

# Running Stream-like Programs on Heterogeneous Multi-core Systems



Paul Carpenter

Advisor: [Eduard Ayguade](#)

Co-advisor: [Alex Ramirez](#)

[Computer Architecture Department](#)

[Universitat Politècnica de Catalunya](#)

A thesis submitted for the degree of

*Doctor of Philosophy*



## ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi: Running Stream-like Programs on Heterogeneous Multi-core Systems

Autor de la tesi: Paul Carpenter

Acorda atorgar la qualificació de:

- ☐ No apte
- ☐ Aprovat
- ☐ Notable
- ☐ Excel·lent
- ☐ Excel·lent Cum Laude

Barcelona, ..... de/d' ..... de .....

El President

El Secretari

.....  
(nom i cognoms)

.....  
(nom i cognoms)

El vocal

El vocal

El vocal

.....  
(nom i cognoms)

.....  
(nom i cognoms)

.....  
(nom i cognoms)



## Abstract

All major semiconductor companies are now shipping multi-cores. Phones, PCs, laptops, and mobile internet devices will all require software that can make effective use of these cores. Writing high-performance parallel software is difficult, time-consuming and error prone, increasing both time-to-market and cost. Software outlives hardware; it typically takes longer to develop new software than hardware, and legacy software tends to survive for a long time, during which the number of cores per system will increase. Development and maintenance productivity will be improved if parallelism and technical details are managed by the machine, while the programmer reasons about the application as a whole.

Parallel software should be written using domain-specific high-level languages or extensions. These languages reveal implicit parallelism, which would be obscured by a sequential language such as C. When memory allocation and program control are managed by the compiler, the program's structure and data layout can be safely and reliably modified by high-level compiler transformations.

One important application domain contains so-called stream programs, which are structured as independent kernels interacting only through one-way channels, called streams. Stream programming is not applicable to all programs, but it arises naturally in audio and video encode and decode, 3D graphics, and digital signal processing. This representation enables high-level transformations, including kernel unrolling and kernel fusion.

Kernel unrolling coarsens granularity by batching up work into larger chunks, reducing overheads and potentially enabling data reuse and vectorisation. Kernel fusion combines multiple kernels into a single piece of code, which has two benefits. First, it can be used to match the number of kernels to the number of processors, required for static scheduling. Second, it coarsens granularity, which amortises overhead in a dynamic scheduler. Kernel unrolling and fusion are relatively straightforward to apply when the program is represented as a stream graph, even though they imply extensive changes to memory allocation and program control.

This thesis develops new compiler and run-time techniques for stream programming. The first part of the thesis is concerned with a statically scheduled stream compiler. It introduces a new static partitioning algorithm, which determines which kernels should be fused, in order to balance the loads on the processors and interconnects. A good partitioning algorithm is crucial if the compiler is to produce efficient code. The algorithm also takes account of downstream compiler passes—specifically software pipelining and buffer allocation—and it models the

compiler’s ability to fuse kernels. The latter is important because the compiler may not be able to fuse arbitrary collections of kernels.

This thesis also introduces a static queue sizing algorithm. This algorithm is important when memory is distributed, especially when local stores are small. The algorithm takes account of latencies and variations in computation time, and is constrained by the sizes of the local memories.

The second part of this thesis is concerned with dynamic scheduling of stream programs. First, it investigates the performance of known online, non-preemptive, non-clairvoyant dynamic schedulers. Second, it proposes two dynamic schedulers for stream programs. The first is specifically for one-dimensional stream programs. The second is more general: it does not need to be told the stream graph, but it has slightly larger overhead.

This thesis also introduces some support tools related to stream programming. StarssCheck is a debugging tool, based on Valgrind, for the StarSs task-parallel programming language. It generates a warning whenever the program’s behaviour contradicts a pragma annotation. Such behaviour could otherwise lead to exceptions or race conditions. StreamIt to OmpSs is a tool to convert a streaming program in the StreamIt language into a dynamically scheduled task based program using StarSs.

The main contributions of this thesis are:

- The Abstract Streaming Machine (ASM), a machine model and coarse-grain simulator for a statically scheduled stream compiler.
- A new partitioning heuristic for stream programs, which balances the load across the target, including processors and communication links. It considers its effect on downstream passes, and it models the compiler’s ability to fuse kernels.
- Two static queue sizing algorithms for stream programs, which determine the sizes of the buffers used to implement streams. The optimal buffer sizes are affected by latency and variability in computation costs, and are constrained by the sizes of local memories, which may be small.
- Two new low-complexity adaptive dynamic scheduling algorithms for stream-like programs.
- StarssCheck, a debugging tool for StarSs. This tool checks memory accesses performed by tasks and the main thread, and warns if the StarSs pragmas are incorrect.

## Acknowledgements

First and foremost, I would like to thank my advisor, Eduard Ayguade, and co-advisor, Alex Ramirez. This dissertation could not have been written without their patient guidance throughout. They gave me the freedom to try my own ideas and make my own mistakes, while gently steering me in the right direction when necessary. Alex knows how to take a vague idea, make it real, evaluate it, and produce a clear and compelling set of contributions. He persuaded me to stop describing my work with words like *simple* and *obvious*. Eduard had the experience and clarity to make sure I was steadily advancing in the right direction. Both were remarkably generous with their time, and they made research an enjoyable and fulfilling experience.

I am indebted to the *Barcelona Supercomputing Center* under the warm leadership of Mateo Valero. The BSC supported this work by providing a friendly and productive working environment, access to resources and projects, and financial support. Thanks to all the administrative staff, especially Trini Carneros at UPC DAC, who put up with my difficulty in performing even the most undemanding of administrative tasks.

I would like to thank the funding bodies that enabled this work to take place: the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625), the European Commission in the context of the ACOTES project (contract no. IST-34869), ENCORE project (ICT-FP7-248647), and the HiPEAC Network of Excellence (contract nos. IST-004408 and FP7/ICT 217068).

I wish to thank my parents, who have always been there for me. They always believed that everyone should have a passion for something and the opportunity to make their passion into their career. They bought my first computer, a Texas Instruments TI-99/4A, when I was aged eight. They continued to support my interests in science and computing, and my brother's interests in animal medicine and surgery. Now, both their sons are doing what they enjoy.

Finally, I wish to express my warmest thanks to Anja and our two young sons, Ruben and Daniel. Anja has an extraordinary ability to help others grow. She takes an interest in people, and she gives them the push to send that email, or make that call. She was instrumental in my move from successful professional to graduate student, and she has supported me in uncountable ways while I was writing this dissertation. Every day, Anja and my sons show me what is most important in life.

---



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Approaches to parallelism . . . . .	3
1.1.1	Data parallelism . . . . .	3
1.1.2	Task-level parallelism . . . . .	5
1.1.3	Streaming parallelism . . . . .	6
1.2	Compile-time vs run-time decisions . . . . .	7
1.3	The ACOTES stream compiler . . . . .	8
1.4	Task-level languages . . . . .	9
1.4.1	StarSs and OmpSs . . . . .	9
1.5	Streaming languages . . . . .	12
1.5.1	StreamIt language . . . . .	12
1.5.2	SPM (Stream Programming Model) . . . . .	16
1.6	Tool flow and support tools . . . . .	17
1.6.1	Debugging using StarssCheck . . . . .	18
1.6.2	Performance visualisation . . . . .	18
1.6.3	StreamIt to OmpSs . . . . .	18
1.7	Contributions and publications . . . . .	19
1.8	Thesis outline . . . . .	20
<b>2</b>	<b>Abstract Streaming Machine</b>	<b>23</b>
2.1	Scope of the ASM . . . . .	23
2.2	ASM Machine Description . . . . .	26
2.3	ASM Program Description . . . . .	28
2.4	Platform characterisation . . . . .	32
2.5	Validation of the ASM . . . . .	35
2.6	Using the ASM . . . . .	38
2.6.1	Static partitioning . . . . .	38
2.6.2	Static buffer sizing . . . . .	39
2.7	Related work . . . . .	39
<b>3</b>	<b>Compile-time Decisions</b>	<b>41</b>
3.1	Motivation . . . . .	41
3.1.1	Convexity . . . . .	42
3.1.2	Connectivity . . . . .	44
3.1.3	Queue sizes . . . . .	45

## CONTENTS

---

3.2	Static partitioning . . . . .	48
3.2.1	The partitioning problem . . . . .	48
3.2.2	The partitioning algorithm . . . . .	50
3.2.3	Evaluation . . . . .	54
3.3	Static buffer sizing . . . . .	58
3.3.1	The buffer sizing problem . . . . .	58
3.3.2	The buffer sizing algorithm . . . . .	59
3.3.3	Evaluation . . . . .	64
3.4	Related work . . . . .	65
3.5	Conclusions . . . . .	68
<b>4</b>	<b>Run-time Decisions</b>	<b>69</b>
4.1	The dynamic scheduling problem . . . . .	70
4.1.1	Interface to the dynamic scheduler . . . . .	70
4.1.2	Throttle policy . . . . .	71
4.1.3	Objective function: comparing schedulers . . . . .	72
4.2	Survey of DAG scheduling techniques . . . . .	72
4.2.1	The online scheduling policies . . . . .	73
4.2.2	Theoretical evaluation . . . . .	76
4.3	Adaptive schedulers . . . . .	81
4.3.1	Intuition . . . . .	81
4.3.2	Monitoring . . . . .	83
4.3.3	Updating priorities for <code>apriority</code> . . . . .	84
4.3.4	Updating priorities for <code>gpriority</code> . . . . .	85
4.4	Experimental evaluation . . . . .	88
4.5	Conclusions . . . . .	93
<b>5</b>	<b>Support Tools</b>	<b>97</b>
5.1	Debugging using StarssCheck . . . . .	97
5.1.1	Common StarSs errors . . . . .	98
5.1.2	How StarssCheck works . . . . .	99
5.1.3	Evaluation . . . . .	103
5.1.4	Related work . . . . .	106
5.2	Performance visualisation using Paraver Animator . . . . .	107
5.3	StreamIt to OmpSs conversion . . . . .	108
5.3.1	The conversion process . . . . .	109
5.3.2	Example: simplified FM Radio . . . . .	112
5.3.3	Current limitations . . . . .	115
5.3.4	High level transformations . . . . .	117
<b>6</b>	<b>Conclusions</b>	<b>121</b>
	<b>Glossary</b>	<b>123</b>
	<b>Bibliography</b>	<b>125</b>
	<b>Index</b>	<b>137</b>

# List of Figures

1.1	Metrics for Intel processors . . . . .	2
1.2	Data parallelism: four cores updating an array of data . . . . .	4
1.3	Gantt chart illustrating task-level parallelism . . . . .	4
1.4	Streaming parallelism: GNU radio FM demodulation . . . . .	4
1.5	The ACOTES iterative stream compiler . . . . .	9
1.6	Example StarSs code ( <i>bmod</i> function) . . . . .	10
1.7	Extracts from StarSs code for Cholesky decomposition . . . . .	11
1.8	Dependency graphs for Cholesky decomposition . . . . .	11
1.9	StreamIt components: pipelines, splitjoins, and feedback loops . . . . .	13
1.10	Example StreamIt 2.1 program, based on FMRadio5 . . . . .	14
1.11	Stream graph for the example StreamIt 2.1 program . . . . .	15
1.12	Example SPM program from [M <sup>+</sup> 11] . . . . .	16
1.13	Tool flow used in this thesis . . . . .	17
1.14	Thesis structure . . . . .	21
1.15	Timeline . . . . .	22
2.1	The ACOTES iterative stream compiler, reproduced from Figure 1.5 . . . . .	24
2.2	Topology of three example targets . . . . .	25
2.3	Processor and interconnect parameters of the ASM and example values . . . . .	27
2.4	Memory parameters of the ASM and values for two example targets . . . . .	28
2.5	Cost and latency of communication between tasks . . . . .	29
2.6	Building tasks from subtasks . . . . .	30
2.7	Example stream program with data-dependent flow . . . . .	31
2.8	Representation of data-dependent flow between tasks and subtasks . . . . .	32
2.9	Synthetic stream benchmarks . . . . .	33
2.10	Results for producer-consumer benchmark on Cell B.E. . . . .	33
2.11	Time per iteration for the chain and chain2 benchmarks on Cell B.E. . . . .	34
2.12	Comparison of real and simulated traces . . . . .	36
2.13	Detail on the main phases in the search algorithm . . . . .	38
2.14	Memory constraint graph for the Cell Broadband Engine . . . . .	39
3.1	Example partitions of the StreamIt filterbank benchmark onto a 3-core SMP . . . . .	42
3.2	Traces for five iterations of filterbank, scheduled using SGMS . . . . .	43
3.3	Motivation of connectivity . . . . .	45
3.4	If $k_2$ and $k_3$ are fused into one task, then the entire graph must be fused . . . . .	45

