ready from executing. Other multi-threaded architectures such as the HEP, the future Tera computer, and APRIL (see chapter 9 for a review of these architectures) which allow suspended threads to keep their assigned resources, will not be able to share register sets as efficiently as the SAM. In a multiprocessor context, it is necessary to have many ready threads to hide the unavoidable latencies of interprocessor communications. The question is whether limited resources such as high-speed registers should be reserved for the lifetime of a thread; the reservation of these resources is necessary to minimize context-switching times.

⁴The minimum number of sixteen was used because the SEU pipe needed a minimum of eleven active super-actors to keep it filled if all the active super-actors were sequential.

8.2.2 A Performance Measure for Comparisons Between Various Architectures

To compare the performance of the SAM to other architectures, we need a performance measure which can reflect the degree of exploited fine grain parallelism resulting from the instruction level concurrency in the architecture, and the ability of the compiler in generating code to utilize such parallelism. To that end, we would like to introduce a performance measure called *FAB*, for Floating-point Arithmetic and logic operations per machine Beat.⁵ This measurement is computed by dividing the total number of *only* the floating-point arithmetic and logic operations by the total number of machine cycles required to execute a certain benchmark program. A machine cycle is defined as one pipe beat of the execution unit.

$$FAB = \frac{\text{total no. FP AL operations}}{\text{total no. machine cycles}}$$

Note that memory access operations, register transfer or other data movement operations, NOPs, etc. are not included in the number of floating-point ALU operations. Instead such instructions are reflected in the denominator of the above formula, and are deemed *overhead* or *supporting instructions*. The number of memory loads and stores for a given benchmark is a reflection of the idiosyncrasies of an architecture, and thus should not be included as 'useful' FP instructions.⁶ (For example, in VLIW [38] machines—Very Long Instruction Word machines—many register-to-register moves are used to move data to the proper execution unit.) More important, the programmer is only interested in how fast the machine can perform the FP operations as specified in the high-level algorithm (which does not include loads and stores).

Just as with MFLOPS and MIPS measurements, different benchmark programs will produce different FABs when run on the same architecture. Using the same set of benchmark programs, the quality of different compilers can also be compared using this measure.

⁵For non-numerical applications, one can use *IAB*—Integer Arithmetic/logic instructions per machine Beat—for measuring integer performance.

⁶The FAB measurement is similar to the inverse of the CPF (cycles per floating-point operation) measurement as described in [62] except that they include all floating-point instructions including loads and stores.

A higher FAB indicates a greater ability in utilizing the power of floating-point ALU units without disruption. Interpreted from a different angle, an architecture with a higher FAB rating can deliver the same MFLOPS performance with a more moderate hardware technology than one with a lower FAB. To summarize the advantages of the FAB measurement:

- it implicitly measures the ability of a processor architecture to overlap FP AL operations with support operations; thus, it also implicitly measures the degree of exploited fine-grain parallelism contributing to the overall floating-point performance;
- it measures the normalized floating-point performance of the processor architecture, i.e., how fast it can issue *useful* FP instructions per machine cycle;
- it is largely device technology independent; and
- it can also be used to measure the impact of compilers.

In the SAM, the sustained FAB rating for the 8-loop unrolled-16 version of SAXPY is 0.85 if the start-up and wind-down times are ignored. The maximum FAB rating for this version of SAXPY on the SAM is 0.86 (32 fp operations in every 37 SEU instructions). To put this value into perspective, the IBM RS/6000 produces a FAB rating of nearly one for the SAXPY loop.⁷ However, the RS/6000 uses the combined floating-point add-multiply instruction which issues two FP operations each cycle and only has a floating-point pipe of three stages. Many researchers have doubted whether a processor architecture which supports fine-grain processing (due to the overhead in supporting concurrently active threads) would ever be able to compete with the state-of-the-art von Neumann processors on a one-to-one basis for any class of applications. These results have shown that it is indeed possible. If one considers the limitations of a conventional cache for large scientific computations—the performance of the machine drops precipitously when the problem becomes too large for the cache [101, 106]—then the SAM will definitely have an advantage since the size of high-speed memory (the d-R-cache) does not affect the performance of the SAM as long as the application has enough exposed parallelisms (this was implicitly demonstrated in table 8.2).

⁷Unrolling does not benefit this benchmark program on the RS/6000 because the induction variable processing and array indexing are already overlapped.

8.3 Summary

In this chapter, we have performed some simulation experiments to examine the performance of the Super-Actor Machine for a class of loop kernels typical of scientific numerical applications. We have shown that the SAM can hide the local memory latencies from the execution unit if the kernels were structured in a fashion where inherent parallelism is exposed. It was also shown that the SAM can tolerate increased local memory latencies if there are enough exposed parallelisms. Furthermore, it was shown that limited high-speed memory (registers) can be used more effectively in the SAM than other multi-threaded machines which rely on ordinary registers. Lastly, a performance measure was introduced to show the effectiveness of the architecture in performing useful computations. With this measure, it was shown that the SAM can compete with a state-of-the-art superscalar processor for the class of loop kernels we have examined.

Chapter 9

Related Work

In this chapter, a non-exhaustive survey of other multi-threaded architectures is presented. It is very encouraging to note that an increasing number of architecture researchers are examining multi-threaded computing to answer the fundamental problems of von Neumann multiprocessing and pure dataflow computing. Moreover, we note that recent independent research have been conducted on applying the argument-fetching principle to multi-threaded architectures.

9.1 The Denelcor Heterogeneous Element Processor

The Denelcor HEP[105, 104] was the first commercial multi-threaded architecture which attempted to address the fundamental problems of multiprocessing. The HEP system could contain up to sixteen PEMs (*Process execution modules*) and 128 *data memory modules* (DMMs). The most interesting aspect of the HEP is its PEMs. In one PEM, multiple processes are interleaved into an eight-stage pipeline. As long as there are eight processes ready to execute, i.e., eight *Process Status Words* (PSWs) – each containing a program counter and other information – are residing in the *PSW queue*, the execution pipeline can be kept fully busy. A *task* is a collection of *processes* and each process represents a

thread of control. There are provisions for 128 active processes -- 64 user processes and 64 supervisor processes -- for each PEM.

For processes waiting on a memory access or synchronization event, there is a separate functional unit called the scheduler function unit (SFU) which is responsible for transferring data to and from the 2048 general-purpose registers and putting the waiting process back onto the PSW queue. Tolerating local memory latencies was not an issue in the HEP since it was a shared-memory machine where each PEM does not have any form of local main memory. Instead, that problem is shifted to the encompassing task of tolerating global memory latencies. The HEP relies on fast context-switching and a massive register file for tolerating memory latencies.

Fine-grain synchronization support in the HEP is provided in the form of *full/empty bits* on memory locations. Synchronizing instructions which wait on the full/empty bit will circulate in a queue in the SFU if the memory location it was synchronizing on was empty. Synchronization support is also found in the register file via full/empty bits on each register.

The limitations of HEP are as follows:

1. Each instruction within a sequential thread must incur the full eight stage latency of the pipe and slow down the execution of the program if it is highly sequential. (A pipelined von Neumann processor would only incur the eight-stage latency for the first instruction of the sequential thread. However, it would be quite difficult to keep the entire pipe filled most of the time.)
2. Limiting the number of active processes to 64 can present problems in some applications which have more than 64 active processes, especially when most of those active processes could be waiting on a synchronization event. While such active processes are waiting, they would still occupy the resources allocated to active user processes and thus prevent other processes not waiting on a synchronization event from executing.
3. Having only registers as fast memory penalizes computations with subscripted variables since allocating registers for them is difficult, thus all subscripted variable access must go through main memory unless indicated otherwise at compile time -- a challenging task as shown in [20].

4. A major problem with providing only binary semaphores (guard values in which at most two signals can be incident on an actor) for fine-grain synchronization is the extra cumulative synchronization costs when compared to synchronizations with general counting semaphores. Sarkar[98] shows that binary semaphores may require $O(N^2)$ operations as compared to $O(N)$ operations when general counting semaphores are used for synchronization in a dependence graph with N vertices. This implies that the execution unit must incur this added overhead and slow down the execution unnecessarily.
5. Finally, instructions which wait on a full bit in the register must circulate through the execution pipe and cause unwanted bubbles (see p. 427 of [6]).

9.2 Horizon and the Tera Computer

The demise of Denelcor Inc. did not spell the end of multi-threaded computers of HEP's lineage. In 1988, Burton Smith came out with a design for the Horizon Computer[110]. In the Horizon, a processing element (PE) can support a maximum of 128 *i-streams* – processes in the HEP. In this incarnation of the HEP, instructions in an *i-stream* can be executed sequentially instead of overlapped with other *i-streams*. A special look-ahead bit associated with each instruction is used to help in the pipelining of the instructions from a single *i-stream*. Only when there must be a switch to another *i-stream* – caused by synchronizing instructions or instructions depending on memory access instructions – will the PE execute instructions from another *i-stream*. Another major difference is that an instruction in an *i-stream* is now a Long Instruction Word (LIW) so that multiple instructions can be issued in one cycle. In one cycle, a PE can issue a memory accessing instruction, an arithmetic instruction and a control instruction (a branching instruction). Just as in the HEP, the Horizon relies on a huge register file to address the issue of tolerating memory latencies and utilizes full/empty bits for fine-grain synchronization.

To summarize the Horizon architecture, it has partially addressed the first problem of HEP where sequential threads are less penalized by the use of LIW which can decrease the completion time of sequential threads. The problem of supporting a limited number of active threads has been alleviated with the introduction of hardware support for 128 *i-streams*.

Also, registers no longer have full/empty bits, so instructions cannot circulate through the execution pipe waiting on a register to become filled. However, the other problems of HEP are still problems for the Horizon due to its exclusive use of a large register file for tolerating memory latencies and the use of full/empty bits (binary semaphores) for fine-grain synchronization.