## LIST OF FIGURES

---

3.5	Effect of consumer queue length on chain8 and producer-consumer . . . . .	47
3.6	Memory-performance tradeoff . . . . .	48
3.7	The mapping phase of the ACOTES compiler . . . . .	49
3.8	Topology of the targets used in this section . . . . .	50
3.9	First level partition in the initial partition algorithm . . . . .	51
3.10	Convergence of the refinement phase as a function of the number of iterations . . . . .	55
3.11	Normalised execution time for the StreamIt 2.1.1 benchmarks for the three variants of the heuristic algorithm . . . . .	55
3.12	Vocoder benchmark on SMP 3 with accelerator using software pipelining. . . . .	57
3.13	Memory constraint graph for the Cell Broadband Engine . . . . .	58
3.14	Example timed event graph used by the critical cycle algorithm . . . . .	59
3.15	Example weighted wait-for graphs . . . . .	60
3.16	Example where baseline fails . . . . .	62
3.17	Token algorithm: bichain4 example . . . . .	63
3.18	Comparison of the buffer size update and cycle detection algorithms . . . . .	66
4.1	Example traces with static and dynamic scheduling . . . . .	70
4.2	Dynamic scheduler in context . . . . .	71
4.3	Number of ready tasks: channelvocoder, using oldest-first . . . . .	72
4.4	Metrics used by the scheduling policies . . . . .	73
4.5	Lattice of ready queues for a heuristic . . . . .	75
4.6	Worst case examples for exhaustion . . . . .	77
4.7	Worst case examples for back pressure . . . . .	78
4.8	Illustration for <i>oldest</i> back pressure . . . . .	80
4.9	Motivation for busyness statistic . . . . .	82
4.10	Example statistics after three iterations . . . . .	83
4.11	Example showing why all kernel statistics should be reset . . . . .	85
4.12	Example that shows the benefit from updating the ancestors . . . . .	86
4.13	Kernel dependency graph for H.264 decoder skeleton . . . . .	88
4.14	Scalability of the benchmarks . . . . .	90
4.15	Average and worst case results . . . . .	92
4.16	Worst case results: detail of efficiency of most robust schedulers . . . . .	93
4.17	Comparison of the scheduling heuristics: StreamIt . . . . .	94
4.18	Comparison of the scheduling heuristics: non-StreamIt . . . . .	95
5.1	Example mistakes found by StarssCheck . . . . .	100
5.2	Structure of StarssCheck . . . . .	100
5.3	Translated version of the <i>bmod</i> function from Figure 1.6 . . . . .	101
5.4	Starssgrind Contexts . . . . .	102
5.5	VEX IR for a single store instruction . . . . .	103
5.6	Performance results for Sparse LU factorisation . . . . .	103
5.7	Worst case “nasty” benchmark . . . . .	105
5.8	Potential false negatives and false positives . . . . .	106
5.9	Three instants during the execution of the H.264 decoder on 25 processors . . . . .	107
5.10	Main phases in StreamIt to OmpSs conversion . . . . .	109
5.11	Output from the build program . . . . .	110

## LIST OF FIGURES

---

5.12	Translation of an example StreamIt function . . . . .	111
5.13	Various FIFOs . . . . .	111
5.14	Non-peek version of the example StreamIt 2.1 program in Figure 1.10 . . . . .	113
5.15	Translated code exactly as generated by <code>str2oss</code> : part 1 . . . . .	114
5.16	Translated code exactly as generated by <code>str2oss</code> : part 2 . . . . .	115
5.17	Example <code>str2oss</code> control file . . . . .	116
5.18	An example where <code>str2oss</code> generates a function with too many arguments . . . . .	119

## LIST OF FIGURES

---

# List of Tables

1.1	Selection of task parallel languages and libraries . . . . .	5
1.2	Selection of stream programming languages . . . . .	6
2.1	Kernels and mappings of the GNU radio benchmark . . . . .	37
3.1	The number of pipeline stages . . . . .	56
4.1	Known DAG scheduling heuristics . . . . .	73
4.2	Online scheduling policies and cost . . . . .	74
4.3	Throughput for one-dimensional stream programs . . . . .	76
4.4	Statistics for adaptive schedulers . . . . .	83
4.5	Benchmarks used in the evaluation . . . . .	89
5.1	StarssCheck client requests . . . . .	101
5.2	Translation statistics for StreamIt benchmarks . . . . .	120

## LIST OF TABLES

---



# Chapter 1

## Introduction

All major semiconductor companies are now shipping multi-cores. Phones, PCs, laptops, and mobile internet devices already contain multiple processors instead of just one, and will require software that uses an increasing number of them effectively. Writing high-performance parallel software is difficult, time-consuming and error prone, increasing both development time and cost. Software outlives hardware; it typically takes longer to develop new software than hardware, and legacy software tends to survive for a long time, during which the number of cores per system will increase. Development and maintenance productivity will be improved if parallelism and technical details are managed by the machine, so that the programmer can concentrate on the application as a whole.

This thesis develops techniques to automatically map a certain class of program, known as *stream* programs, to multiple processors. Stream programs are significant subcomponents of many important applications, including those involving audio and video encoding and decoding, software radio, and 3D graphics. Moreover, due to their modularity and regularity, they ought to work well on multiple processors. In addition to investigating how to automatically map stream programs to multi-processors, this thesis introduces some tools that can be used to find bugs and improve performance.

Putting multiple CPUs into the same system is by no means new. The Burroughs D825 was the first Symmetric Multiprocessor; it had four processors, and it was launched in 1962 [AHSW62]. What *is* new, however, is that mainstream computing is currently being forced into multiprocessing. There are, in general, three ways through which computers become faster. First, *transistors* have been getting faster and smaller at an exponential rate, so last year's design, on today's semiconductor process, will be faster.<sup>1</sup> Second, smaller transistors give more room for complicated mechanisms to speed up a single processor, either through increasing the clock rate by overlapping instructions in a pipeline, or through super-scalar issue, by doing several unrelated instructions at once [HP07]. Both these techniques are limited, however, by excessive power consumption [OH05]. Third, an effective way to use an increasing number of transistors is to have on the same chip, several processors.

Figure 1.1 illustrates these trends, using data for Intel processors. Figure 1.1(a) shows that the total number of transistors has been doubling roughly every two years, consistently for about forty years, and that the trend is continuing still. Figure 1.1 (b) shows that clock

---

<sup>1</sup>The various parts of the semiconductor industry have been coordinating their capital and research investments through Moore's law [Moo65], shown in Figure 1.1.

# 1. INTRODUCTION

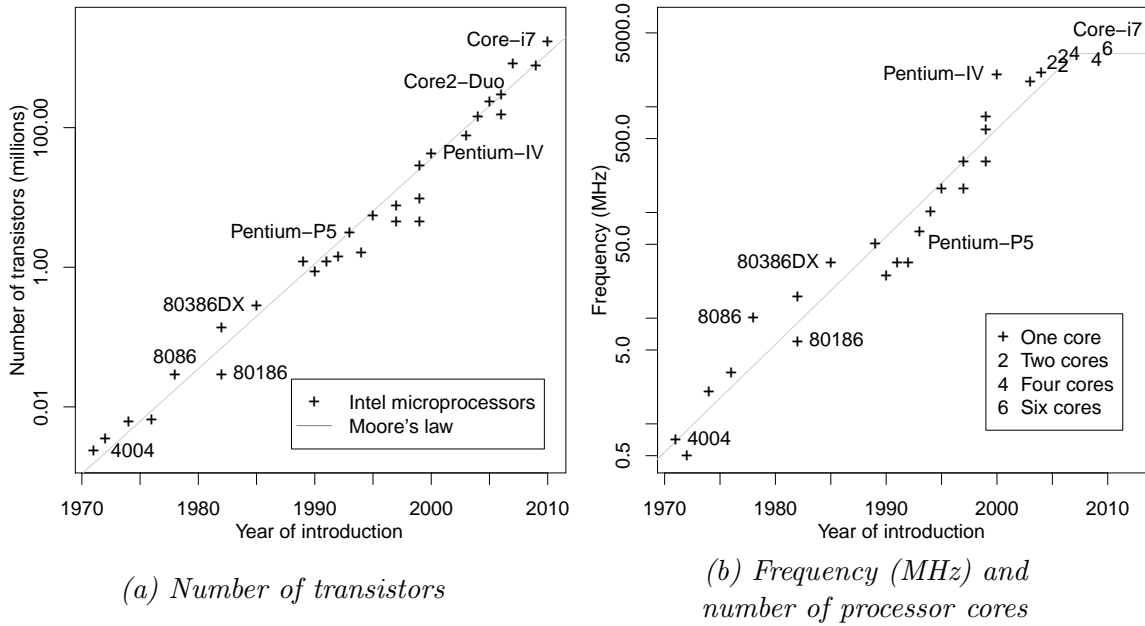


Figure 1.1: Metrics for Intel processors, using a logarithmic y axis

frequency has been doubling roughly every three years, but that the trend has flattened in recent years, with transistors instead being used for multiple processor cores. Multiprocessing has been common in embedded systems and standard in supercomputing for many years, the former for power consumption and specialisation<sup>1</sup> and to isolate subsystems from each other, and the latter because it is the only way to get the necessary performance. All supercomputers in the biannual TOP500 ranking [TOP] have had two or more processors ever since 1997, and the average number of processor cores in the machines in the November 2010 list is about 13,000.<sup>2</sup>

Multiple processors are pointless if they are not all kept busy, ignoring the benefits from specialisation for now; and they need to run, in parallel, on their own relatively large pieces of work. It is hard to write programs that do this. The first reason is that many computations, like many human activities, cannot easily be broken up into parts that can be done at the same time. An example is shown later in this section, in Figure 1.8(a). One way to find parallelism in such a program is to use a different algorithm. Another way is to use speculation to exploit the parallelism that exists but is hard to describe.

The second reason is that, with some exceptions, breaking a single program into parts and managing their execution in parallel currently has to be done by the programmer. Moreover, doing so exposes some unpleasant technicalities [Lee06]: primarily, the need to **lock** data structures so two threads don't try to change the same data at the same time and mess it up, and **weak memory consistency**, where interactions between processors reveal that

<sup>1</sup>An example is the TI OMAP1 series, containing a TI DSP for signal processing and an ARM core for applications, and benefiting from lower power consumption through specialisation [CCG+00].

<sup>2</sup>The average could be criticised since it is dominated by the top few machines, but it is less dependent on the arbitrary number "500". The most cores is 294,912 and the median is 6,372.

they are not the sequential machines they pretend to be [AG02]. The programmer must manage all this, while also presumably thinking about the big picture, in terms of what the application is ultimately trying to do. Interactions between multiple processors are inherently non-deterministic, because the program's behaviour can depend on the precise timing. Many bugs will therefore happen intermittently, and be difficult to reproduce, diagnose, and fix.

## 1.1 Approaches to parallelism

The parallelism that exists in software often has some kind of structure. A parallel programming model is a way for the application programmer to reveal this structure to the machine. The machine should then assign work to processors and deal with technicalities, allowing the programmer to concentrate on whatever the program is trying to achieve, which is probably also rather challenging. The main benefits are a shorter development time and lower cost, similar to the advantages of higher-level languages such as C or Python, compared with assembly language.

This thesis uses the three types of parallelism described in the following subsections.

### 1.1.1 Data parallelism

Data parallelism [HS86] exists when the same operation has to be done, independently and many times over, on a large amount of data. Examples are doubling every entry in a large table of numbers and finding the sum of a list of values. The latter is an example of a reduction, the bulk of which is data parallel, if the sum is broken into parts using associativity of addition. Data parallelism can be found in linear algebra, manipulating vectors and matrices, and many scientific applications, including modelling physics and predicting the weather.

Data parallel operations can be distributed across multiple processors quite easily, conceptually at least, as illustrated in Figure 1.2. Crucially, the allocation of work to processors is done by the machine, rather than the programmer. At the small scale, data parallelism can also be exploited within a single instruction, using SIMD instructions,<sup>1</sup> such as those in Intel MMX and SSE, IBM/Motorola AltiVec, and ARM NEON.

Data parallel programs can be written in languages such as OpenMP [Org08] and High-Performance Fortran (HPF) [HPFF93; HPFF97]. OpenCL [Khr10] and CUDA [NVI08] support data parallelism on GPUs. Google's MapReduce [DG08] is a form of data parallelism at the large scale.

An alternative is to have the machine look for data parallelism in an ordinary sequential program. If a sequence of similar operations, in a loop, and supposedly to be done in order, do not depend on each other, then they can be done in parallel, and the result will be the same. The earliest work on data dependence analysis dates back to the 1970s [Lam74; KM<sup>+</sup>72; Mur71], and a considerable amount of research has been done since then [AK02].

---

<sup>1</sup>Single Instruction Multiple Data

## 1. INTRODUCTION

---

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	0.957	0.876	0.648	0.794	0.485	0.413	0.929	0.755	0.296	0.271	0.262	0.997	0.280	0.123	0.377	0.672
2	0.116	0.265	0.175	0.189	0.959	0.174	0.769	0.240	0.641	0.112	0.196	0.177	0.772	0.189	0.623	0.273
3	0.853	0.251					0.362	0.283	0.671	0.414					0.520	0.554
4	0.605	0.114	Processor 1					0.770	0.727	0.437	0.717	Processor 2				
5	0.524	0.185						0.365	0.226	0.109	0.450					
6	0.039	0.624						0.387	0.182	0.655	0.423					
7	0.939	0.782						0.352	0.468	0.574	0.927	0.325	0.277	0.618	0.368	0.037
8	0.789	0.211	0.218	0.498	0.117	0.819	0.183	0.762	0.047	0.468	0.132	0.934	0.062	0.191	0.451	0.156
9	0.415	0.051	0.730	0.580	0.917	0.547	0.479	0.297	0.673	0.059	0.383	0.234	0.136	0.724	0.547	0.822
10	0.077	0.798	0.115	0.455	0.189	0.769	0.033	0.062	0.283	0.912	0.231	0.135	0.031	0.146	0.687	0.434
11	0.227	0.375						0.288	0.064	0.576	0.680				0.321	0.551
12	0.302	0.358	Processor 3					0.484	0.158	0.003	0.285	Processor 4				
13	0.171	0.441						0.817	0.996	0.061	0.719					
14	0.095	0.494						0.872	0.721	0.724	0.509					
15	0.169	0.572	0.183	0.802	0.309	0.041	0.156	0.714	0.857	0.474	0.204	0.854	0.107	0.351	0.998	0.898
16	0.003	0.791	0.016	0.091	0.480	0.962	0.165	0.940	0.296	0.087	0.263	0.284	0.585	0.049	0.840	0.315

Figure 1.2: Data parallelism: four cores updating an array of data

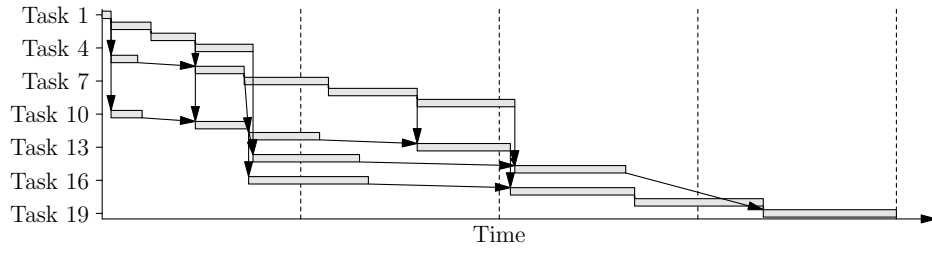


Figure 1.3: Gantt chart illustrating task-level parallelism

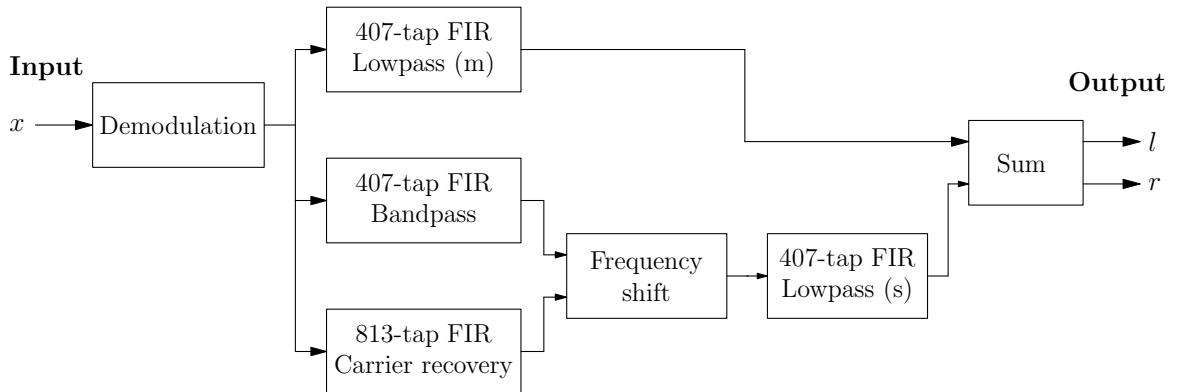


Figure 1.4: Streaming parallelism: GNU radio FM demodulation

## 1.1 Approaches to parallelism

Language	Year <sup>†</sup>	Nesting	Dependencies
Cilk [BJK <sup>+</sup> 95]	1995	Yes	Parent can wait for children
pthread [jee99]	1995	Yes	Shared data and locks
High Performance Fortran (HPF) 2.0 [HPFF97]	1997	Yes	Yes
Jade [RSL02]	2002	Yes	General DAG <sup>1</sup>
StarSs [SP09; BPBL06; PBL08]	2004 <sup>‡</sup>	Yes	General DAG
Sequoia [FHK <sup>+</sup> 06]	2006	Yes	Parent waits for children
Intel Threading Building Blocks [Rei]	2007	Yes	General in Version 3 Update 5
OpenMP 3.0 [Ope09]	2008	Yes	Parent can wait for children
Task Parallel Library (TPL) [LSB09]	2009	Yes	General DAG using futures
StarPU [ATNW09; ATN10]	2009	No	General DAG using tags
OpenCL 1.1 [Khr10]	2010	No	General DAG
OoOJava [JED10]	2010	Yes	General DAG from static analysis
Tagged Procedure Calls (TPC) [TKA <sup>+</sup> 10]	2010	No	Master can wait for tasks
Apple Grand Central Dispatch [App]	2010	Yes	Tasks can wait for other tasks
Intel Cilk Plus [Int10]	2010	Yes	Parent can wait for children

<sup>†</sup> Date of first significant publication

<sup>‡</sup> GridSs

Table 1.1: Selection of task parallel languages and libraries

### 1.1.2 Task-level parallelism

Another way to write parallel programs, known as *task-level parallelism*, is to have one thread, running on one processor, delegate self-contained pieces of work, known as *tasks*, to a pool of interchangeable workers. The workers are normally threads running on different processors. This is easy for people to understand because it resembles concurrency in the real world, and it can be very efficient. Figure 1.3 illustrates task-level parallelism using a Gantt chart.

One of the earliest task-level programming languages is Cilk [BJK<sup>+</sup>95]. In a Cilk program, any thread may *spawn* children, which potentially run in parallel with their parent. If a parent needs some value computed by one of its children, then it should first wait for its children to complete. There can be no dependencies between children; i.e. one child cannot wait for another child to finish, but parallelism can be *nested*, meaning that children can have children of their own.

In this thesis, task parallelism will be defined using StarSs [SP09; BPBL06; PBL08], which will be described briefly in Section 1.4.1. StarSs supports dependencies between tasks: one task may be dependent on the outputs from other tasks, its *predecessors*, meaning that it cannot start until they have finished.

Table 1.1 lists some other languages and libraries for task parallelism, including OpenMP 3.0 [Ope09], TPL (Task Parallel Library) [LSB09] for .NET, Apple’s Grand Central Dispatch [App] in Mac OS X 10.6, and Intel Threading Building Blocks [Rei]. Task parallel programming is also used for specific applications; for example dense linear algebra in the PLASMA [Uni09] and FLAME [FLA10] projects. MPI Microtask [OIS<sup>+</sup>06] for the Cell B.E. breaks a message-passing program into non-blocking units of computation, which are dynamically scheduled as tasks.

## 1. INTRODUCTION

Language	Year <sup>†</sup>	Graph structure	Dimensions
Gedae [LBS]	1987	General directed graph	One-dimensional
Gabriel [BGH <sup>+</sup> 90]	1990	General directed graph	One-dimensional
Ptolemy II [EJL <sup>+</sup> 03]	2003	General directed graph	One-dimensional
StreamIt [TKA02; CAG06]	2001	Series-parallel	One-dimensional
StreamC/KernelC [Mat02]	2002	General directed graph	One-dimensional
DataCutter [BKSS02]	2002	General directed graph	One-dimensional
Brook [Buc03]	2003	General directed graph	Multi-dimensional
S-Net [GSS06]	2006	Series-parallel	One-dimensional
SPM [CRM <sup>+</sup> 07; ACO08]	2007	General DAG	One-dimensional
Fractal [MRC <sup>+</sup> 07]	2007	General directed graph	Multi-dimensional
XStream [GMN <sup>+</sup> 08]	2008	General DAG	One-dimensional
GRAMPS [SFB <sup>+</sup> 09]	2009	General directed graph	One-dimensional
IBM InfoSphere SPL [IBM11]	2009	General directed graph	One-dimensional
DUP [GGR <sup>+</sup> 10]	2010	General directed graph	One-dimensional

<sup>†</sup> Date of first significant publication

Table 1.2: Selection of stream programming languages

### 1.1.3 Streaming parallelism

Another, more specialised, way to write parallel programs is known as *streaming parallelism*. Unlike data and task parallelism, which are applicable to many different types of problem, streaming parallelism is domain-specific. Domain-specific languages (DSLs) have long been used to improve *productivity*. They let the program be described at a higher level, although they lack the generality of a general purpose language such as C. For example, Matlab is for scientific computing, and YACC is for writing parsers of context-free grammars. Other examples include BPEL, Make, Excel, SQL, and DiSTil [SB97; VDKV00].

There has recently been considerable interest in the use of domain-specific languages for *parallelism*. The high-level description reveals implicit parallelism, which would be obscured by a sequential language such as C [DYDS<sup>+</sup>10; CSB<sup>+</sup>11]. Examples of domain-specific languages that expose implicit parallelism include Make, OpenGL [Khr] and SQL [CHM95].

Stream programming is suitable for applications that deal with long sequences of data; these sequences are known as *streams*. Streams arise naturally in time-domain digital signal processing, where the streams are discrete-time signals, and 3D graphics, where the streams are long sequences of vertices and triangles. The objective is usually to maximise throughput, or to reduce power or energy consumption for a given throughput. These applications can usually tolerate a large latency compared with the rate at which data arrives.

It is either impossible or impractical to store the entire streams, so computation is done online, on local subsets of the data. Computation is done inside *kernels*, which communicate in one direction, from producer to consumer, through the streams. In digital signal processing, many of the kernels are digital filters. This representation exposes modularity and regularity to the compiler, enabling the compiler transformations discussed below. It is easy to understand stream programming because it is also like concurrency in the real

<sup>1</sup>Directed acyclic graph

world: a new gadget is being produced at the factory, while older ones are in the hands of the consumers.

Streaming programs can be represented graphically, as in Figure 1.4. This is a picture of an FM stereo demodulation example,<sup>1</sup> which is based on GNU radio [GNU]. Each box represents a kernel, and each edge is a stream that carries data from its producer to its consumer. The seven kernels interact in no other way, so they could be working at the same time, on seven different processors.

Digital-signal processing and 3D graphics are examples of one-dimensional streaming, meaning that the streams are inherently one-dimensional. The kernels are therefore either *stateless*, in which case each stream element is processed independently from the others, or *stateful*, in which case dependencies between elements mean that the kernel must be done sequentially. Video encode and decode are examples of multi-dimensional stream programming. Some of the kernels iterate over a two- or three-dimensional space, and contain data parallelism that can only be exploited using wavefronts.

The reason to write an application as a stream program is that a machine can do a good job of mapping a stream program to the hardware. First, it often happens that some kernels involve much less work than others, and it makes sense to combine several small kernels into one. The programmer can safely write many small kernels, knowing that when there are few processors, small kernels will be merged, and the performance will be as good as if the programmer had done it. Deciding which kernels to fuse together is known as *partitioning*, and is addressed in Section 3.2. Second, unlike task-level programming, there is no master thread that can become the bottleneck. Third, the compiler can choose to batch work up into larger chunks, benefiting from lower overheads, and economy of scale through data reuse and vectorisation. This process is known as *unrolling*. Larger pieces of work have disadvantages too: they require more working memory, so might not fit in cache, and they cause data to take longer to go through the stream program, causing a longer latency.

Fourth, it often happens that some kernels themselves contain data parallelism, so can be divided up to run on several processors in parallel. An example is a volume control, which multiplies each audio sample by some slowly changing value that represents the volume. Since each sample has no influence on any other, there is no state, and the work could be shared among several processors. The FM demodulation example in Figure 1.4 contains several Finite Impulse Response (FIR) filters, which are also stateless.

Streaming parallelism can be reduced to task-level parallelism, by breaking each kernel up into a sequence of tasks. It is usually not possible, however, to go the other way: it is hard for a compiler to deduce kernels and streams from the program source, and then use the optimisation techniques mentioned above.

## 1.2 Compile-time vs run-time decisions

The programmer's job is to write the source code, the human readable instructions to the computer. The computer translates the source code to machine readable object code, a process known as compilation, and will usually perform optimisations at the same time. Optimisations at compile time are effective because redundant or unnecessary work is removed

---

<sup>1</sup>This example is from the ACOTES project (Advanced Compiler Technologies for Embedded Streaming)—see Section 1.3.



## 1. INTRODUCTION

---

for good, regular operations can be collected to run in parallel using SIMD instructions (*vectorisation* [LA00]), and program fragments can be customised for a particular context using constant folding [Muc97], polyhedral optimisations [CGT04] and so on. Moreover, analysis costs are paid just once. A second opportunity for optimisation is at install time, when the executable can be tweaked for the specific machine it is going to run on. This thesis will not distinguish between compile time and install time, and will assume that the compiler knows precisely what the target machine will be. The machine is described to the compiler using the ASM (Abstract Streaming Machine), defined in Chapter 2.

The final opportunity for optimisations is at run time, while the program is actually running. This is when the most information is available, when the program’s behaviour can be observed, and the environment is known: whether other programs need the CPUs, the power source, battery level and CPU temperatures, and so on. Optimisation at run time is effective because it can take account of this extra information, but CPU time is spent doing run-time analysis rather than real work, and program fragments cannot be customised at compile time.

In brief, predictable parts of the stream program will be *statically* partitioned and scheduled; i.e. the decisions will be made at compile-time. Chapter 3 addresses the compile-time decisions. The poorly balanced and unpredictable parts of the stream program will be broken up into tasks to be handled as task-level parallelism, and scheduled *dynamically*, at run time. This is especially important for video applications, including H.264. Chapter 4 addresses dynamic scheduling of stream programs.

### 1.3 The ACOTES stream compiler

This work was partly supported by the ACOTES project [ACO; M<sup>+</sup>11],<sup>1</sup> which defined a stream compiler framework, and developed a demonstration stream compiler. The ACOTES compiler framework partitions a stream program to use task-level parallelism, batches up communications through blocking, and statically allocates communications buffers. The stream program is written using the Stream Programming Model (SPM) [CRM<sup>+</sup>07; ACO08; M<sup>+</sup>11], an extension to the C programming language, developed for the ACOTES project and described in Section 1.5.2.

Figure 1.5 shows the compilation flow. The source program is converted from SPM to C, using the Mercurium [BDG<sup>+</sup>04] source-to-source converter. This step translates pragma annotations into calls to the *acolib* (ACOTES library) run-time system, and inserts calls to the trace collection functions. It fuses kernels that are mapped to the same processor, encapsulating them into routines, and inserts statically sized communication buffers, as required, between kernels on different processors. The mapping is determined by the partitioning algorithm described in Chapter 3. The resulting multi-threaded program is compiled using GCC, which was extended within the ACOTES project to perform blocking, polyhedral transformations, and vectorisation. Additional mapping information is provided to GCC using the Iterative Compilation Interface (ICI) [FC07]. The ACOTES compiler is iterative, meaning that the program may be compiled several times, as the search algorithm adjusts the mapping.