Smith is currently leading the Tera Computer Co. and is in the progress of building the Tera Computer[7]. The Tera Computer is based on the Horizon, i.e., a multi-threaded architecture executing long instruction words. There are minor differences such as the introduction of a *look-ahead field* instead of a single look-ahead bit, etc. In any case, the mechanisms for tolerating memory latencies and its fine-grain synchronization support remain the same.

9.3 The Hybrid Dataflow/von Neumann Architecture

Robert Iannucci, from Arvind's group at MIT, proposed a design of the *Hybrid Dataflow/von Neumann Architecture*[73, 75]. The development of this architecture is being continued at IBM's T. J. Watson Research Center and has been renamed the *EMPIRE* Processor[74]. The architecture of the PE is similar to the HEP design in that one execution pipeline is to be shared by the *Scheduling Quantums* (SQs) – threads of control – supported in the PE. Two queues are used to hold enabled *Continuations* (similar to the PSW in the HEP) and suspended ones – continuations which are waiting for synchronization events or memory loads. A PE continuously executes instructions from one SQ until a *possibly suspensive* instruction is fetched. At that time, an instruction from another enabled SQ is fetched. In this manner, pipeline flushes due to possible suspensions are unnecessary.

To tolerate local memory latencies, a cache containing the frames¹ of recently executed SQs, the current active SQ, and that of the next SQ in the enabled queue is used as a high-speed buffer for main memory. A small register file can also be used to store temporary

¹A frame corresponds to the memory space allocated to an invocation of a function, where the function can have multiple SQs.

results for use by instructions within the same SQ. Note that the entire frame must be residing in the cache before an SQ of that particular frame can enter the execution pipe. This can result in unnecessary prefetching if only a few values from that frame are used. Another issue that is not addressed with this scheme is the support for often referenced subscripted variables – all subscripted variable are not cached and accesses must go to main memory, unless determined otherwise at compile time.

Fine-grain synchronization support is provided in the form of full/empty bits on memory locations.² Again, the spectre of inefficient synchronizations with binary semaphores shadows this architecture. A compounding effect is that when a suspensive instruction waiting on a synchronization event enters the execution pipe, it will cause a bubble if the event has not occurred. These ALU bubbles are not insignificant; an example of their impact is reported in a paper by Papadopoulos and Culler[90] for a machine based on the pure dataflow model of execution. However, a hybrid architecture should have less ALU bubbles due to less suspensive instructions. In contrast, the SAM supports the atomic execution of threads where no suspensive instructions are permitted in a super-actor.

9.4 P-RISC and *T

Another spin-off architecture from Arvind's group is in the form of *P-RISC* – Parallel RISC – which was conceptualized by Rishiyur Nikhil and Arvind [84, 87]. The architecture of a processing element is basically a RISC pipeline with a queue for storing enabled instructions, and a port to I-Structure memory. A simple integer ALU is used in place of multiple function units commonly found in other multi-threaded machines, and its memory hierarchy consists of: local main memory, a high-speed cache used for caching a subset of frames,³ and a register file. The P-RISC can have an arbitrary number of active threads similar to Iannucci's machine. At the instruction set architecture level, they added “fork”, “join”, “start” and “loadc” instructions. The fork instructions are used to start other threads

²This is the *I-Structure* mechanism championed by members of Arvind's group.

³A frame here has the same definition as a frame in Iannucci's architecture, that is, the memory space for an invoked function.

and the join instructions are used to synchronize threads. The start instructions are used to create threads, and the loadc instructions for split-phase memory reads. I-Structures[16] are also supported by way of the "I-read" and "I-write" instructions.

Since the tolerance of local memory latencies and the fine-grain synchronization support are identical to Iannucci's hybrid architecture, Nikhil's machine will also face the same limitations. Other problems also exist, for example, an instruction can cause a cache miss in the P-RISC pipeline which can lead to a degradation in performance. Another limitation of this architecture has to do with its load/store nature. Whenever a thread starts executing, registers must be loaded, and whenever there is a possible suspensive instruction in the thread, all required temporary values in the registers must be copied back to main memory before the suspensive instruction can be processed. The last problem has to do with its simple RISC pipeline where the overhead of loads and stores can significantly degrade its performance on 'useful' computations. It would be interesting to investigate the possibility of supporting multiple instruction issuing capabilities to hide the overhead of loads and stores. Nevertheless, we believe Nikhil and Arvind are the first to propose a multi-threaded architecture based on minimal additions to a basic RISC pipeline.

The P-RISC effort has been combined with that of the Monsoon [89] in an architecture called the *T (pronounced "start") [86]. The project seeks to support multi-threading by using a Motorola 88110 superscalar RISC processor as the computing engine and added hardware to perform synchronizations among the threads. One may regard the hardware for synchronization, called the *synchronization processor* as the ISU in the McGill Dataflow Architecture and the 88110 as the IPU of the MDFA. The major difference between *T and its predecessors is that only one thread can be executed upon at any time in the 88110, i.e., instructions from active threads cannot be interleaved. The *T is similar in many respects to the USC Decoupled Architecture and the LGDG Architecture as reviewed below.

9.5 The EM-4

Another multi-threaded architecture which shares the same lineage as the EMPIRE and P-RISC is Sakai's EM-4[97, 96] which is being developed at the Electrotechnical Laboratory of Japan. Sigma-1[64, 121] is the precursor of EM-4 and the Sigma-1 is ETL's effort in building a machine to support Arvind's tagged-token dataflow model of execution. The EM-4 differs from the Sigma-1 in that it is designed to support symbolic computations instead of numeric-intensive applications, and it supports the execution of *Strongly-Connected Blocks* – a thread of execution – along with regular dataflow actors. The architecture of the EM-4 bears great resemblance to the tagged-token dataflow architectures[10], that is, the PE contains a single circular execution pipeline in which fine-grain synchronization is performed along with other arithmetic operations. When the PE encounters the first instruction of a strongly-connected block, the entire PE will only execute instructions of that block.

Tolerating local memory latencies is provided in the form of a small register file to be used by instructions within a strongly connected block. Instructions which must perform some synchronization are still hindered by the response time of local main memory; it is not clear whether any caching memory is used to buffer local main memory to the execution pipe. It would appear that unless the majority of the nodes in a dataflow graph can be included in some strongly-connected blocks, this architecture can suffer from the same limitations which face pure dataflow machines.

Fine-grain synchronization support is provided in the form of full/empty bits on memory locations. So the problems of binary semaphores and performing synchronizations within the execution pipeline are also present in this architecture.

9.6 APRIL

Anant Agarwal et al.[3] proposed a multi-threaded architecture based on a SPARC implementation. In this architecture, the main objective is to efficiently support sequential

computations and perform a rapid context-switch when a thread executes a remote memory request or synchronization event. It was decided that a PE (APRIL) of the ALEWIFE system should support 4 active threads because of the limitations in registers and the adverse impacts of multiple active threads on the cache performance. In the implementation, the register window facility of the SPARC processor is used for the register set of each active context and context-switching must be performed by a trap handler. Even so, the context-switch can be performed in about 10 cycles. A major problem of only supporting 4 active contexts per PE is what happens when all 4 contexts are waiting on a synchronization event? Swapping contexts to and from memory can be quite expensive. Also, with all RISC based implementations, its performance on floating point-intensive applications is weak.

This architecture tolerates local memory latencies with the use of the register set and a conventional cache between the execution unit and memory. Thus, it suffers from the same cache coherence problems as conventional von Neumann multiprocessors. Also, cache misses will lower its performance. Fine-grain synchronization support is performed via full/empty bits on memory locations, so the inefficiency of binary semaphores is also a problem.

9.7 A Modern “Static” Architecture

A multi-threaded architecture which shares the same parent [33] as the SAM architecture was proposed by Jack Dennis[30]. A processing element of the *Modern “Static” Architecture* resembles the Argument-Fetching Dataflow Architecture. The main difference in Dennis’ new machine is the support of threads of computations (called *instructions* in this architecture) by the combined instruction queue-execution system (the IPU of the MDFA—see section 3.2). In the proposal, the execution system can support up to 8 active threads and overlap the execution of instructions (called *sections*) in the execution pipe.

Local memory latencies are tolerated via the use of small register sets for each active thread, and a scheme of employing queues between the memory accessing stages in the

execution system and the interleaved memory. Controllers regulating the queues of the memory accessing stages, called *Memory Transaction Controllers*, are responsible for *elasticizing* the execution pipe so that instructions (sections) with different memory demands can “hop over” other pending sections, and execution stages do not get backed up as often from congestions in succeeding stages. Putting these controllers in perspective, they are like the parking store as mentioned in section 2.1, and such devices have their limitations. Moreover, as the processing speed of the execution pipe increases with respect to the memory accessing times, the queue sizes in the Memory Transaction Controllers must increase accordingly, which can possibly increase the latency of the execution system.

Fine-grain synchronization is supported by an instruction scheduler (the ISU of the MDFA) which is removed from the execution path, so for the synchronization aspect, this architecture will enjoy the same benefits as that of the SAM architecture.

9.8 The Decoupled Graph/Computation Architecture

At the University of Southern California, Evripidou and Gaudiot[37] proposed the *USC Decoupled Graph/Computation Architecture* based on a *Multilevel Data-Flow Execution Model*. This execution model proposes that a dataflow graph should now consist of three types of actors: scalar actors (the basic one instruction actor of dataflow), vector macro-actors, and compound macro-actors (or CMA) which are made up of scalar and/or vector instructions. The arcs between these actors serve to signal a particular actor when it can be executed, i.e., its data has been logically produced and it can proceed. This model and the super-actor execution model are very similar in the types of actors and the way actors are executed atomically. The major difference is that the actors in the multilevel data-flow execution model do not have a temporary state where the necessary data is prefetched to fast memory before they can be executed. The lack of this state can cause the underlying architecture to stall when the required data is not immediately accessible.