---

<sup>1</sup>Advanced Compiler Technologies for Embedded Streaming (contract no. IST-34869)



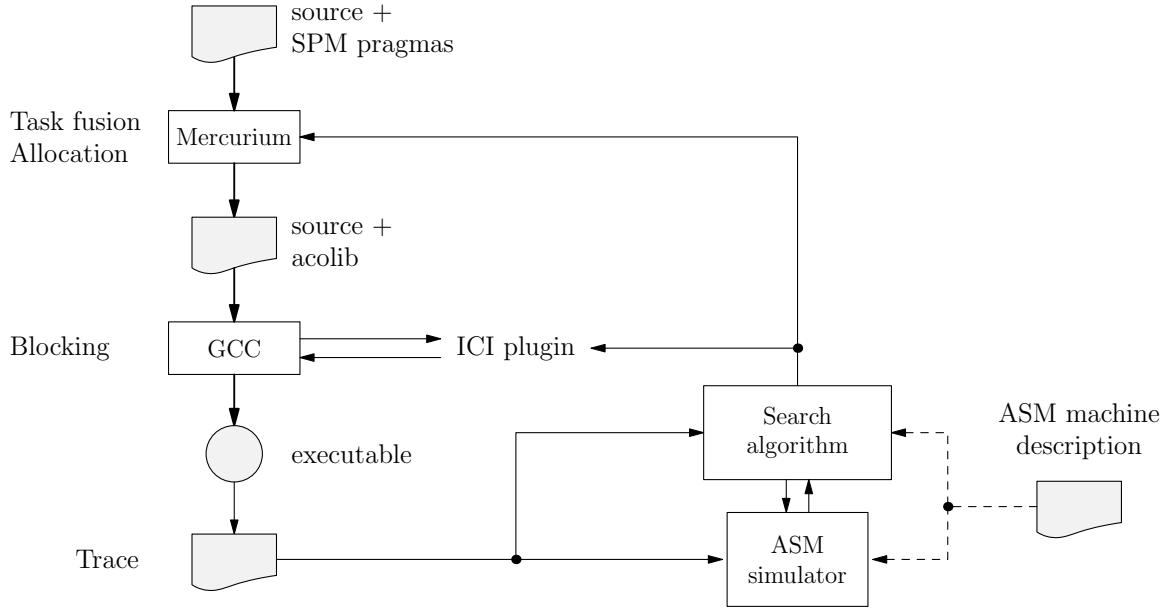


Figure 1.5: The ACOTES iterative stream compiler

The *ASM simulator* executes a mapped stream program at a coarse granularity, and generates statistics which are used to improve the mapping. The inputs to the simulator are the *ASM machine description*, which describes the target, and the *ASM program model*, which describes the program. The ASM simulator is driven by a trace, which allows it to follow conditions and model varying computation times.

The ASM simulator is used inside the small feedback loop in the bottom right of Figure 1.5; for example in determining the size of the stream buffers, using the algorithm in Section 3.3 [CRA10b]. The trace format has been designed to allow a single trace to be reused for several different mappings. Section 2.6 describes in more detail the partitioning algorithm and its interaction with the ASM.

The ASM simulator allowed work to start on the mapping heuristics before the compiler transformation infrastructure was completed. Additionally, and experiments are repeatable because there is no experimental error.

## 1.4 Task-level languages

### 1.4.1 StarSs and OmpSs

Task-level parallelism will be expressed, in this thesis, using Star Superscalar (StarSs), an extension of the C programming language, developed at BSC-CNS.<sup>1</sup> The extensions are “pragmas” that tell the compiler about tasks, but they don’t change what the program does. If a StarSs program is given to an ordinary compiler, that does not support StarSs, it will ignore the pragmas, and the program will still work—just not in parallel.

<sup>1</sup>Barcelona Supercomputing Center–Centro Nacional de Supercomputación.

## 1. INTRODUCTION

---

```
1 #pragma css task input(row,col) inout(inner)
2 void bmod(float row[32][32],
3          float col[32][32],
4          float inner[32][32])
5 {
6     for (int i=0; i<32; i++)
7         for (int j=0; j<32; j++)
8             for (int k=0; k<32; k++)
9                 inner[i][j] -= row[i][k]*col[k][j];
10 }
```

Figure 1.6: Example StarSs code (*bmod* function from LU factorisation)

StarSs is an umbrella term for the common parts of GRID Superscalar [SP09], Cell Superscalar (CellSs) [BPBL06], and SMP Superscalar (SMPSs) [PBL08], plus generalisations for GPUs (Graphics Processing Units) and clusters. CellSs targets the Cell Broadband Engine [CRDI05], which has distributed memories, and needs the run-time system to program DMA transfers to get data from one processor to another. SMPSs targets Symmetric Multiprocessors, which have a global shared address space, so DMA transfers are not required. For more information please refer to the conference publications cited above and the CellSs [Bar09] and SMPSs [Bar08] manuals.

OmpSs is an implementation of StarSs, which also integrates the OpenMP standard. The OmpSs compiler accepts both the StarSs syntax and the newer OmpSs syntax [DFA<sup>+</sup>09].

Figure 1.6 shows a short function, the *bmod* function from LU factorisation, annotated with its StarSs pragma. This function works on subblocks of larger arrays: the subblocks called *row* and *col* are inputs, and therefore not modified by the function, and the subblock called *inner* is read and modified (*inout*). Pragas contain some extra information, beyond the direction of data transfer. First, a pragma says that the function should be a task in the first place; i.e. that it is big enough, and worth offloading to another processor, given the overheads of a software run-time.<sup>1</sup> Second, when the formal parameter is a pointer rather than an array, the pragma gives the array's size (an example is given in Figure 5.12(b) on page 111). This information is needed, but otherwise not given by the C source.

The program starts executing on one processor, in the master thread. When the master thread calls a function like *bmod* marked as a task, the function does not execute right away. Instead, a task, to do that function, is added to a run-time dependency graph, to be executed some time in the future in a worker thread.

The dependency graph tracks dependencies between tasks, since some of the inputs to *bmod* may not be ready yet. They may be outputs from tasks that haven't yet been done. Not all task-level languages allow tasks to depend on earlier tasks, instead needing the main thread to wait for the predecessors to complete before it can fire off any task that needs their outputs. Supporting dependencies is an important feature of StarSs, and it is easy to imagine that it might improve performance.

Figure 1.7 shows the function prototypes and part of the compute function from the Cholesky decomposition benchmark. Figure 1.8 shows dependency graphs for two different problem sizes.<sup>2</sup> The different colours correspond to the four functions, and the numbers in subfigure (a) show the order in which the tasks were created. In practice, the whole

---

<sup>1</sup>Task Superscalar [ERB<sup>+</sup>10] and D<sup>2</sup>NOW [KET06] propose hardware support for fine-grained tasks.

<sup>2</sup>The problem size is given by the value of *N* in the source code.

```

1 #pragma css task input(a, b) inout(c)
2 void sgemmm_t(float a[M][M], float b[M][M],
3             float c[M][M]);
4
5 #pragma css task inout(a)
6 void spotrf_t(float a[M][M]);
7
8 #pragma css task input(a) inout(b)
9 void strsm_t(float a[M][M], float b[M][M]);
10
11 #pragma css task input(a) inout(b)
12 void ssyrk_t(float a[M][M], float b[M][M]);
13
14 ...
15
16 float A[N][N][M][M]; // NxN blocked matrix,
17                       // with MxM blocks
18
19 for (int j = 0; j < N; j++)
20 {
21     for (int k = 0; k < j; k++)
22         for (int i = j+1; i < N; i++)
23             sgemmm_t(A[i][k], A[j][k], A[i][j]);
24
25     for (int i = 0; i < j; i++)
26         ssyrk_t(A[j][i], A[j][j]);
27
28     spotrf_t(A[j][j]);
29
30     for (int i = j+1; i < N; i++)
31         strsm_t(A[j][j], A[i][j]);
32 }

```

Figure 1.7: Extracts from StarSs code for Cholesky decomposition

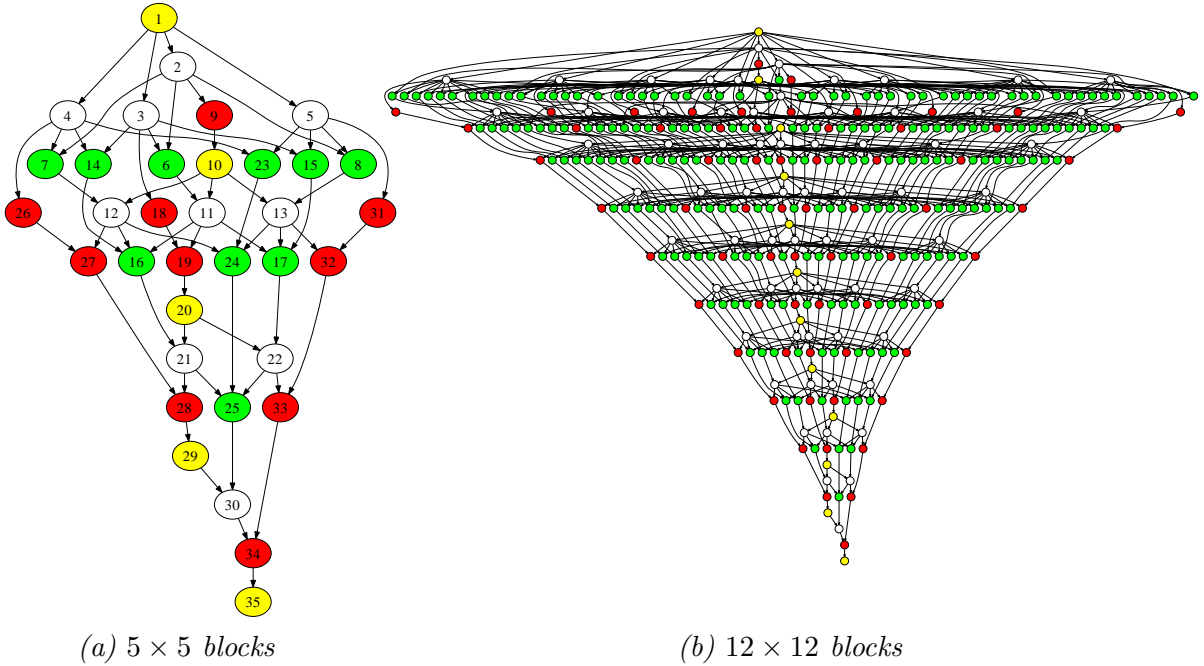


Figure 1.8: Dependency graphs for Cholesky decomposition

## 1. INTRODUCTION

---

dependency graph may never exist in this complete form, because tasks appear only after they have been created, and are removed once they have been done.

Since several versions of one particular array may be in flight at once, the OmpSs runtime *renames* arrays to break false dependencies, in a similar way to register renaming in a superscalar processor.<sup>1</sup> Even if task B modifies an array that is going to be needed by task A, task B can be offloaded and even start executing before task A has finished. Task A will always read the version it was supposed to, even if that version has since been superseded by a new one. If the main thread needs the output of some task, it must request the data using a pragma. The pragma waits for the task to complete and ensures that the live data is copied back.

The compiler could potentially have looked through the function in Figure 1.6, and written the pragma itself, without a person having to do it. This function is rather easy to analyse statically, but other functions are harder.<sup>2</sup> Alternatively, it could watch the function, while the program is running, and see that it never seems to modify `row`, for example, guessing that the annotation should be `input(row)`. This approach needs a lot of test cases to ensure that no special cases are overlooked. StarSs could therefore be used as a form of intermediate language, to separate machine analysis from the run-time system.

In any case, we are interested here in using the OmpSs runtime to implement dynamic scheduling for a streaming language, by splitting kernels into tasks. So the pragmas are going to be generated by the machine anyway.

## 1.5 Streaming languages

### 1.5.1 StreamIt language

StreamIt [TKA02; CAG06] is a programming language for streaming applications, developed at MIT. All computation is performed by *filters*, connected together by *FIFOs*, with decentralised control rather than a master thread. Each FIFO carries data from a single producer filter to a single consumer filter. Data arrives at the consumer in the order sent; that is first-in first-out.

The stream graph is built hierarchically, using *pipelines*, *splitjoins*, and *feedback loops*. These *components*<sup>3</sup> are illustrated in Figure 1.9. All components, including filters, have a single input FIFO and a single output FIFO.

A **pipeline** connects its children in sequence, so the output of each child component is the input to the next—see Figure 1.9(a). A **splitjoin** connects its children in parallel, as shown in Figure 1.9(b). The input FIFO is split among its children, either by duplication, or by distributing its elements in a weighted round robin fashion. The output FIFOs are always joined using weighted round robin. Splits and joins are the only vertices in the stream graph

---

<sup>1</sup>The current implementation of OmpSs does not support renaming.

<sup>2</sup>Deciding whether an argument is *inout* rather than *input*, for instance, is NP-hard, since a function could modify an argument only after discovering that an NP-hard decision problem can be solved. A pointer passed to `sprintf` is written through only in the uncommon case that the format directive is `%n`, something that the compiler may not easily be able to rule out.

<sup>3</sup>The StreamIt term for a filter, pipeline, splitjoin, or feedback loop is a “stream”. Since in this thesis, streams are the edges that carry data, this section uses the word component instead.

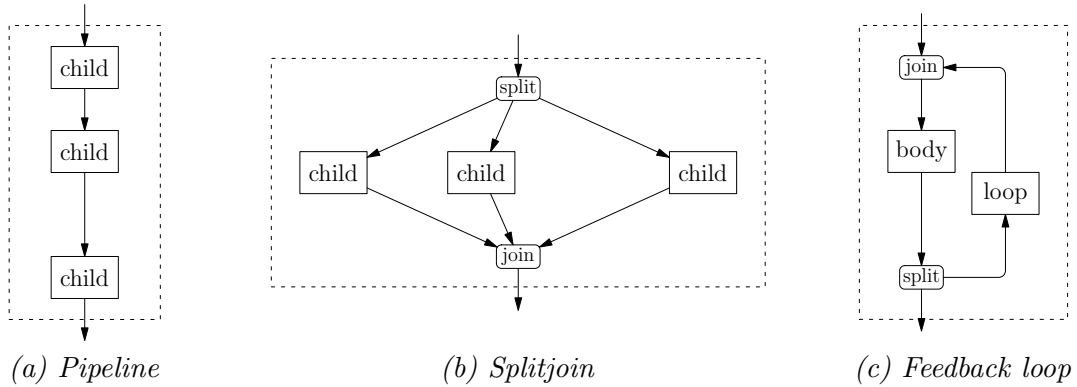


Figure 1.9: StreamIt components: pipelines, splitjoins, and feedback loops

that can have more than one predecessor or successor; but they only distribute or gather data, rather than being free to do arbitrary work.

A **feedback loop** introduces a feedback path from the output of its *body* child back to its input, as shown in Figure 1.9(c). The feedback path optionally contains a component, referred to as the *loop*. It may need one or more elements prequeued onto the output of the loop component so that it does not deadlock.

Figure 1.10 shows an example StreamIt program, which is a bandpass filter, implemented as the difference of two lowpass filters.<sup>1</sup> The stream graph is deduced at compile time, and is shown in Figure 1.11. The top level of the program is the pipeline defined in lines 54 to 65. It contains four children, inserted using the `add` statements, which in turn read the input, calculate the two lowpass filters, subtract elements in pairs to get the difference, and write the output.

The two lowpass filters are contained in the splitjoin called `BPFCore`, defined in lines 40 to 45. The splitter duplicates each element on the input FIFO to go to both filters, the filters happen on its two branches, and the joiner interleaves their outputs.

The lowpass filter is defined in lines 15 to 38. It is of type `float→float`, meaning that its input FIFO holds elements of type `float`, and so does its output FIFO. Line 16 declares its local data: a coefficient array whose number of elements is equal to `taps`. This array is built by the initialisation function in lines 17 to 28.

Every filter has a *work function* that keeps getting called, until the program has finished. In this case the work function is defined in lines 29 to 38. Each time this work function is called; i.e. each time it *fires*, it pops a fixed number of elements from its input FIFO and pushes a fixed number of elements into its output FIFO. The StreamIt 2.1 language supports work functions that push or pop a variable number of elements; in which case, the programmer may specify the minimum, maximum, and average number of elements. All StreamIt benchmarks, however, currently have fixed rates.

Here, the number of elements popped is equal to `1 + decimation`. The exact number must be determinable at compile time. Since the filters are added in lines 42 and 43, both times with `decimation` equalling zero, both filters pop one element each time they fire. This is known in Figure 1.11, which was produced by the StreamIt compiler. Similarly, they both

<sup>1</sup>This is inefficient because the two filters could be merged into one, using linearity, but the example is taken from the `fm` StreamIt benchmark, which works like this.

## 1. INTRODUCTION

---

```
1  /*
2  *   Copyright 2001 Massachusetts Institute of Technology
3  *
4  *   Permission to use, copy, modify, distribute, and sell this software and its
5  *   documentation for any purpose is hereby granted without fee, provided that
6  *   the above copyright notice appear in all copies and that both that
7  *   copyright notice and this permission notice appear in supporting
8  *   documentation, and that the name of M.I.T. not be used in advertising or
9  *   publicity pertaining to distribution of the software without specific,
10 *   written prior permission. M.I.T. makes no representations about the
11 *   suitability of this software for any purpose. It is provided "as is"
12 *   without express or implied warranty.
13 */
14
15 float->float filter LowPassFilter(float rate, float cutoff, int taps, int decimation) {
16     float[taps] coeff;
17     init {
18         int i;
19         float m = taps - 1;
20         float w = 2 * pi * cutoff / rate;
21         for (i = 0; i < taps; i++) {
22             if (i - m/2 == 0)
23                 coeff[i] = w/pi;
24             else
25                 coeff[i] = sin(w*(i-m/2)) / pi / (i-m/2) *
26                     (0.54 - 0.46 * cos(2*pi*i/m));
27         }
28     }
29     work pop 1+decimation push 1 peek taps {
30         float sum = 0;
31         for (int i = 0; i < taps; i++)
32             sum += peek(i) * coeff[i];
33         push(sum);
34         for (int i=0; i<decimation; i++)
35             pop();
36         pop();
37     }
38 }
39
40 float->float splitjoin BPFCore (float rate, float low, float high, int taps) {
41     split duplicate;
42     add LowPassFilter(rate, low, taps, 0);
43     add LowPassFilter(rate, high, taps, 0);
44     join roundrobin;
45 }
46
47 float->float filter Subtractor {
48     work pop 2 push 1 {
49         push(peek(1) - peek(0));
50         pop(); pop();
51     }
52 }
53
54 void->void pipeline SimplifiedFMRadio5 {
55     float samplingRate = 250000000; // 250 MHz
56     float cutoffFrequency = 108000000; // 108 MHz
57     float low = 55.0;
58     float high = 97.998856;
59     int taps = 128;
60
61     add FileReader<float>("input.in");
62     add BPFCore(samplingRate, low, high, taps);
63     add Subtractor();
64     add FileWriter<float>("output.out");
65 }
```

Figure 1.10: Example StreamIt 2.1 program, based on FMRadio5

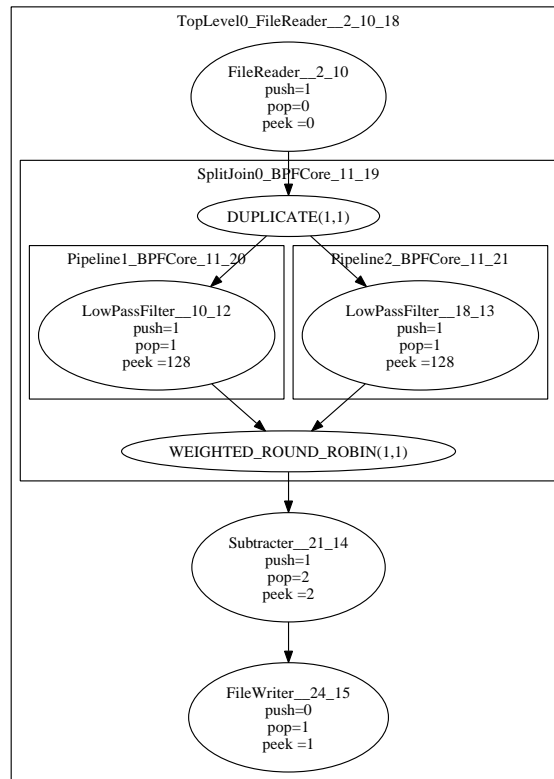


Figure 1.11: Stream graph for the example StreamIt 2.1 program

## 1. INTRODUCTION

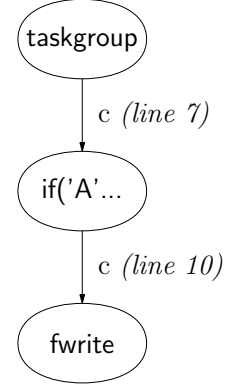
---

```

1 int main()
2 {
3     char c;
4     #pragma acotes taskgroup
5     while (fread(&c, sizeof(c), 1, stdin))
6     {
7         #pragma acotes task input(c) output(c)
8         if ('A' <= c && c <= 'Z')
9             c = c - 'A' + 'a';
10
11     #pragma acotes task input(c)
12         fwrite(&c, sizeof(c), 1, stdout);
13     }
14     return 0;
15 }

```

(a) SPM source code for *tolower*



(b) Streaming graph  
(flattened)

Figure 1.12: Example SPM program from [M<sup>+</sup>11]

push one output element each time they fire, and they look ahead, or *peek* 128 elements in the stream, counting from the starting point. That is, they need 127 elements in addition to the one they will pop.

Each filter is *stateless*, meaning that the calls to their work functions are independent, so the filter contains data parallelism. A *stateful* filter has dependencies from one firing of the work function to the next, because its work function writes to local data. For example, it could be an adaptive filter that modifies its coefficient array, `coeff`. The compiler would see that the work function modifies local data, and make sure that it is done sequentially.

This example illustrates the features of StreamIt that are important in this thesis. For more information about StreamIt, refer to the language definition [CAG06], which is only twenty pages in length. The example does not illustrate *static blocks*, which contain read-only data visible to all filters. It also does not illustrate some of the more advanced features unused by the StreamIt benchmarks, and not supported by our conversion tool: feedback loops, described above, *messaging*, which is a mechanism for asynchronous messaging between tasks, *prework* functions, which replace the work function the first time a kernel fires, or support for *variable push and pop rates*.

This dissertation uses the StreamIt 2.1 language definition [CAG06], and the compiler and benchmarks from the StreamIt 2.1.1 distribution, dated January 2007.

### 1.5.2 SPM (Stream Programming Model)

The SPM [CRM<sup>+</sup>07; ACO08; M<sup>+</sup>11] (Stream Programming Model) is an extension to the C programming language to support stream programming, developed in the ACOTES project. Whereas StreamIt is a new language that requires the program to be restructured to look like a streaming program, SPM uses pragmas to annotate an ordinary C program. The pragmas identify parts of the program to be made into kernels. Like StarSs, if an SPM program is given to an ordinary C compiler, which doesn't support SPM, it will ignore the pragmas, and the program will still work. That is, SPM maintains the sequential semantics of C.



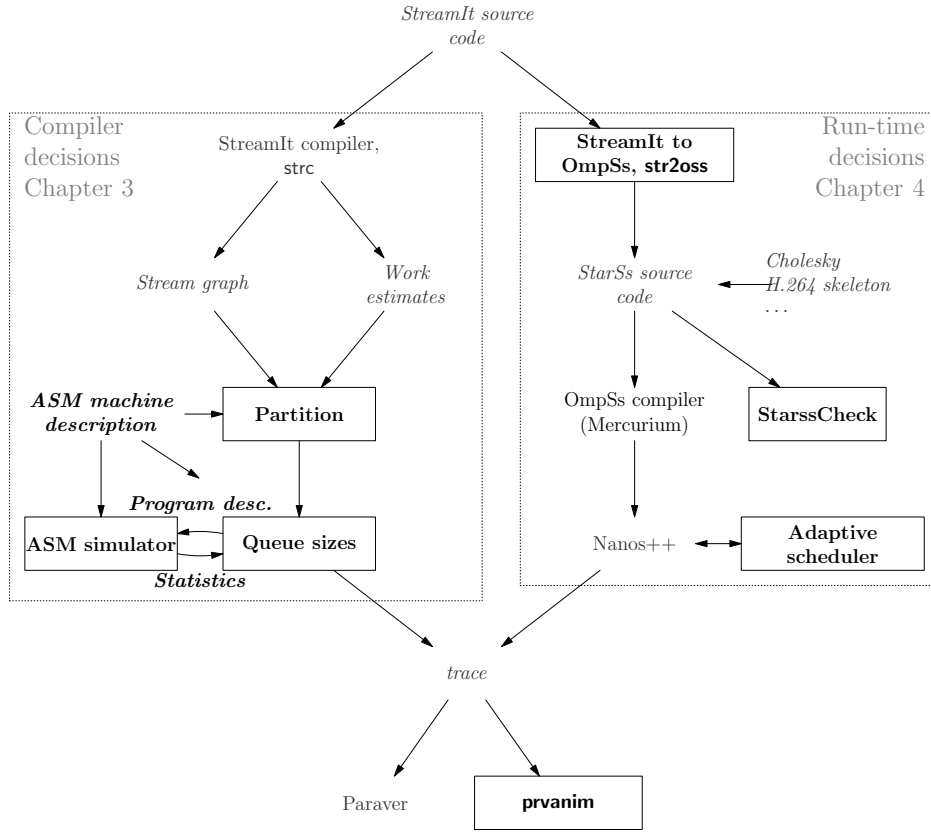


Figure 1.13: Tool flow used in this thesis. Items in bold are contributions of this thesis.

Figure 1.12 shows an example program using the SPM, and its stream graph. The streaming part of a program is known as a *taskgroup*, and it comprises the loop following the `acotes taskgroup` pragma, here on lines 5 to 13. This taskgroup has two *tasks*, each of which contains the statement or block following an `acotes task` pragma. The task’s inputs and outputs are identified using the input and output clauses.

The SPM program begins running in a single thread. When execution reaches the taskgroup, all of its tasks are created, each becoming a kernel in the language of this thesis, and the program starts processing data in streams, passing inputs and outputs through the streams. More information can be found in the ACOTES documentation [CRM<sup>+</sup>07].

## 1.6 Tool flow and support tools

Figure 1.13 shows the tool flow used in this thesis, with the contributions of the thesis shown in bold. The left-hand side of the diagram is for the compile-time decisions in Chapter 3, and the right-hand side is for the run-time decisions in Chapter 4.

On the left, the StreamIt source code is compiled using the StreamIt compiler, which, in addition to creating an executable, also creates a file representing the stream graph, in dot format [GKN06], and a text file giving its estimate, for each filter, of the amount of work per firing. These files are the input, together with the ASM machine description, to

## 1. INTRODUCTION

---

the partitioning algorithm in Section 3.2. After partitioning, the buffer sizes are determined using the buffer sizing algorithm in Section 3.3.

The right-hand side of the diagram is for the run-time decisions in Chapter 4. The StreamIt source code is first translated to StarSs, using the tool described in Section 5.3. The StarSs source code is built using the Mercurium OmpSs compiler.

The various dynamic scheduling techniques, including the two proposed adaptive techniques, were compared using a set of OmpSs benchmarks built in this way.

### 1.6.1 Debugging using StarssCheck

When people start using a new programming language and compiler, they will soon discover that some of their programs don't work. They will need to find out why, before they can fix them. A programming language without debug tools may be a fine research vehicle, but it is unlikely to be widely adopted, as users become frustrated by bugs in their code, blame the compiler, and think that the language is hard to use.

Section 5.1 describes a debugging tool, *StarssCheck* [CRA10a], that was developed as part of this thesis. It finds bugs in StarSs programs, but similar ideas could be used in a tool supporting SPM. StarssCheck was used, for example, to check the output of the StreamIt to OmpSs converter described in Section 5.3. The reasons for targeting StarSs rather than the SPM are that StarSs is more mature, and it already has real users.