The proposed processing element architecture consists of a graph unit and computation unit—units which have the same function as the ASU in the SAM and the data processing

unit in the McGill Dataflow Architecture. The graph unit is responsible for processing acknowledgement signals (done signals in the SAM terminology) from the computation unit and send ready signals (fire signals in the SAM) to the computation unit when an actor or macro-actor can be executed. Queues are used to buffer signals between the graph and computation units. As in the SAM, data is not passed in tokens between the two main units; the computation unit simply deposits and fetches them from memory as required. The three main differences between this architecture and the SAM are the memory hierarchy employed for reducing memory latencies, the function units for executing the actors, and the mechanism for synchronizing and scheduling actors. The suggested memory hierarchy is a cache-based subsystem where a *Queue and Cache Controller* monitors the queues and tries to pre-fetch the *context-blocks* which contain the instances of the actors (the context-block corresponds to an overlay in the SAM terminology). This strategy is similar to the one employed in Iannucci's EMPIRE [73] and thus has similar drawbacks. Another difference has to do with the function units for processing executable actors; they have suggested one computation unit à la MDFA, whereas the SAM proposes heterogeneous function units—function units which are optimized for their own particular tasks. Lastly, the scheduling unit in the USC architecture still relies on a waiting-matching store since the *token relabeling* technique [52] is employed. (The relabeling scheme requires that tags of tokens be hierarchically created and deleted at run time, i.e., fields of bits are added to the tag when a sub-context (a context can be regarded as an instance of a code block) is created and tokens can be matched more than once. Thus it would be quite difficult to employ the Explicit Token Store technique as found in P-RISC[84] and Monsoon[89]).

9.9 The LGDG Architecture

Another architecture which bears resemblance to the MDFA is the *Large Grain Dataflow Graph Architecture* (LGDG Architecture) proposed by Dai and Giloi [26]. The premise of this machine is the LGDG computation model. This model consists of 'O-nodes' which operate on data, and 'C-nodes' which operate on the signals passed between the O-nodes. An O-node basically corresponds to the group of instructions within a super-actor and a

C-node corresponds roughly to an s-node in the MDFA/SAM. The operational semantics of this model is also similar to the super-actor execution model; the major difference is the lack of temporary states for priming the high-speed memory.

The LGDG architecture consists of a graph-level unit (the ASU in the SAM) and a node-level unit (the data processing unit of the MDFA). An O-node is executed atomically in a node-level unit and when it is finished, a dummy or boolean token is sent to the graph-level unit. A dummy or boolean token is the same as the done signal in the SAM; a dummy token is a done signal with a condition code of 'U' and a boolean token is a done signal with a condition code of 'T' or 'F'. To activate an O-node, a dummy token is sent to the node-level unit, thus this architecture is also based on the argument-fetching paradigm. The LGDG architecture is similar to the SAM, the major difference is that Dai and Giloi suggested that the node-level unit be composed of multiple homogeneous processors known as N-RISC processors (for non-branch RISC processor, i.e., no conditionals are handled by the processor). Since multiple processors are used to implement the node-level unit, then some cache coherency schemes must be employed if a cache-based memory hierarchy is used. Furthermore, there is no guarantee that an N-RISC processor will not stall due to memory accesses. The graph-level unit is very similar to the ASU in that a mechanism is used to decrement and check the guard values, instead of some waiting-matching hardware as proposed in the USC machine.

Chapter 10

Conclusion

This dissertation has documented our research efforts in addressing the problems of local memory latencies and synchronization overheads facing von Neumann and pure dataflow processors. We brought forth a design of a multi-threaded architecture called the *Super-Actor Machine* which incorporates the following architectural features

- a memory organization which can guarantee that all accesses from the execution unit be satisfied with high-speed memory, and
- an efficient thread scheduling mechanism which is loosely-coupled from the execution unit

so that the two problems can be effectively addressed. We have shown that a processing element of the Super-Actor Machine can tolerate local memory latencies and fine-grain synchronization overheads while sustaining 99% throughput of its execution unit.

To arrive at this juncture of our research, we started our work with a base architecture called the *McGill Dataflow Architecture*. This pure dataflow architecture is based on the concept of argument-fetching where results of instructions are stored in data memory and subsequent accesses by successor instructions simply fetch the required data from data memory. This entails a logical separation of the data processing aspects from instruction

scheduling, which in turn, naturally dictates a physical separation of the instruction execution unit and instruction scheduling unit at the implementation level. The advantages of the McGill Dataflow Architecture versus other proposed dataflow architectures based on the argument-flow principle were illustrated via analysis. Based on these advantages, the newly proposed multi-threaded architecture incorporates this concept of a loosely-coupled scheduling mechanism.

Through simulation studies, it was shown that pure dataflow machines with fine-grain scheduling required the scheduling mechanism to have a greater throughput than the instruction execution mechanism. This conclusion steered our research towards multi-threaded architectures where aggregates of one or more instructions are scheduled via the data-driven paradigm while instructions within an aggregate are scheduled via a simple instruction counter. This model of execution lessened the demands on the scheduling mechanism from having to have a greater throughput than the execution unit in order to keep it usefully busy.

In the course of researching multi-threaded architectures, the Super-Actor Machine was proposed and formally described. This architecture supports a new execution model in which an aggregate of instructions, called a *super-actor*, is first 'prepared' before it can be dispatched to the execution unit. This *Super-Actor Execution Model* stipulates that in the preparation phase of an aggregate, the aggregate must first be assured that all its necessary operands are in high-speed memory and that space for its results have been reserved in high-speed memory before any of its instructions are processed. A novel memory organization called a *register-cache* was proposed and detailed to support this new model. Furthermore, a preparation unit was also introduced into the circular pipeline so as to interface with the register-cache.

Compilation techniques for converting a well-formed dataflow graph into a graph of super-actors were also investigated. An algorithm, called SAGA, was proposed to generate deadlock-free and determinate super-actor graphs. Also, optimization techniques to generate effective codes such as dataflow software pipelining, 'vectorization', etc. were also examined in the context of super-actor graphs.

Simulation results from a detailed simulator were obtained, and it was clearly shown that given a sufficient amount of parallelism in the application program—we investigated loop kernels which are typical of scientific code—the Super-Actor Machine can sustain a 99% throughput of its execution unit while tolerating local memory latencies and fine-grain synchronization overheads. Moreover, extra instruction-level parallelism can be employed to tolerate the increased memory latencies anticipated in future generation processors.

10.1 Future Work

Much work remains in investigating the performance issues of the Super-Actor Machine. In this section, we list some topics for future research:

- simple performance models for analyzing the effectiveness of the Super-Actor Machine should be formally defined. These models will further our understanding of the mechanisms in the SAM.¹ This work might entail a fresh look at how architectural modules can be specified and how the properties of such modules can be merged to yield properties of the entire system.
- Techniques for generating code specific to the Super-Actor Machine have only been detailed in this dissertation and have not been implemented yet. Definitely, the next stages of this work will require the implementation of a code generator and the further investigation of different optimization strategies.
- The detailed simulator must be augmented with a Super-Actor Machine Assembly Language Assembler so that more involved simulation studies can be performed. This addition should be enhanced with debugging support facilities so that development times of SAMAL code can be minimized.
- The investigation of efficient multi-processor support should be performed, especially with regard to sending and receiving blocks of data. It is quite evident that the SAM will function effectively if blocks of data were operated on at a time (in fact, most architectures with cache-like memory which fetches blocks of data at a time have this property).

¹This work is a subject of our ongoing research [69].

Appendix A

Dataflow Software Pipelining

This code block takes as input two arrays A and B and produces another array X in a *monolithic* fashion¹, such that:

The technique of dataflow software pipelining involves the arrangement of a dataflow graph such that successive computations can follow each other through one copy of the code block. If we present a sequence of values to the inputs of a dataflow graph, these values can flow through the program in a pipelined fashion. For the static dataflow model of computation, software pipelining is essential in exploiting the parallelism within a loop body, and thus, is a necessary optimization for numerical scientific applications.

For example, the code block

```
X := for i in 1,n
      returns array of
      exp(exp(2*A[i], 2) + exp(2*B[i], 2), 2)
end for
```

which computes the expression

¹Monolithic fashion implies that a portion or the whole array is produced by one code block.

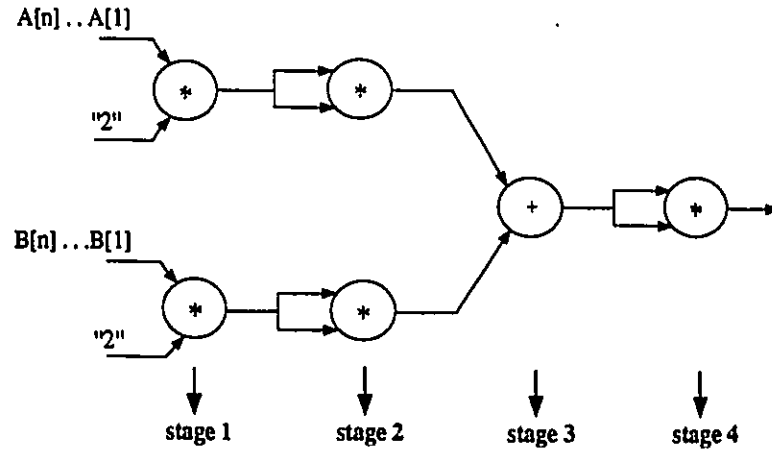


Figure A.1: An example of a dataflow software pipeline.

$$X[i] = ((2 * A[i])^2 + (2 * B[i])^2)^2, \forall i \in (1, n)$$

can be software pipelined (figure A.1). The array elements of the result array X can be evaluated in parallel because there are no data dependencies among them. That is, successive elements of the input array A and B will be fetched and fed into the dataflow graph, e.g., $A[1], A[2], \dots, A[n]$ and $B[1], B[2], \dots, B[n]$, and the computation proceeds in a pipelined fashion. This figure also illustrates the fine-grain parallelism that exists in the code block. Instructions that belong to the same stage can be executed in parallel, since there are no data dependencies among them. Moreover, during the execution of the program, multiple stages can be executed concurrently, e.g., stages 1 and 3 are enabled and can be executed in parallel and the same applies to stage 2 and stage 4. In this sense, there are basically two execution phases in a dataflow software pipeline: one phase to execute the actors in the odd numbered stages and the other phase for the even numbered phases. The power of fine-grain parallelism is displayed by programs that form a large pipeline in which many instructions in multiple stages can execute concurrently.

A.1 Dataflow Software Pipelining for Idealized Machines

Dataflow software pipelining was proposed as a model for structuring fine-grain parallelism [35, 83] and has been studied mostly under an idealized dataflow architecture model with infinite resources[41]². Here is a summary of some of the main results. A graph is *balanced* if every path from an input node to an output node contain exactly the same number of actors. A graph is said to be maximally pipelined if it is balanced. To achieve maximum pipelining, a basic technique (called *balancing*) is used to transform an unbalanced dataflow graph into a balanced graph by introducing FIFO buffers—or strings of identity (ID) actors—on certain arcs. An example of balancing a software pipe is shown in figure A.2. Figure A.2(a) shows an unbalanced software pipeline which basically has 2 stages, and (b) shows a balanced pipe where the introduction of two ID actors has created a four stage pipeline and increased the exposed parallelism. To optimally balance a graph, a minimum amount of buffering is introduced into the graph such that its execution can be fully pipelined. The technique of optimally balancing an acyclic dataflow graph can be formulated into certain linear programming problems which have efficient algorithmic solutions; algorithms which can be used to perform code optimization in a dataflow compiler [42].

²Recently, preliminary studies of dataflow software pipelined code on a more realistic architecture have been performed[45, 46, 49]. Those articles illustrate a new technique for structuring a software pipelined code block such that it makes a balanced use of the data processing and instruction scheduling mechanisms of the target architecture.

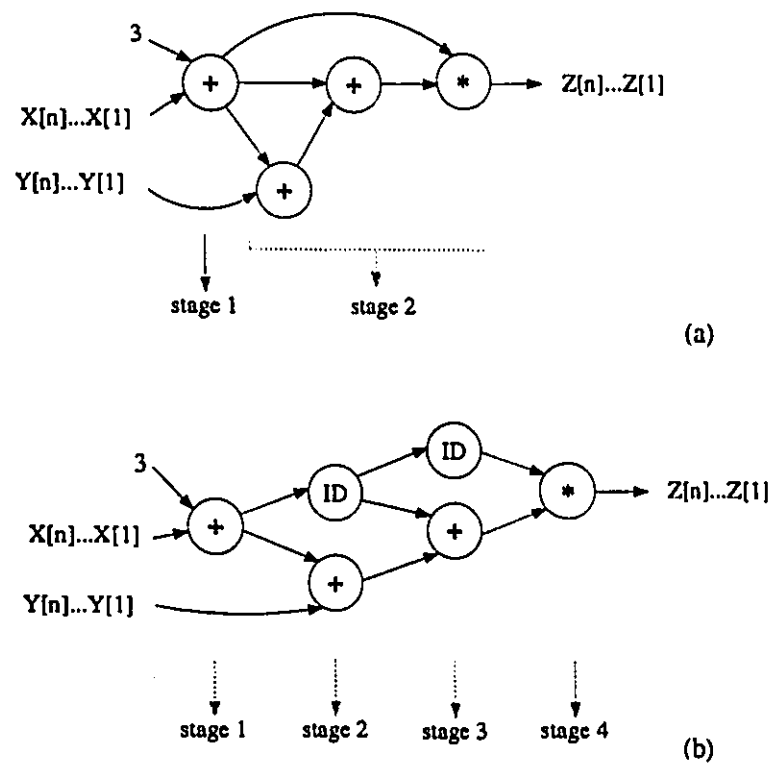


Figure A.2: Balancing a dataflow software pipeline.

Appendix B

Functions of the Advanced Machine Model

For each agent in the advanced machine model, there is a corresponding procedure it executes. But first, we introduce a new piece of information in the actor attribute memory: the actor type. We use the symbol, $AP.AT$, to denote the location containing the actor's type information. $AA[AP.AT]$ has three possible values: *par-sa*, *seq-sa*, and *other-actor* for parallel super-actors, sequential super-actors and other-actors (long-latency actors and support actors), respectively. Definitions of other symbols can be found in the tables on pages 85 and 103.

The *SA-activation* agent executes the *activate* procedure as listed in the intermediate machine model, the only difference is that it now operates on the Ready-SA, and Active-SA Pool.

```
procedure SA-activate ()
  do (forever)
    if (resources available) then
      from ready-SA pool, pick  $\langle OP, AP \rangle$ 
      allocate resources to  $\langle OP, AP \rangle$ 
      put  $\langle OP, AP \rangle$  into active-SA pool
```

enddo

The *SA-readying* agent performs a procedure which is slightly different than the *ready* procedure in the intermediate model; it selects only parallel or sequential super-actors for processing.

```

procedure SA-ready ()
  do (forever)
    from Enabled Pool, pick  $\langle OP, AP \rangle$  with
       $AA[AP.AT] = \text{'par-sa' or 'seq-sa'}$ 
    if (no free block in  $FB[ ]$ ) then wait
    /* fetch instructions of enabled actor instance into fast memory */
     $ptr := \text{label of a free block}$ 
    fetch  $I[AA[AP.I]]$  to  $I[AA[AP.I] + AA[AP.N_I] - 1]$  and put into  $FB[ptr]$ 
     $PR\{OP, AP\}[IP] := ptr$ 
    /* now fetch data blocks into fast memory */
    do ( $i := 1$  to  $AA[AP.N_{BA}]$ )
      if (no free block in  $FB[ ]$ ) then wait
       $ptr := \text{label of a free block}$ 
      fetch data block at  $D[\text{block-addr}(AA[AP.BA.i])]$  and put into  $FB[ptr]$ 
       $PR\{OP, AP\}[AA[AP.BA.i.b-ptr]] := ptr$ 
    enddo
    put  $\langle OP, AP \rangle$  into ready-SA pool
  enddo

```

The *SA-execution* agent executes the following procedure:

```

procedure SA-execute ()
  do (forever)
    from active-SA pool, pick  $\langle OP, AP \rangle$  with
       $D[OP + AA[AP.rs]] = \text{'non-terminated'}$ 
    if ( $AA[AP.AT] = \text{'seq-sa'}$ ) then
       $cnt := 0$ 
      while ( $cnt < AA[AP.N_I]$ ) do
        perform  $(OP, FB[PR\{OP, AP\}[IP] + cnt], AA[AP.N_I], cnt)$ 

```

```

        cnt := cnt + 1
    endwhile
else
    In parallel do (x := 0 to (AA[AP.NI] - 1))
        perform (OP, FB[PR{OP, AP}[IP] + x], AA[AP.NI], x)
    enddo
endif
put (OP, AP) back in active-SA pool
D[OP + AA[AP.rs]] := 'terminated'
enddo

```

The *OA-activation* agent performs a straightforward task:

```

procedure OA-activate ()
do (forever)
    from enabled pool, pick (OP, AP) with AA[AP.AT] = 'other-actor'
    put (OP, AP) into active-OA pool
    D[OP + AA[AP.rs]] := 'non-terminated'
enddo

```

The *OA-execution* agent performs a procedure which is a big case statement similar to the one in the *execute* procedure of the basic abstract machine model (section 5.2.2).

```

procedure OA-execute ()
do (forever)
    from active-OA pool, pick (OP, AP)
    with D[OP + AA[AP.rs]] = 'non-terminated'
    cnt := 0
    while (cnt < AA[AP.NI]) do
        perform (OP, AP, I[AA[AP.I] + cnt], AA[AP.NI], cnt)
        cnt := cnt + 1
    endwhile
    put (OP, AP) back in active-OA pool
    D[OP + AA[AP.rs]] := 'terminated'
enddo

```

Finally, we list the procedure for the deactivation-enabling agent. The *decrement-reset* procedure is the same as the one in the intermediate model.

```

procedure deactivate-enable ()
  do (forever)
    from active-SA pool or active-OA pool, pick  $\langle OP, AP \rangle$  with
       $D[OP + AA[AP.rs]] = \text{'terminated'}$ 
    if (actor from active-SA pool) then
      deallocate assigned resources and copy data blocks back to main memory
      and free those data blocks
    do ( $i := 0$  to  $(AA[AP.N_{su}] - 1)$ )
      decrement-reset ( $OP, AA[AP.SL_u + i]$ )
    enddo
    if ( $D[OP + AA[AP.cc]] = \text{'true'}$ ) then
      do ( $i := 0$  to  $(AA[AP.N_{st}] - 1)$ )
        decrement-reset ( $OP, AA[AP.SL_t + i]$ )
      enddo
    if ( $D[OP + AA[AP.cc]] = \text{'false'}$ ) then
      do ( $i := 0$  to  $(AA[AP.N_{sf}] - 1)$ )
        decrement-reset ( $OP, AA[AP.SL_f + i]$ )
      enddo
    enddo

```

Appendix C

An Assembly Language for the SAM

In this appendix, we describe a preliminary version of an assembly language for the Super-Actor Machine, called SAMAL (Super-Actor Machine Assembly Language). Programs expressed in SAMAL can be straightforwardly translated to an appropriate machine representation for execution on the Super-Actor Machine. Some issues in the assembly process are examined in chapter 7.