### 1.6.2 Performance visualisation

In this thesis, the main reason to bother to write and compile an application to run in parallel is to improve its performance. When the performance is disappointing, or in some way surprising, it is important to understand why, and this requires some way to see how the application progresses.

The main tools used for this purpose in this thesis were Paraver [CEP] and *prvanim* (Paraver Animator). Paraver is a trace visualisation tool, developed at BSC-CNS. Paraver reads a trace in a straightforward format [CEP01], which is a text file that can be easily created either using custom code or the Mintaka library [Nanb].

Section 5.2 describes the *prvanim* tool, which was developed in the course of this thesis. It takes a Paraver trace, and produces an animation that shows the progress of the application through time. It is a simple tool, which has, nonetheless, proven quite useful.

### 1.6.3 StreamIt to OmpSs

In order to use the same StreamIt benchmarks throughout the thesis, these benchmarks had to be translated to StarSs, so they could be compiled by the OmpSs compiler. For that purpose, we developed *str2oss*, a source-to-source compiler that translates from StreamIt to StarSs. This tool is described in Section 5.3. It does not support kernel fusion, but it does support unrolling, using a control file given by the user.

Each filter is translated into a work function, and, if required, an initialisation function. The work function does the work required by one firing of the filter, and is marked as a StarSs task. The tool creates a main thread, which allocates memory for the streams, and implements the steady state by calling the tasks in sequence.

## 1.7 Contributions and publications

The main contributions and publications of this thesis are:

The **Abstract Streaming Machine (ASM)**, a machine model and coarse-grain simulator for a statically scheduled stream compiler.

- [CRM<sup>+</sup>07] Paul Carpenter, David Rodenas, Xavier Martorell, Alejandro Ramirez, and Eduard Ayguadé. A streaming machine description and programming model. Proc. of the International Symposium on Systems, Architectures, Modeling and Simulation, Samos, Greece, July 16–19, 2007.

*Early versions of the ASM and SPM, giving an example machine description and program description in Python syntax. The sections related to the SPM do not describe a contribution of this thesis.*

- [ACO08] ACOTES. IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version. 2008.

*The final version of the ASM from the ACOTES project. Chapter 2 is based on the material in this report.*

- [CRA09b] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. The Abstract Streaming Machine: Compile-Time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors. In SAMOS Workshop 2009, pages 12–13. *Best paper award.*

*A condensed version of the ACOTES Deliverable D2.2.*

- [CRA11] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. The Abstract Streaming Machine: Compile-Time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors. Transactions on HiPEAC, 5(3).

*Extended version of the paper in the SAMOS Workshop 2009, similar to the ACOTES Deliverable D2.2.*

- [M<sup>+</sup>11] Harm Munk et al. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. International Journal of Parallel Programming, 39:397–450, 2011.

*Paper presenting the outcomes of the whole ACOTES project. Only the parts relating to the ASM describe a contribution of this thesis.*

---

A new **partitioning heuristic** for stream programs, which balances the load across the target, including processors and communication links. It considers its effect on downstream passes, and it models the compiler’s ability to fuse kernels.

- [CRA09a] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Mapping Stream Programs onto Heterogeneous Multiprocessor Systems. In CASES ’09: Proceedings of the 2009 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, pages 57–66, 2009.

## 1. INTRODUCTION

---

Two static **queue sizing algorithms** for stream programs, which determine the sizes of the buffers used to implement streams. The optimal buffer sizes are affected by latency and variability in computation costs, and are constrained by the sizes of local memories, which may be small.

- [CRA10b] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Buffer Sizing for Self-timed Stream Programs on Heterogeneous Distributed Memory Multiprocessors. In High Performance Embedded Architectures and Compilers, 5th International Conference, HiPEAC 2010, pages 96–110.
- 

Two new low-complexity **adaptive dynamic scheduling algorithms** for stream-like programs.

---

**StarssCheck, a debugging tool for StarSs.** This tool checks memory accesses performed by tasks and the main thread, and warns if the StarSs pragmas are incorrect.

- [CRA10a] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs. Euro-Par 2010–Parallel Processing, pages 2–13, 2010.
- 

Figure 1.15 shows a timeline, which places the contributions of this thesis in context. It also shows the two European projects that supported this work. The ACOTES project, which ran from mid-2006 to mid-2009, partially supported the ASM and compile-time heuristics. The ENCORE project, which started in March 2010, partially supported the run-time work, which includes StarssCheck and str2oss. The timeline also shows some external milestones, related to the Cell B.E., StarSs and StreamIt.

### 1.8 Thesis outline

Figure 1.14 shows the structure of the rest of this thesis. Chapter 2 describes the ASM, the Abstract Streaming Machine, that characterises the machine on which the stream program is going to run. The ASM tells the compiler how many processors there are, and how they are connected. Chapter 3 concerns the decisions made at compile time: the partitioning algorithm, and an algorithm that decides how big the communications buffers should be. The output from the compiler is an executable using the Nanos++ library [Nana]. This library includes an implementation of the OMP Superscalar runtime. Chapter 4 deals with run-time scheduling of stream programs.

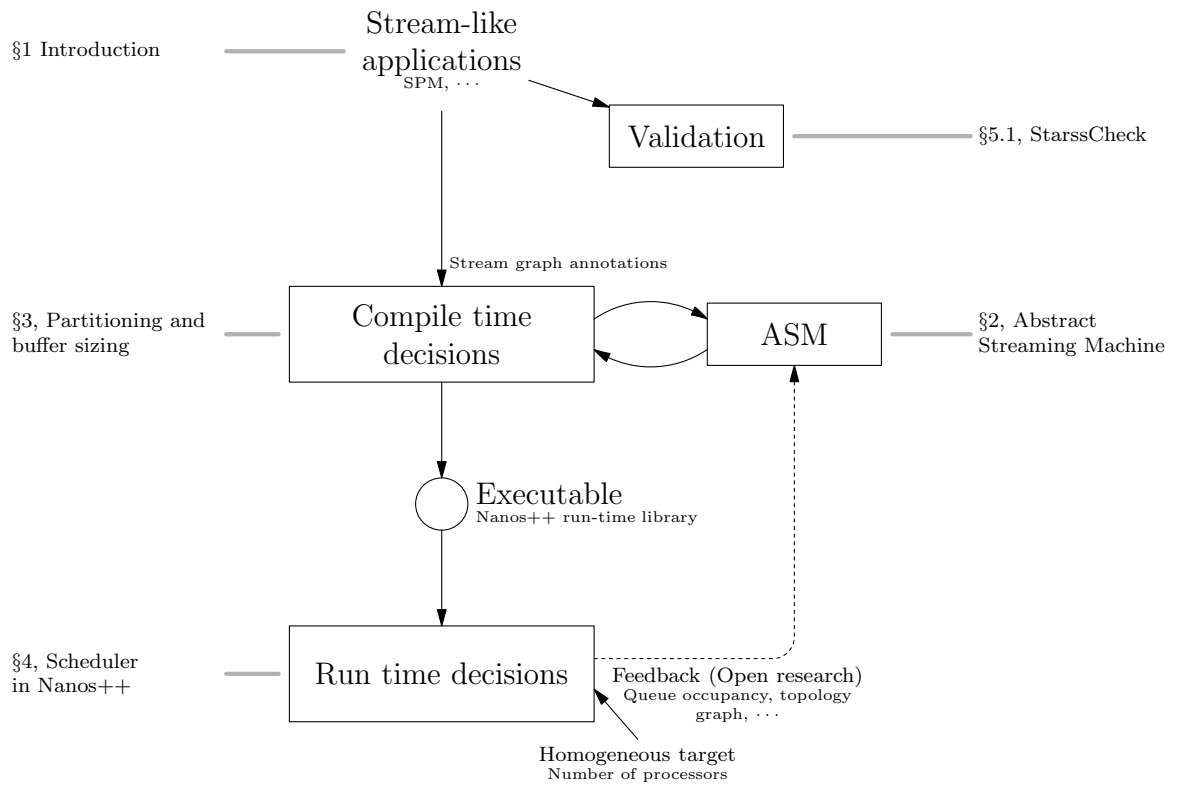


Figure 1.14: Thesis structure

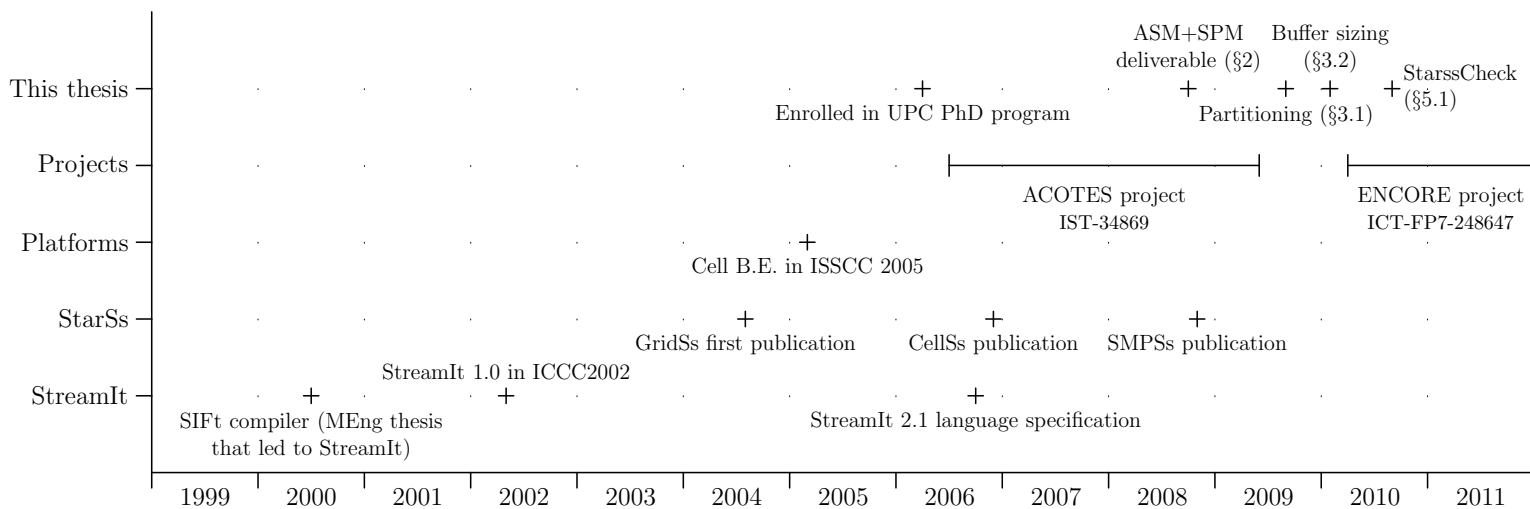


Figure 1.15: Timeline

## Chapter 2

# Abstract Streaming Machine

The first part of this thesis is concerned with the stream compiler. The stream compiler builds an executable that runs efficiently on the target machine, applying, amongst others, the transformations explained in Chapter 3. These transformations require the target to be characterised using some kind of model. This chapter describes the Abstract Streaming Machine (ASM), the machine description and performance model developed as part of the ACOTES project [M<sup>+</sup>11]. The ACOTES project developed the framework for a stream compiler illustrated in Figure 2.1, and discussed in Section 1.3.

There are two parts to the ASM. The *ASM machine description* describes the system architecture of the target as a graph, and is sufficient for the partitioning algorithm, which balances loads on the processors and interconnects. Other transformations, such as queue sizing, need to see the system’s dynamic behaviour. When the program’s behaviour is simple and regular, an analytical model of its progress may suffice. For example, the *critical cycle* algorithm discussed in Section 3.14 assumes that the program repeats exactly the same operations, with the same execution times and communication latencies over and over, and represents the whole program’s execution using a cyclic directed graph. When the program’s behaviour is irregular, there is no alternative to simulation or real execution.

The *ASM simulator* is a coarse-grain simulator, which models the execution of a mapped stream program on the target, producing statistics and a Paraver trace. The simulator can itself be driven by a trace, obtained from a real execution, which allows it to follow conditions and model varying computation times. The ASM simulator needs an execution model for the program, and this is given by the *ASM program model*. The trace format and program model have been designed to allow a single trace to be reused for several different mappings.

### 2.1 Scope of the ASM

The ASM supports homogeneous and heterogeneous targets; that is, the processors’ instruction sets and microarchitectures may or may not be all the same. A *homogeneous* multiprocessor, comprised of identical cores, is easier to design, and easier to program. A heterogeneous target, in contrast, may have lower power consumption because different cores can be tuned for different functions [CCG<sup>+</sup>00], and it mitigates Amdahl’s law [Amd67; KTJR05], since work that cannot be done in parallel can be done by a more powerful, but less frugal, processor.

## 2. ABSTRACT STREAMING MACHINE

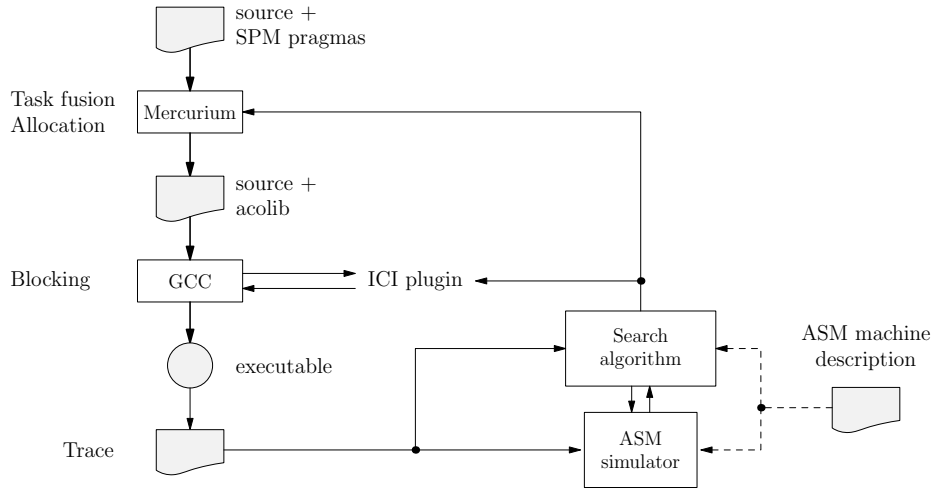


Figure 2.1: The ACOTES iterative stream compiler, reproduced from Figure 1.5

The ASM also supports both shared and distributed memory. Shared memory is required by many multithreaded C programs, so is common for machines with about 32 or fewer cores, but it is expensive to scale *cache-coherent* shared memory to large numbers of processors.<sup>1</sup> The cost to implement cache coherence is perhaps 5–10% of the total power consumption of the data cache accesses [ESD02] in an eight-way SMP, and the cost grows with the number of processors. Distributed memory is easier to implement in hardware, but it is harder to program because data can only be sent between cores using explicit messages such as *send-receive* (two-sided communication) or *get-put* (one-sided communication).