C.1 Super-Actors

The assembler directive for specifying a super-actor is **SA**. The *type* of the super-actor, be it sequential ('S') or parallel ('P'), must be specified along with its identifier (*label*). A fast-path candidate has a type specifier of 'FS' or 'FP' depending on whether it is sequential or parallel.

Super-Actors
<i>Directive</i>
SA type label sig-list [t-sig-list, f-sig-list] ([data-block-desc] ¹) ([instruction] ¹)
<i>Instructions</i>
optr oprnd1 [oprnd2] result brx [oprnd1] offset wrtto to-loc line

An unconditional signal list (*sig-list*) has the form (*U: actor-id1,...*) which contains actor identifiers. Each identifier has a prefix to indicate whether the signal is an acknowledgement signal ('>') or simply a signal ('<'). This indicator is used by the assembler to calculate the appropriate weights of count signals. The true and false signal lists (*t-sig-list* and *f-sig-list* respectively) are optional and are dependent on whether the super-actor is a switch super-actor or not. The true and false signal lists have the form (*T: actor-id1,...*) and (*F: actor-id1,...*), respectively. The array of data block descriptors (*data-block-desc*) is information pertaining to the operand/result memory blocks the super-actor operates upon, i.e., these are load instructions for the R-cache loader. (Readers are reminded that symbols of the form [*xx*]¹ imply one or more *xx*'s.) A data block descriptor can be a memory block address, an offset from the local overlay's base address which indicates the memory address of a data block, or a pointer value. A pointer value locates an overlay slot which contains the memory address of a data block; a pointer value is used for indirect data block addressing (see page 100 for indirect data block addressing). In a data block descriptor, the prefix '#' indicates a memory block address, an offset from the overlay base address (called an *overlay offset*) is indicated by '&', and a pointer value by '@'. Furthermore, a prefix of 'O' or 'R' identifies whether the data block descriptor is an operand or result line, and the prefix 'L' indicates a mandatory load of the data block into d-R-cache.

The operator (*optr*) of a super-actor instruction can be an arithmetic or logic instruction such **ladd** for integer add, **fmul** for floating-point multiply, etc. The operand field of these instructions can be an immediate value, a R-cache line descriptor, an indirection descriptor

or a register name (see section 6.2.6). An immediate value is written as '#*i*', the form '*rc i.j*' is used to indicate a data block descriptor, an indirection pointer is '@*i.j*', and a register name, '*reg i*'. The result field has the same modes as the operand field except that no immediate values are allowed.

Branch instructions **brx** are restricted to sequential super-actors. The operand field is required if the branch instruction is **brt** or **brf** because the memory pointed to by the operand field must contain a condition code which was generated by a previous relational or logic instruction. The operand can be a R-cache line descriptor, an indirection descriptor or a register name (most often, it will be a register name). An unconditional branch, **br** does not require the operand field specification. The *offset* is an integer value (an immediate value) which the branch instruction must add to the respective instruction counter if the condition has been met (or unconditionally for **br**).¹

The **wrtto** instruction is used to copy a d-R-cache line specified by *line* (an index of the data block descriptor array) to either the support-actor execution pipe d-cache ('sppta') or the L-actor execution unit d-cache ('leu') as indicated by *to-loc*. Since the data cache line sizes in the support-actor execution pipe and LEU may be different from the d-R-cache line size, the *line* specifier may also contain a value indicating which portion of the d-R-cache line should be copied. For example, let's assume that the d-R-cache line is sixteen words long, and the other two data caches have four words per line. If the d-R-cache line is partitioned into four 4-word segments, then *line* can be a tuple indicating the d-R-cache line and which 4-word segment to be copied. In any case, we will leave this to future refinements of the architecture.

C.1.1 Support-Actors

Instructions in a support-actor are to be executed in a simplified RISC pipeline, thus standard RISC instructions will be used, i.e., all instructions are register-to-register and only load (**ld**) and store (**st**) instructions are used to access data memory.

¹In the simulated machine of chapter 8, the offset value is one less than the actual since every ready context in the SEU must have its instruction counter incremented before an instruction is fetched.

Support-Actors
<i>Directive</i>
SPPTA label sig-list ([instruction] ¹)
<i>Instructions</i>
ld reg1 immed res-reg st reg1 reg2 nop optr oprnd1 [oprnd2] result

The directive for specifying a support-actor is simpler because no data block descriptors need be specified. The **ld** instruction performs the operation:

$$\text{register}[\text{res-reg}] := \text{DM}[\text{register}[\text{reg1}] + \text{immed}]$$

and the **st** instruction:

$$\text{DM}[\text{register}[\text{reg2}]] := \text{register}[\text{reg1}]$$

nop instructions can be used to fill delayed load slots if no other instruction can be inserted. **optr** instructions are instructions like integer **add**, **mul**, etc. and their operands are either register names (**reg i**) or immediate values (**#i**). No floating point instructions are supported.

C.1.2 Long-Latency Actors

Long-latency actors consists of **nop** instruction, so the directive only serves the purpose of identifying the instruction and its associated list of actors to signal.

Support-Actors
<i>Directive</i>
LA label sig-list instruction
<i>Structure memory operations</i>
SMalloc size res SMdealloc oprnd SMread name offset dest [len] SMwrite name offset src [len]
<i>Inter-PE communications</i>
Send src overl dest recv-sa [len]
<i>Function applications</i>
Apply f-name arg1-offset res1-offset [len] Return ret1-offset [len]

SMalloc and **SMdealloc** are used to allocate and deallocate structure memory space. Structure memory can be used for representing arrays, etc. The *size* field specifies how many words the structure memory object must be and can be an overlay offset (&*i*) or an immediate value (#*i*). The field *res* is an overlay offset which points to the location containing the base address of the just allocated structure memory object. *oprnd* is an overlay offset pointing to a location which contains the base address of the structure memory object to deallocate. The *name* field of **SMread** and **SMwrite** is an overlay offset where the base address of the structure memory object can be found. *offset* is some location in the overlay (specified as an overlay offset) which contains an offset value from the structure memory object's base address. The offset plus the base address points to the first (or only one, depending on the optional *len* field)² element to be read or written. *len* can be an overlay base address offset or an immediate value. *dest* is the destination address where the structure memory element(s) is to be put. And *src* is the source address which contains

²Indexing and bounds checking can be performed by compiler supplied code segments or some hardware logic.

the data to be written to structure memory. Both *dest* and *src* can be overlay offsets or absolute address values (immediate operand).

The *src* of the **Send** instruction specifies a location within the overlay of the first source value to be copied to the location specified by the *overl* and *dest* values. The *overl* and *dest* indicate the overlay and offset within that overlay (the first destination location) to receive the first source value. *src*, *overl* and *dest* can be overlay offsets or immediate values. *recv-sa* is the id of the actor—the overlay base address (*overl*) along with the id forms the actor instance's identifier—which is to be signaled once the data has been copied. *recv-sa* can be an overlay offset or immediate value. The optional length argument (*len*) specifies the number of source values to be copied to the destination. Note that acknowledgement signals can be implemented by a **Send** instruction where the memory address of the source and destination values point to non-existent memory locations.

At the machine code level, function application instructions will be converted into a combination of calls to a memory resource manager which invokes the overlay operations as described in section 5.2.3, and some **Send** instructions. Information about the function will be extracted automatically by the assembler and put into the local constant area of a program segment. In the **apply** instruction, *f-name* is the function name and is an actual label; the assembler will be responsible for matching the labels to actual memory locations—locations where function invocation information can be found. *argl-offset* is the location of the first argument to be passed when the function is invoked and *resl-offset* specifies the location where the first return value can be stored. Both can be an overlay offset value. The optional length field *len*—an overlay offset or an immediate value—indicates how many arguments are to be passed to the callee function. The *retl-offset* in the **return** instruction is the location of the first return value and the optional length field indicates how many return values are to be sent back to the caller function. The *retl-offset* can be an overlay offset; the *len* field can be an overlay offset or an immediate value.

C.1.3 Miscellaneous

To specify a function, the **Func** directive is used. *no-args* is an immediate value specifying the number of input arguments the function is expecting, and *actor* is an actor directive. A virtual top node, one for each function definition, is specified by the directive **TN**. Its only component is an unconditional list. A virtual bottom node need not be specified since the 'return' L-actor terminates a function invocation. A virtual merge node is specified with the **MN** directive and simply contains the label of the node and an unconditional signal list.

Miscellaneous
<i>Directive</i>
Func label no-args ([actor] ¹)
TN sig-list
MN label sig-list

Appendix D

SAMAL Code for the Benchmark Programs

The Super-Actor Machine Assembly Language code for SAXPY, SAXPBYBC, Livermore Loop 1 and SAXPY3 are listed below. The SAMAL codes are a direct translation from the actual “machine code” used in the simulations¹. All the codes show the unrolled four times version of the loops in which the super-actor ‘begin’ is the top node, i.e., it is signaled to start the computation. (Note that the top node, ‘begin’, is also the loop pre-processing actor for the loop.) Unrolling to eight and sixteen times are straightforward and are not shown here. Creating multiple instances of the loop bodies, e.g., two, four and eight instances requires the simple replication of an overlay for each loop instance and triggering the top node of each instance at the beginning of the simulation run.

¹In the machine code for the simulated architecture, a **wrtto** instruction is simply indicated in a field attached to the result field of an appropriate instruction—the last instruction which writes to that d-R-cache line. All **wrtto** instructions simply force a write-back of the d-R-cache line back to main memory since the LEU has not been implemented for the simulations.

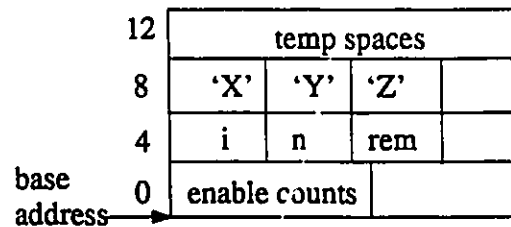


Figure D.1: Overlay layout for SAXPY.

D.1 SAXPY

SAXPY computes this expression:

```

Z := for i in 1, n
    returns array of
    a * X[i] + Y[i]
end for

```

The overlay layout for one unrolled-4 SAXPY loop instance is shown in figure D.1. Note that overlay does not include any function application information since the LEU has not been implemented in the simulation yet.

In the following SAMAL code, value 'a' is stored in memory location 0. The code only applies to an array size which is a multiple of four; code for handling the remaining elements (for an array size not a multiple of four) are not included. The number of remaining elements to be processed is deposited in location six of the overlay at the end of the loop.


```

SA P begin (U: >m)
  (O&4)
  (isub rc1.0 #4 rc1.0)

MN m (U: >comp)

SA FS comp (U: <comp) (T: <begin) (F: >xf)
  (O&4)
  (iadd rc1.0 #4      rc1.0
   isub rc1.1 rc1.0 reg1
   i<  reg1 #0      reg2
   brf  reg2 #1
   iadd reg1 #4      rc1.2
   i<  reg1 #0      reg1)

SPPTA igen1 (U: <xf <igen1)
  (ld  reg0 #8      reg1
   nop
   add  reg1 #4      reg1
   add  reg0 #8      reg2
   st   reg1 reg2
   nop)

SPPTA igen2 (U: <yf <igen2)
  (ld  reg0 #9      reg1
   ld  reg0 #10     reg2
   add  reg1 #4      reg1
   add  reg2 #4      reg2
   add  reg0 #9      reg3
   st   reg1 reg3
   add  reg3 #1      reg3
   st   reg2 reg3)

SA P xf (U: >yf >igen1 >m)
  (OL@8 O#0 R&12)
  (fmul rc1.0 rc2.0 rc3.0
   fmul rc1.1 rc2.0 rc3.1
   fmul rc1.2 rc2.0 rc3.2
   fmul rc1.3 rc2.0 rc3.3)

```

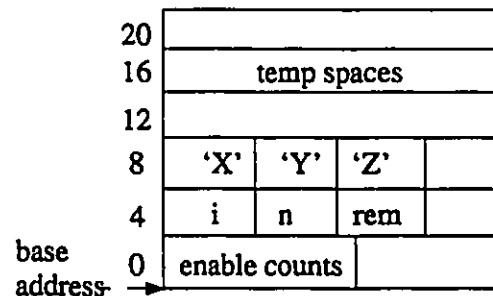


Figure D.2: Overlay layout for SAXPBYP.

```

SA P yf (U: <xf >igen2 <yf)
(O&12 OL@9 R@10)
(fadd rc1.0 rc2.0 rc3.0
 fadd rc1.1 rc2.1 rc3.1
 fadd rc1.2 rc2.2 rc3.2
 fadd rc1.3 rc2.3 rc3.3
 wrtto 'LEU #3)

```

D.2 SAXPBYP

SAXPBYP computes this expression:

```

Z := for i in 1, n
    returns array of
    a * X[i] + b * Y[i] + c
end for

```

The overlay layout for one unrolled-4 SAXPBYP loop instance is shown in figure D.2.

In the following SAMAL code, value 'a' is stored in memory location zero, 'b' in one and 'c' in two.

```

SA P begin (U: >m)
  (O&4)
  (isub rcl.0 #4 rcl.0)

MN m (U: >comp)

SA FS comp (U: <comp) (T: <begin) (F: >xf >yf)
  (O&4)
  (iadd rcl.0 #4      rcl.0
   isub rcl.1 rcl.0 reg1
   i<  reg1 #0      reg2
   brf  reg2 #1
   iadd reg1 #4      rcl.2
   i<  reg1 #0      reg1)

SPPTA igen1 (U: <xf <igen1)
  (ld  reg0 #8      reg1
   nop
   add  reg1 #4      reg1
   add  reg0 #8      reg2
   st   reg1 reg2
   nop)

SPPTA igen2 (U: <yf <igen2)
  (ld  reg0 #9      reg1
   nop
   add  reg1 #4      reg1
   add  reg0 #9      reg2
   st   reg1 reg2
   nop)

SPPTA igen3 (U: <add2 <igen3)
  (ld  reg0 #10     reg1
   nop
   add  reg1 #4      reg1
   add  reg0 #10     reg2
   st   reg1 reg2

```

```
    nop)

SA P xf (U: >add1 >igen1 >m)
    (OL@8 O#0 R&12)
    (fmul rc1.0 rc2.0 rc3.0
     fmul rc1.1 rc2.0 rc3.1
     fmul rc1.2 rc2.0 rc3.2
     fmul rc1.3 rc2.0 rc3.3)

SA P yf (U: >add1 >igen2 >m)
    (O#0 OL@9 R&16)
    (fmul rc1.1 rc2.0 rc3.0
     fmul rc1.1 rc2.1 rc3.1
     fmul rc1.1 rc2.2 rc3.2
     fmul rc1.1 rc2.3 rc3.3)

SA FP add1 (U: >add2 <xf <yf)
    (O&12 O&16 R&20)
    (fadd rc1.0 rc2.0 rc3.0
     fadd rc1.1 rc2.1 rc3.1
     fadd rc1.2 rc2.2 rc3.2
     fadd rc1.3 rc2.3 rc3.3)

SA P add2 (U: <add1 <add2 >igen3)
    (O&20 O#0 R@10)
    (fadd rc1.0 rc2.2 rc3.0
     fadd rc1.1 rc2.2 rc3.1
     fadd rc1.2 rc2.2 rc3.2
     fadd rc1.3 rc2.2 rc3.3
     wrtto 'LEU #3)
```

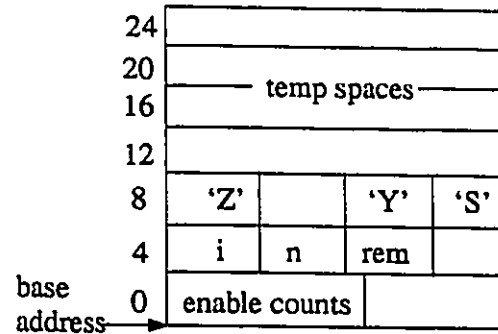


Figure D.3: Overlay layout for Livermore Loop1.

D.3 Livermore Loop1

Livermore Loop1 computes the expression:

```

S := for i in 1, n
    returns array of
    q + (Y[i] * (r * Z[i + 10] + t * Z[i + 11]))
end for

```

The overlay layout for one unrolled-4 livermore loop1 loop instance is shown in figure D.3.

In the following SAMAL code, value 'q' is stored in memory location zero, 'r' in one and 't' in two.

```

SA S begin (U: >m)
  (O&4 O&8)
  (isub rc1.0 #4 rc1.0
   iadd rc2.0 #8 rc2.0
   iadd rc2.0 #4 rc2.1)

```

MN m (U: >comp)

SA FS comp (U: <comp) (T: <begin) (F: >z1 >z2)
 (O&4)
 (iadd rc1.0 #4 rc1.0
 isub rc1.1 rc1.0 reg1
 i< reg1 #0 reg2
 brf reg2 #1
 iadd reg1 #4 rc1.2
 i< reg1 #0 reg1)

SPPTA igen1 (U: <mult1 <igen1)
 (ld reg0 #10 reg1
 nop
 add reg1 #4 reg1
 add reg0 #10 reg2
 st reg1 reg2
 nop)

SPPTA igen2 (U: <z1 <z2 <igen2)
 (ld reg0 #8 reg1
 ld reg0 #9 reg2
 add reg1 #4 reg1
 add reg2 #4 reg2
 add reg0 #8 reg3
 st reg1 reg3
 add reg3 #1 reg3
 st reg2 reg3)

SPPTA igen3 (U: <add1 <igen3)
 (ld reg0 #11 reg1
 nop
 add reg1 #4 reg1
 add reg0 #11 reg2
 st reg1 reg2
 nop)

SA P z1 (U: >igen1 >z3 >m)
 (O#0 O@8 O@9 R&12)
 (fmul rc1.0 rc2.2 rc4.0

```

    fmul rc1.1 rc2.3 rc4.1
    fmul rc1.0 rc2.3 rc4.2
    fmul rc1.1 rc3.0 rc4.3)

SA P z2 (U: >igen2 >z3 >m)
(O#0 O@9 R&16)
(fmul rc1.0 rc2.0 rc3.0
 fmul rc1.1 rc2.1 rc3.1
 fmul rc1.0 rc2.1 rc3.2
 fmul rc1.1 rc2.2 rc3.3)

SA FP z3 (U: >mult1 <z1 <z2\
(O&12 O&16 R&20)
(fadd rc1.0 rc1.1 rc3.0
 fadd rc1.2 rc1.3 rc3.1
 fadd rc2.0 rc2.1 rc3.2
 fadd rc2.2 rc2.3 rc3.3)

SA P mult1 (U: >add1 <z3 >igen1)
(OL@10 O&20 R&24)
(fmul rc1.0 rc2.0 rc3.0
 fmul rc1.1 rc2.1 rc3.1
 fmul rc1.2 rc2.2 rc3.2
 fmul rc1.3 rc2.3 rc3.3)

SA P add1 (U: <mult1 <add1 >igen3)
(O#0 O&24 R@11)
(fadd rc1.0 rc2.0 rc3.0
 fadd rc1.0 rc2.1 rc3.1
 fadd rc1.0 rc2.2 rc3.2
 fadd rc1.0 rc2.3 rc3.3
 wrtto 'LEU #3)

```

D.4 SAXPY3

The expression computed by SAXPY3 is shown in figure 8.2. The overlay layout for one unrolled-4 SAXPY3 loop instance is shown in figure D.4.