The ASM models shared or distributed memory only as required by the transformations in a stream compiler. If memory is shared, it does not model sharing granularity, whether the shared memory is coherent or not, and the intricacies of the consistency model [AG02]. All of the above differ significantly between implementations, but are needed only by the run-time developer. Similarly, the ASM does not tell the run-time developer how to program DMA transfers.

The ASM is also not needed for code generation, which produces object code for individual functions, which run on a single core. This problem is addressed quite adequately by existing compilers, and it would be both pointless and prohibitively expensive to try to duplicate.<sup>2</sup>

The ASM is also the execution model for the ASM simulator. ACOTES supports static scheduling, and has no support for dynamic scheduling. The ASM program model, therefore is a model for statically scheduled stream programs, which is the focus of Chapter 3.

<sup>1</sup>Expensive, but not impossible. The Blacklight supercomputer contains two SGI Altix UV subsystems, each cache-coherent shared memory with 2,048 processor cores.

<sup>2</sup>Regarding the latter point, GCC 4.5.2 has 300,000 lines of target-specific machine description (.md) files.



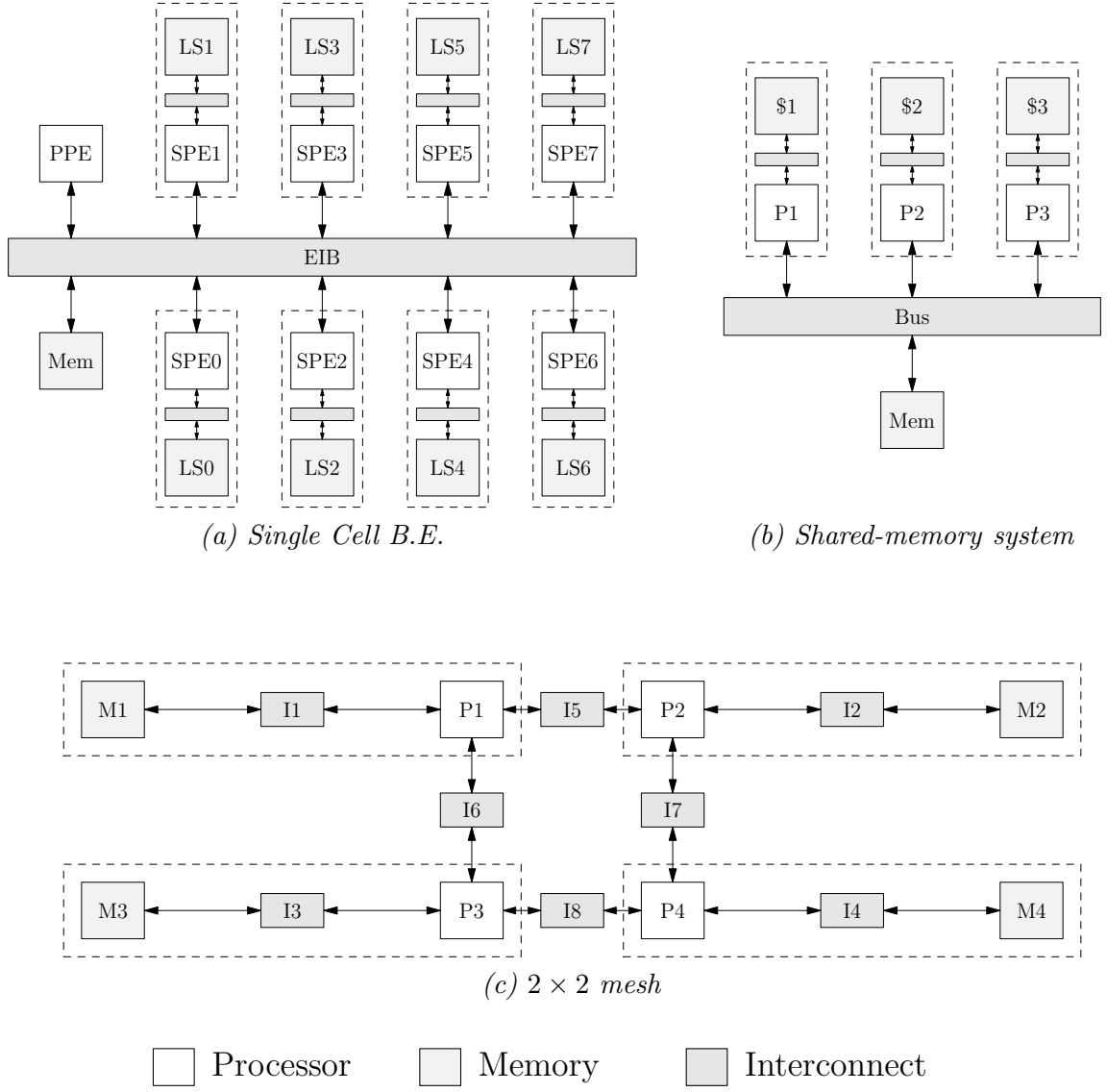


Figure 2.2: Topology of three example targets

### 2.2 ASM Machine Description

The target is represented as a undirected bipartite graph  $H = (V, E)$ , where  $V$  is the set of vertices, which represent *resources*, and  $E$  is the set of edges, which represent *interconnects*. The resources are the *processors* and the *memories*. Figure 2.2 shows the topology of three example targets. The machine description defines the machine visible to software, provided by the OS and *acolib*, which may be different from the physical hardware. For example, Playstation 3 has a Cell B.E. (processor) [CRDI05], which has one PPE and eight SPE accelerators, but the Operating System makes just six of the SPEs available to software; furthermore, the OS does not reveal the mapping from virtual to physical processor. We also assume that the processors used by the stream program are not time-shared with other applications while the program is running.

Figure 2.3 shows the parameters used to characterise each resource in the system, together with their values for the Cell B.E. with the Cell implementation of *acolib*, and estimated values for an SMP. Each processor core has a separate definition, allowing the ASM to support both heterogeneous and homogeneous systems.

Each processor is defined using the parameters in Figure 2.3(a). As discussed above, the details of the processor’s ISA and micro-architecture are already described in the compiler’s back-end, so are not duplicated in the ASM. The ASM processor description lists the costs of the *acolib* calls. The costs of `ProducerSend` and `ConsumerAcquire` are given by a staircase function; i.e. a fixed cost, a block size, and an incremental cost for each complete or partial block after the first. This variable cost is necessary both for FIFOs and for distributed memory with DMA. For distributed memory, the size of a single DMA transfer is often limited by hardware, so that larger transfers require additional processor time in `ProducerSend` to program multiple DMA transfers. The discontinuity at 16KB in Figure 2.10, seen on the Cell B.E., is due to this effect.

The `addressSpace` and `hasIO` parameters provide constraints on the compiler mapping, but are not required to evaluate the performance of a valid mapping. The former defines the local address space of the processor; i.e. which memories are directly accessible and where they appear in local virtual memory, and is used to place stream buffers. The model assumes that the dominant bus traffic is communication via streams, so either the listed memories are private local stores, or they are shared memories accessed via a private L1 cache. In the latter case, the cache should be sufficiently effective that the cache miss traffic on the interconnect is insignificant. The `hasIO` parameter defines which processors can perform system IO, and is a simple way to ensure that tasks that need system IO are mapped to a capable processor.

Each interconnect is defined using the parameters shown in Figure 2.3(b). The system topology is given by the `elements` parameter, which for a given interconnect lists the adjacent processors and memories. Each interconnect is modelled as a bus with multiple channels, which has been shown to be a good approximation to the performance observed in practice when all processors and memories on a single link are equidistant [GLB00]. If there are more messages than channels, then messages have to wait, and are arbitrated using a first-come-first-served policy. There is a single unbounded queue per bus to hold the messages ready to be transmitted. The compiler statically allocates streams onto buses, but the choice of channel is made at runtime. The `interfaceDuplex` parameter defines for each resource; i.e. processor or memory, whether it can simultaneously read and write on different channels.

## 2.2 ASM Machine Description

Parameter	Description	Cell	SMP
name	Unique name in platform namespace	'SPE $n$ '	'CPU $n$ '
clockRate	Clock rate, in GHz	3.2	2.4
hasIO	True if the processor can perform IO	False	True
addressSpace	List of the physical memories addressable by this processor and their virtual address	[(LS $n$ ,0)]	[('Mem',0)]
pushAcqCost	Cost, in cycles, to acquire a producer buffer (before waiting)	448	20
pushSendFixedCost	Fixed cost, in cycles, to push a block (before waiting)	1104	50
pushSendUnit	Number of bytes per push transfer unit	16384	0
pushSendUnitCost	Incremental cost, in cycles, to push pushUnit bytes	352	0
popAcqFixedCost	Fixed cost, in cycles, to pop a block (before waiting)	317	50
popAcqUnit	Number of bytes per pop transfer unit	16384	0
popAcqUnitCost	Incremental cost, in cycles, to pop popUnit bytes	0	0
popDiscCost	Cost, in cycles, to discard a consumer buffer (before waiting)	189	20

(a) *Definition of a processor*

Parameter	Description	Cell	SMP
name	Unique name in platform namespace	'EIB'	'FSB'
clockRate	Clock rate, in GHz	1.6	0.4
elements	List of the names of the elements (processors and memories) on the bus	['PPE', 'SPE0',..., 'SPE7']	['CPU0', ..., 'CPU3']
interfaceDuplex	If the bus has more than one channel, then define for each processor whether it can transmit and receive simultaneously on different channels	[True, ..., True]	[False, ..., False]
interfaceRouting	Define for each processor the type of routing from this bus: storeAndForward, cutThrough, or None	[None,..., None]	[None,..., None]
startLatency	Start latency, $L$ , in cycles	80	0
startCost	Start cost on the channel, $S$ , in cycles	0	0
bandwidthPerCh	Bandwidth per channel, $B$ in bytes per cycle	16	16
finishCost	Finish cost, $F$ , in cycles	0	0
numChannels	Number of channels on the bus	3	1
multiplexable	False for a hardware FIFO that can only support one stream	True	True

(b) *Definition of an interconnect*

Figure 2.3: Processor and interconnect parameters of the ASM and values for two example targets (measured on Cell and estimated for a four-core SMP)

## 2. ABSTRACT STREAMING MACHINE

Parameter	Description	Cell	SMP
name	Unique name in platform namespace	'LSn'	'Mem'
size	Size, in bytes	262144	2147483648
clockRate	Clock rate, in GHz	3.2	0.4
latency	Access latency, in cycles	6	4
bandwidth	Bandwidth, in bytes per cycle	16	8

Figure 2.4: Memory parameters of the ASM and values for two example targets

The bandwidth and latency of each channel is controlled using four parameters: the start latency ( $L$ ), start cost ( $S$ ), bandwidth ( $B$ ), and finish cost ( $F$ ). In transferring a message of size  $n$  bytes, the latency of the link is given by  $L + S + \lfloor \frac{n}{B} \rfloor$  and the cost incurred on the link by  $S + \lfloor \frac{n}{B} \rfloor + F$ . This model is natural for distributed memory machines, and amounts to the assumption of cache-to-cache transfers on shared memory machines. Figure 2.5 shows the temporal behaviour of a single message transfer on a bus.

Hardware routing is controlled using the `interfaceRouting` parameter, which defines for each processor whether it can route messages from this interconnect: each entry can take the value `storeAndForward`, `cutThrough` or `None`. Memory controllers and routers are modelled as a degenerate type of processor.

Each memory is defined using the parameters shown in Figure 2.4. The latency and bandwidth figures are currently unused in the model, but may be used by the compiler to refine the estimate of the run time of each task. The memory definitions are used to determine where to place communications buffers, and provide constraints on blocking factors.

### 2.3 ASM Program Description

The ASM program model is the execution model of the compiled program running under the ASM simulator. The compiled stream program is a connected directed graph of tasks and point-to-point streams, as described later in Section 3.2.1. Kernels are present in the source program, whereas tasks, each of which implements one or more kernels, are present in the executable.

Tasks communicate using four *acolib* communications primitives, which use a push model similar to the DBI (Direct Blocking In-order) variant of TTL [vdWKH<sup>+</sup>04]. These primitives push or pop *buffers*, which contain a fixed number of elements chosen by the compiler. The buffer sizes can be different at the producer and consumer ends, but the following description assumes they are the same, to avoid extraneous detail. A *block* is the contents of one buffer, and  $i$  and  $j$  count blocks, starting at zero. The first argument,  $s$ , is the stream. Each end of the stream has a fixed number of buffers, chosen by the compiler using the algorithm in Section 3.3, and denoted  $n_p(s)$  and  $n_c(s)$ .