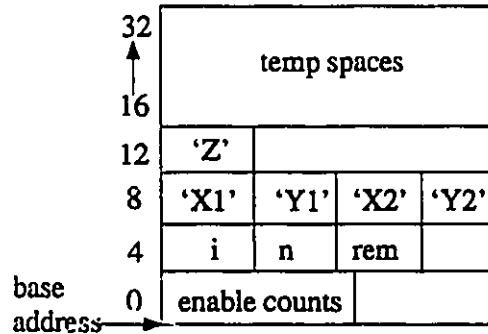


Figure D.4: Overlay layout for SAXPY3.

In the following SAMAL code, value 'a1' is stored in memory location zero, 'a2' in four and 'a3' in eight. Though the instruction list of super-actors x_2 , x_3 , and y_2 are given in the code, the actual machine code for x_2 and x_3 simply have pointers to the instructions of super-actor x_1 since they are all the same. The same is true for y_1 's and y_2 's instructions.

```

SA P begin (U: >m)
  (O&4)
  (isub rc1.0 #4 rc1.0)

MN m (U: >comp)

SA FS comp (U: <comp) (T: <begin) (F: >x1 >x2)
  (O&4)
  (iadd rc1.0 #4 rc1.0
   isub rc1.1 rc1.0 reg1
   i< reg1 #0 reg2
   brf reg2 #1
   iadd reg1 #4 rc1.2
   i< reg1 #0 reg1)

SPPTA igen11 (U: <x1 <igen11)
  (ld reg0 #8 reg1
   nop
   add reg1 #4 reg1)

```



```

add reg0 #8 reg2
st reg1 reg2
nop)

```

```

SPPTA igen12 (U: <y1 <igen2)
(ld reg0 #9 reg1
nop
add reg1 #4 reg1
add reg0 #9 reg2
st reg1 reg2
nop)

```

```

SPPTA igen21 (U: <x2 <igen21)
(ld reg0 #10 reg1
nop
add reg1 #4 reg1
add reg0 #10 reg2
st reg1 reg2
nop)

```

```

SPPTA igen22 (U: <y2 <igen22)
(ld reg0 #12 reg1
nop
add reg1 #4 reg1
add reg0 #12 reg2
st reg1 reg2
nop)

```

```

SPPTA igen3 (U: <y3 <igen3)
(ld reg0 #12 reg1
nop
add reg1 #4 reg1
add reg0 #12 reg2
st reg1 reg2
nop)

```

```

SA P x1 (U: >y1 >igen11 >m)
(OL@8 0#0 R&16)
(fmul rc1.0 rc2.0 rc3.0
fmul rc1.1 rc2.0 rc3.1

```

```
    fmul rc1.2 rc2.0 rc3.2
    fmul rc1.3 rc2.0 rc3.3)

SA P x2 (U: >y2 >igen21 >m)
  (OL@10 O#4 R&24)
  (fmul rc1.0 rc2.0 rc3.0
   fmul rc1.1 rc2.0 rc3.1
   fmul rc1.2 rc2.0 rc3.2
   fmul rc1.3 rc2.0 rc3.3)

SA FP x3 (U: <y1 >y3)
  (O&20 O#8 R&32)
  (fmul rc1.0 rc2.0 rc3.0
   fmul rc1.1 rc2.0 rc3.1
   fmul rc1.2 rc2.0 rc3.2
   fmul rc1.3 rc2.0 rc3.3)


SA P y1 (U: <x1 >igen12 >x3)
  (O&16 OL@9 R&20)
  (fadd rc1.0 rc2.0 rc3.0
   fadd rc1.1 rc2.1 rc3.1
   fadd rc1.2 rc2.2 rc3.2
   fadd rc1.3 rc2.3 rc3.3)

SA P y2 (U: <x2 >igen22 >y3)
  (O&24 OL@11 R&28)
  (fadd rc1.0 rc2.0 rc3.0
   fadd rc1.1 rc2.1 rc3.1
   fadd rc1.2 rc2.2 rc3.2
   fadd rc1.3 rc2.3 rc3.3)

SA P y3 (U: <x3 >igen3 <y2)
  (O&32 O&28 R@12)
  (fadd rc1.0 rc2.0 rc3.0
   fadd rc1.1 rc2.1 rc3.1
   fadd rc1.2 rc2.2 rc3.2
   fadd rc1.3 rc2.3 rc3.3
   wrtto 'LEU #3)
```



Bibliography

- [1] W. B. Ackerman. Efficient implementation of applicative languages. Technical Report 323, Laboratory for Computer Science, MIT, Cambridge, MA, March 1984.
 - [2] W. B. Ackerman and J. B. Dennis. VAL—a value-oriented algorithmic language. Technical Report 218, Laboratory for Computer Science, MIT, 1979.
 - [3] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiawicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104–114. ACM and IEEE, 1990.
 - [4] T. Agerwala and Arvind. Special issue on data flow systems. *IEEE Computer*, 15(2), February 1982.
 - [5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.
 - [6] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings Publishing, Redwood City, California, 1989.
 - [7] R. Alverson et al. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing, June 11–15, 1990, Amsterdam, Netherlands*, pages 1–6. ACM, 1990. Also in *ACM SIGARCH Computer Architecture News*, 18:3, September, 90.
 - [8] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.
- 

- [9] Arvind. Personal communication, 1990.
- [10] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.
- [11] Arvind and D. E. Culler. Managing resources in a parallel machine. In J. V. Woods, editor, *Fifth Generation Computer Architecture*, pages 103–121. Elsevier Science Publishers, 1986.
- [12] Arvind, D. E. Culler, and K. Ekanadham. The price of asynchronous parallelism: An analysis of dataflow architectures. Computation Structures Group Memo 278, Laboratory for Computer Science, MIT, 1988.
- [13] Arvind, M. L. Dertouzos, R. S. Nikhil, and G. M. Papadopoulos. Project dataflow—the Monsoon architecture and the Id programming language. Computation Structures Group Memo 285, Laboratory for Computer Science, MIT, March 1988.
- [14] Arvind and R. A. Iannucci. A critique of multiprocessing von Neumann style. In *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, pages 426–436, 1983.
- [15] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. Computation Structures Group Memo 226, Laboratory for Computer Science, MIT, 1987.
- [16] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, October 1989.
- [17] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.
- [18] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, and demand-driven interpretation of imperative languages. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, White Plains, NY, June 20–22, 1990*, page 257. ACM, 1990.

- [19] D. Callahan and A. Porterfield. Data cache performance of supercomputer applications. In *Proceedings of the Supercomputing '90 Conference*, New York, NY, 1990.
- [20] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990. White Plains, NY.
- [21] L. M. Censier and P. Feautrier. A new solution to the coherence problem in multicache systems. *IEEE Transactions on Computers*, pages 1112–1118, December 1978.
- [22] G. J. Chaitin et al. Register allocation via coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 98–105, 1982.
- [23] F. Chow and J. Hennessey. Register allocation by priority-based coloring. *Conference Record of ACM SIGPLAN Symposium on Compiler Construction*, 1984.
- [24] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150, 1988.
- [25] D. E. Culler and G. M. Papadopoulos. Monsoon: an explicit token-store architecture. In *Proceedings of the International Symposium on Computer Architecture*, 1990.
- [26] K. Dai and W. K. Giloi. A basic architecture supporting LGDG computation. In *Proceedings of the 1990 International Conference on Supercomputing*, Amsterdam, the Netherlands, 1990.
- [27] J. B. Dennis. First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1974.
- [28] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

- [29] J. B. Dennis. Data flow for supercomputers. In *Proceedings of the 1984 CompCon*, March 1984.
- [30] J. B. Dennis. The evolution of 'static' data-flow computing. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1990.
- [31] J. B. Dennis, J. B. Fosseien, and J. P. Linderman. Data flow schemas. In *International Symposium on Theoretical Programming*, LNCS 5, pages 187–215. Springer-Verlag, Berlin, 1972.
- [32] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the Supercomputing '88 Conference*, pages 368–373, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.
- [33] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. Technical Report TR-SOCS-88.06, School of Computer Science, McGill University, Montreal, February 1988.
- [34] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *The Second Annual Symposium on Computer Architecture*, pages 126–132, January 1975.
- [35] J. B. Dennis and K. S. Weng. Application of data flow computation to the weather problem. In *High Speed Computer and Algorithm Organization*, pages 143–157, New York, 1977. Academic Press.
- [36] J. B. Dennis, Y-P.L. Willie, and W. B. Ackerman. The MIT data flow engineering model. In *Proceedings of the IFIP 9th World Computer Congress*, Paris, France, September 1983.
- [37] P. Evripidou and J.-L. Gaudiot. The USC decoupled multilevel data-flow execution model. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.

- [38] J. A. Fisher. Wide instruction word architectures: solving the supercomputer software problem. In A. Lichnewsky and C. Saez, editors, *Supercomputing*, pages 55–71. Elsevier Science Publishers, New York, 1987.
- [39] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.
- [40] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A second opinion on data flow machines and languages. *IEEE Computer*, 15(2):58–69, February 1982.
- [41] G. R. Gao. Aspects of balancing techniques for pipelined data flow code generation. *Journal of Parallel and Distributed Computing*, 6:39–61, 1989.
- [42] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining*. Kluwer Academic Publishers, Boston, December 1990.
- [43] G. R. Gao. A flexible architecture model for hybrid data-flow and control-flow evaluation. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.
- [44] G. R. Gao, H. H. J. Hum, and J. M. Monti. Towards an efficient hybrid dataflow architecture model. In *Proceedings of PARLE '91, Parallel Architectures and Languages Europe*, Eindhoven, the Netherlands, June 1991.
- [45] G. R. Gao, H. H. J. Hum, and Y. B. Wong. An efficient scheme for fine-grain software pipelining. In *Proceedings of the CONPAR '90-VAPP IV Conference*, pages 709–720, Zurich, Switzerland, September 1990.
- [46] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Limited balancing —an efficient method for dataflow software pipelining. In *Proceedings of the International Symposium on Parallel and Distributed Computing, and Systems*, New York, NY, October 1990.
- [47] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Proceedings of PARBASE '90—International Conference on Databases, Parallel Architectures, and Their Applications*, pages 112–116, Miami Beach, FL, March 7–9 1990. IEEE Computer Society.

- [48] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Towards efficient fine-grain software pipelining. In *Proceedings of the ACM International Conference on Supercomputing*, pages 369–379, Amsterdam, the Netherlands, June 1990.
- [49] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Toward efficient fine-grain software pipelining and the limited balancing technique. *International Journal of Mini and Microcomputers*, 13(2):57–68, 1991.
- [50] G. R. Gao and R. Tio. Instruction set definition for the argument-fetching data-flow machine. ACAPS Technical Memo 01, School of Computer Science, McGill University, Montreal, February 1988.
- [51] G. R. Gao, R. Tio, and H. H. J. Hum. Design of an efficient dataflow architecture without dataflow. In *Proceedings of the International Conference on Fifth-Generation Computers*, pages 861–868, Tokyo, Japan, December 1988.
- [52] J. L. Gaudiot. Structure handling in data flow systems. *IEEE Transactions on Computers*, C-35(6):489–502, 1986.
- [53] J. L. Gaudiot and L. Bic, editors. *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.
- [54] P. P. Gelsinger et al. Microprocessors circa 2000. *IEEE Spectrum*, pages 43–47, October 1989.
- [55] D. Ghosal and L. N. Bhuyan. Performance evaluation of a dataflow architecture. *IEEE Transactions on Computers*, 39(5):615–627, May 1990.
- [56] J. R. Goodman and M.-C. Chiang. The use of static column RAM as a memory hierarchy. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 167–174, 1984.
- [57] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon data-flow processor. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 36–45, Israel, June 1989.

- [58] T. R. Gross. *Code Optimization of Pipeline Constraints*. PhD thesis, Stanford University, 1983.
- [59] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.
- [60] R. H. Halstead Jr and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, 1988.
- [61] J. L. Hennessy and N. P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, pages 18–29, Sept. 1991.
- [62] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.
- [63] L. Hennessy. VLSI processor architecture. *IEEE Transactions on Computers*, C-33(12):1221–1246, 1984.
- [64] K. Hiraki, S. Sekiguchi, and T. Shimada. Efficient vector processing on a dataflow supercomputer SIGMA-1. In *Proceedings of IEEE Computer Society and ACM SIGARCH Supercomputing '88 Conference*, Orlando, FL, 1988.
- [65] H. H. J. Hum and G. R. Gao. A register-cache for fine-grain multi-thread computing. ACAPS Technical Memo 13, School of Computer Science, McGill University, Montreal, August 1990.
- [66] H. H. J. Hum and G. R. Gao. Efficient support of concurrent threads in a hybrid dataflow/von Neumann architecture. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, December 1991.
- [67] H. H. J. Hum and G. R. Gao. A novel high-speed memory organization for fine-grain multi-thread computing. In *Proceedings of PARLE '91, Parallel Architectures and Languages Europe*, Eindhoven, the Netherlands, June 1991.

- [68] H. H. J. Hum and G. R. Gao. A high-speed memory organization for hybrid dataflow/von neumann computing. *Future Generation Computer Systems*, to appear in 1992.
- [69] H. H. J. Hum and G. R. Gao. Modeling the Super-Actor Machine. Acaps technical memo, School of Computer Science, McGill University, Montreal, 1992. in preparation.
- [70] H. H. J. Hum and Y. B. Wong. A prototype of an argument-fetching dataflow machine interpreter/simulator. ACAPS Design Note 13, School of Computer Science, McGill University, Montreal, May 1989.
- [71] H. H. J. Hum and Y. B. Wong. The design and implementation of an argument-fetching dataflow machine testbed. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, Ottawa, Ont., September 1990.
- [72] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw Hill Book Company, New York, 1984.
- [73] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140. ACM, June 1988.
- [74] R. A. Iannucci. *Parallel Machines: Parallel Machine Languages*. Kluwer Academic Publishers, Boston, MA, 1990.
- [75] R. A. Iannucci. *A Dataflow / von Neumann Hybrid Architecture*. PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, May 1988. Technical Report MIT/LCS/TR-228.
- [76] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282. Boston, MA, 1988.

- [77] N. P. Jouppi, J. Bertoni, and D. W. Wall. A unified vector/scalar floating-point architecture. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 134–143, Boston, Massachusetts, 1989.
- [78] M. G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI*. MIT Press, Cambridge, 1985.
- [79] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the Eighth International Symposium on Computer Architecture*, 1981.
- [80] J. R. McGraw et al. SISAL: Streams and iteration in a single assignment language—language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, 1985.
- [81] F. H. McMahon. The Livermore FORTRAN Kernels: A computer test of numerical performance ranges. Technical Report UCRL-537415, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.
- [82] J. M. Monti. Interprocessor communication supports for a multiprocessor dataflow machine. Master's thesis, School of Computer Science, McGill University, Montreal, March 1991.
- [83] L. B. Montz. Safety and optimization transformations for data flow programs. Technical Report 240, Laboratory for Computer Science, MIT, Cambridge, MA, January 1980.
- [84] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 262–272, Israel, 1989.
- [85] R. S. Nikhil. Id (Version 88.0) reference manual. Computation Structures Group Memo 284, Laboratory for Computer Science, MIT, March 1988.

- [86] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: a killer micro for a brave world. Computational Structures Group Memo TM-325, M. I. T. Laboratory for Computer Science, July 1991.
- [87] R.S. Nikhil. The parallel programming language Id and its compilation for parallel machines. Computation Structures Group Memo 313, Laboratory for Computer Science, MIT, July 1990.
- [88] E. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, Aug. 1991.
- [89] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, MIT, 1988.
- [90] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the Seventeenth Annual International Symposium of Computer Architecture*, Seattle, WA, pages 82–91, 1990.
- [91] S. Z. Pasha and E. H. Welbon. Performance-directed design guidance using simulation. In Mamata Misra, editor, *IBM RISC System/6000 Technology*. International Business Machines Corporation, 1990. Order No. SA23-2619.
- [92] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, 1985.
- [93] A. Plas. LAU system architecture. In *Proceedings of the 1976 International Conference on Parallel Processing*, Aug. 1976.
- [94] A. Plas, D. Comte, O. Gelly, and J. C. Syre. LAU system architecture: A parallel data driven processor based on single assignment. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 293–302, 1976.
- [95] J. E. Rumbaugh. A parallel asynchronous computer architecture for data flow programs. Technical Report MIT/LCS/TR-150, Laboratory for Computer Science, MIT, 1975.

- [96] S. Sakai et al. An architecture of a dataflow single chip processor. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 46–53, Israel, 1989.
- [97] S. Sakai et al. Pipeline optimization of a data-flow machine. In J. L. Gaudiot and L. Bic, editors, *Advanced Topics in Dataflow Computing*. Prentice-Hall, 1990.
- [98] V. Sarkar. Synchronization using counting semaphores. *Proceedings of the ACM 1988 International Conference on Supercomputing*, pages 627–637, July 1988.
- [99] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. This monograph is a revised version of the Author's Ph.D. dissertation published as Technical Report CSL-TR-87-328, Stanford University, April 1987.
- [100] C. Scheurich and M. Dubois. The design of a lockup-free cache for high-performance multiprocessors. In *Proceedings of the Supercomputing '88 Conference*, Orlando, FL, 1988.
- [101] M. L. Simmons and H. J. Wasserman. Performance evaluation of the IBM RISC System/6000: Comparison of an optimized scalar processor with two vector processors. In *Proceedings of Supercomputing '90*, pages 132–141. IEEE, November 1990.
- [102] A. J. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, Nov. 1984.
- [103] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, Sept. 1989.
- [104] B. J. Smith. A pipelined shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, 1978.
- [105] B. J. Smith. The architecture of HEP. In J. S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and its Application*, pages 41–55. The MIT Press, 1985.

- [106] K. So and V. Zecca. Program locality of vectorized applications running on the IBM 3090 with vector facility. *IBM Systems Journal*, 27(4):436–451, 1988.
- [107] K. So and V. Zecca. Cache performance of vector processors. In *15th International Symposium on Computer Architecture*, pages 261–268. IEEE, June 1988.
- [108] H. S. Stone and J. Cocke. Computer architecture in the 1990s. *IEEE Computer*, pages 30–38, Sept. 1991.
- [109] S. S. Thakkar, editor. *Selected Reprints on Dataflow and Reduction Architectures*. IEEE Computer Society Press, Washington, D.C., 1987.
- [110] Mark R. Thistle and Burton J. Smith. A processor architecture for Horizon. In *Proceedings of the Supercomputing '88 Conference*, Orlando, FL, 1988.
- [111] J. E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the AFIPS Fall Joint Computer Conference*, 1964.
- [112] K. W. Todd. Function sharing in a static data flow machine. In *Proceedings of the 1982 International Conference on Parallel Processing*, pages 137–139, 1982.
- [113] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.
- [114] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, 1986.
- [115] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data-driven and demand-driven computer architecture. *Computing Surveys*, 14(1):93–143, March 1982.
- [116] P. C. Treleaven, R. P. Hopkins, and P. W. Rautenbach. Combining data flow and control flow computing. *Computer Journal*, 25(2):207–217, 1982.
- [117] A. H. Veen. Dataflow machine architecture. *Computing Surveys*, 18(4):365–396, December 1986.

- [118] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 176–188, April 1991.
- [119] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.
- [120] C.-L. Wu and T.-Y. Feng. *Tutorial: Interconnection Networks for Parallel and Distributed Processing*. IEEE Computer Society Press, Washington, D.C., 1984.
- [121] T. Yuba et al. Sigma-1: A dataflow computer for scientific computations. *Computer Physics Communications*, 37:141–148, 1985.