**ProducerAcquire( $s, k$ )** Wait for the producer buffer for block  $i + k$  to be available, meaning that the DMA transfer of block  $i + k - n_p(s)$  has completed

## 2.3 ASM Program Description

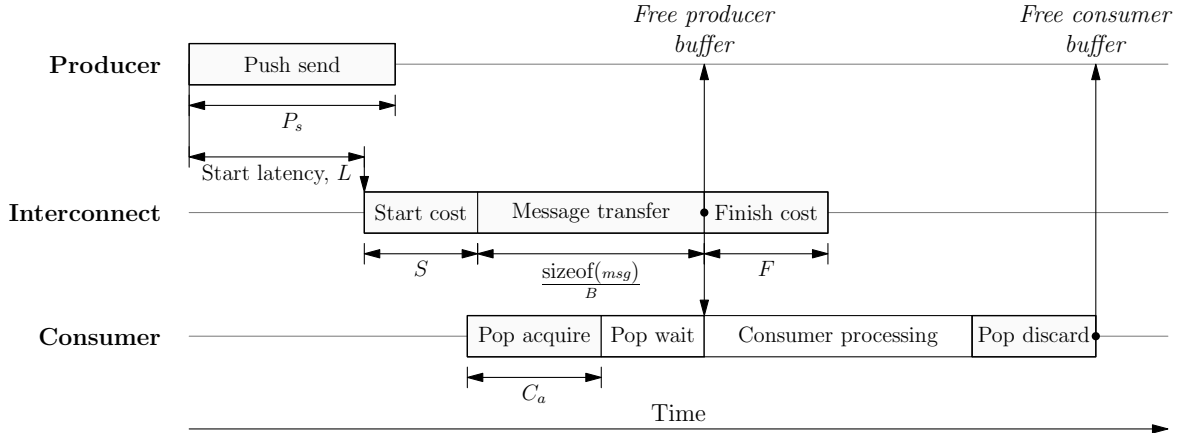


Figure 2.5: Cost and latency of communication between tasks

**ProducerSend(s)** Wait for the consumer buffer for block  $i$  to be available, meaning that the producer has received acknowledgement that block  $i - n_c(s)$  has been discarded. Then send the block and increment  $i$

**ConsumerAcquire(s, k)** Wait for block  $j + k$  to arrive in the consumer buffer

**ConsumerDiscard(s)** Discard block  $j$ , send acknowledgement, and increment  $j$

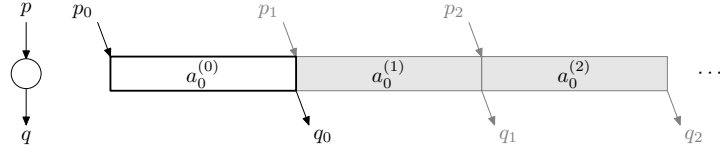
The program model uses a trace, and the same trace can be reused for several different mappings of the program onto the target—as illustrated by the small feedback loop in the bottom right of Figure 1.5. This reuse avoids recompiling the whole program via Mercurium and GCC, just to obtain a new trace. Because tasks may have complex irregular behaviour, the trace contains control flow information inside the tasks.

The basic unit of sequencing inside a task is the *subtask*, which pops a fixed number of elements from each input stream and pushes a fixed number of elements on each output stream. In detail, the work function for a subtask is divided into three consecutive phases. First, the *acquire phase* obtains the next set of full input buffers and empty output buffers, using **ProducerAcquire** and **ConsumerAcquire**. Second, the *processing phase* works locally on these buffers, and is modelled using a fixed processing time, determined from a Paraver [CEP] trace. Finally, the *release phase* discards the input buffers using **ConsumerDiscard**, and sends the output buffers using **ProducerSend**, releasing the buffers in the same order they were acquired. This three-stage model is not a deep requirement of the ASM, and was introduced as a convenience in the implementation of the simulator, since our compiler will naturally generate subtasks of this form.

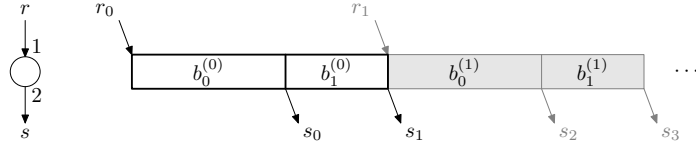
A task is the concatenation of one or more subtasks. Figure 2.6(a), (b) and (c) show how to represent some tasks that perform an arbitrary fixed sequence of communication and computation. The superscript is the iteration number of the task. Although a stream has exactly one producer and one consumer task, it may be accessed from more than one subtask. For example, Figure 2.6(b) has two subtasks,  $b_0$  and  $b_1$ , and they both push elements on stream  $s$ . In order to support control flow, all subtasks of all tasks are placed into a common *control-flow hierarchy*. Subtasks are executed conditionally or repeatedly

## 2. ABSTRACT STREAMING MACHINE

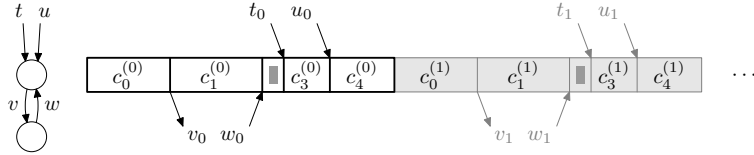
---



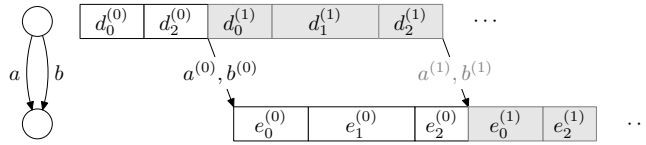
(a) Task containing a single subtask  $a_0$ ; the superscript is the iteration number



(b) Interpolation task containing subtasks  $b_0$  and  $b_1$



(c) Irregular task containing subtasks  $c_0, c_1, c_2, c_3$  and  $c_4$



(d) Execution and communication of the program of Figure 2.7 and Figure 2.8

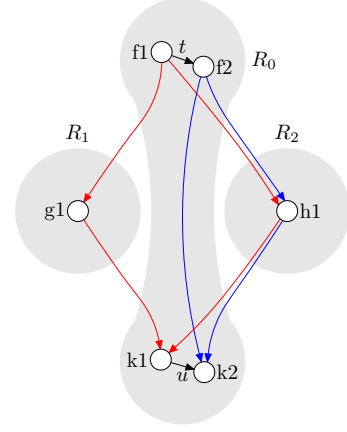
Figure 2.6: Building tasks from subtasks

```

1 #pragma acotes taskgroup
2 while (1)
3 {
4     #pragma acotes task output(a,t)
5     f1(&a, &t);
6     #pragma acotes task input(t) output(b)
7     b = f2(t);
8
9     if (cond)
10    {
11        #pragma acotes task input(a) \
12        output(a)
13        a = g1(a);
14    }
15    else
16    {
17        #pragma acotes task input(b) \
18        output(a,b)
19        h1(&a, &b);
20    }
21
22    #pragma acotes task input(a) output(u)
23    u = k1(a);
24    #pragma acotes task input(b,u)
25    k2(b, u);
26 }

```

(a) SPM source program



(b) The three connectivity sets:  
 $R_0$ ,  $R_1$ , and  $R_2$

Figure 2.7: Example stream program with data-dependent flow

based on a Paraver trace attached to this control-flow hierarchy, with this common trace ensuring that communicating tasks behave consistently.

Each *if* or *while* statement has an associated *control variable*, which gives its sequence of arguments. As part of the conversion from SPM to C, the Mercurium tool inserts calls to the trace collection functions, which record the control variables in the Paraver trace. The control variables are represented using *event records* in the trace; the *event type* identifies the control variable, and the *event value* gives its value.

Figure 2.7(a) is the source code for an example stream program containing six kernels and an *if* statement. Figure 2.7(b) is the stream graph. Because the program contains an *if* statement, the multiplicities of the kernels depend on the data. However, within each shaded region,  $R_0$ ,  $R_1$ , and  $R_2$ , the program is homogeneous Synchronous Dataflow (SDF).

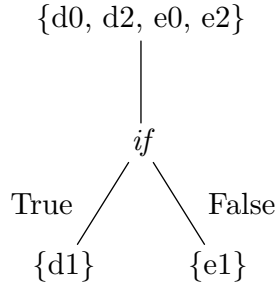
The ASM sees the program after it has been partitioned. Imagine that the partition is as given in Figure 2.8(a), so that task D contains kernels f1, f2, and g1, and task E contains kernels h1, k1, and k2. The tasks execute at the same frequency, but they both contain kernels from inside and outside the *if* statement. Conditional execution of g1 and h1, including modelling of computation times and their pushes and pops, is driven using a control variable in the trace.

Figure 2.8(c) is one way for the compiler to implement the given partition. The tasks are decomposed into subtasks, d0, d1, and d2, and e0, e1, and e2. Figure 2.8(b) shows the control flow hierarchy that controls the execution. The subtasks at the root are always executed, d1 is executed if the control variable is True, and e1 is executed if it is False. Figure 2.6(d) shows an execution trace where the decision values for this node are False, True,  $\dots$ . The

## 2. ABSTRACT STREAMING MACHINE

Task	Kernels
D	f1, f2, g1
E	h1, k1, k2

(a) Partition



(b) Control flow hierarchy

```

1 void D(void)
2 {
3   while (1)
4   {
5     f1(&a, &t);    // d0
6     b = f2(t);    // d0
7     if (cond)     // d1
8       a = g1(a);  // d1
9     push(s, a);   // d2
10    push(t, b);   // d2
11  }
12 }
13
14 void E(void)
15 {
16   while (1)
17   {
18     a = pop(s);   // e0
19     b = pop(t);   // e0
20     if (!cond)    // e1
21       h1(&a, &b); // e1
22     u = k1(a);    // e2
23     k2(b, u);     // e2
24   }
25 }
  
```

(c) Extended C for partition

Figure 2.8: Representation of data-dependent flow between tasks and subtasks

control variable attached to a *while* node is similar, but it counts the number of iterations of the loop.

There are no explicit streams carrying the control variables of *if* or *while* statements between tasks. The compiler ensures that such tasks are consistent with each other, and may in the general case have to add such streams to do so. There are, however, examples where it would be unnecessary. It is assumed that the compiler produces correct code, and the ASM uses the control-flow hierarchy to ensure that its own model is consistent.

A stream is defined by the size of each element, and the location and length of either the separate producer and consumer buffers (distributed memory) or the single shared buffer (shared memory). These buffers do not have to be of the same length. If the producer or consumer task uses the peek primitive, then the buffer length should be reduced to model the effective size of the buffer, excluding the elements of history. The Finite Impulse Response (FIR) filters in the GNU radio benchmark of Section 2.5 are described in this way. It is possible to specify a number of elements to prequeue on the stream before execution begins.

### 2.4 Platform characterisation

The platform is characterised using the small suite of synthetic benchmarks illustrated in Figure 2.9. All benchmarks have variable number of bytes transferred per iteration, denoted  $b$ . The *producer-consumer* benchmark is used to determine basic parameters, and has two actors: a producer, and consumer, with two buffers at each end. The *chain* benchmark, is a linear pipeline of  $n$  tasks, and is used to characterise bus contention. The *chain2* benchmark



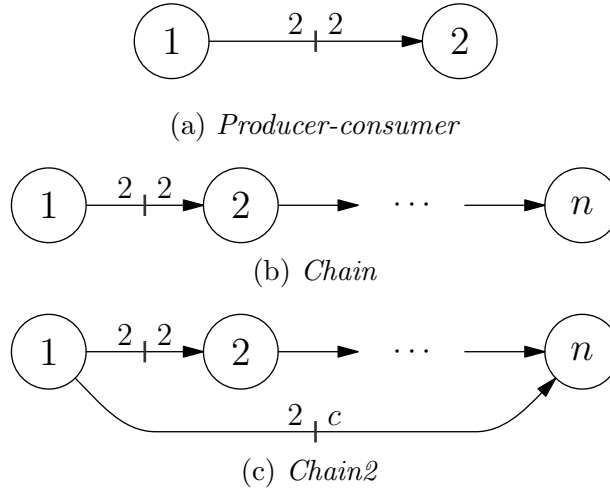


Figure 2.9: Synthetic stream benchmarks

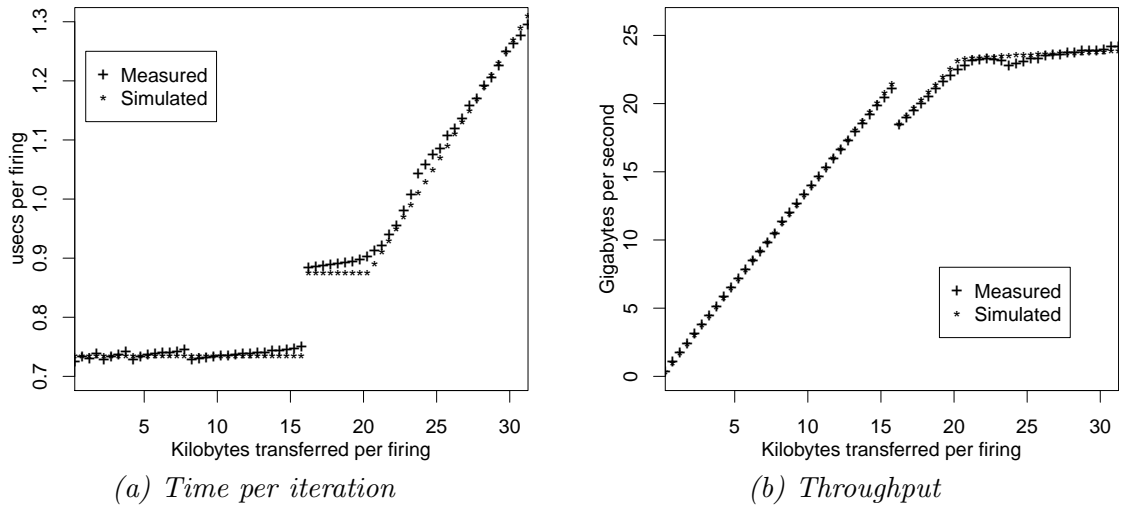


Figure 2.10: Results for producer-consumer benchmark on Cell B.E.

## 2. ABSTRACT STREAMING MACHINE

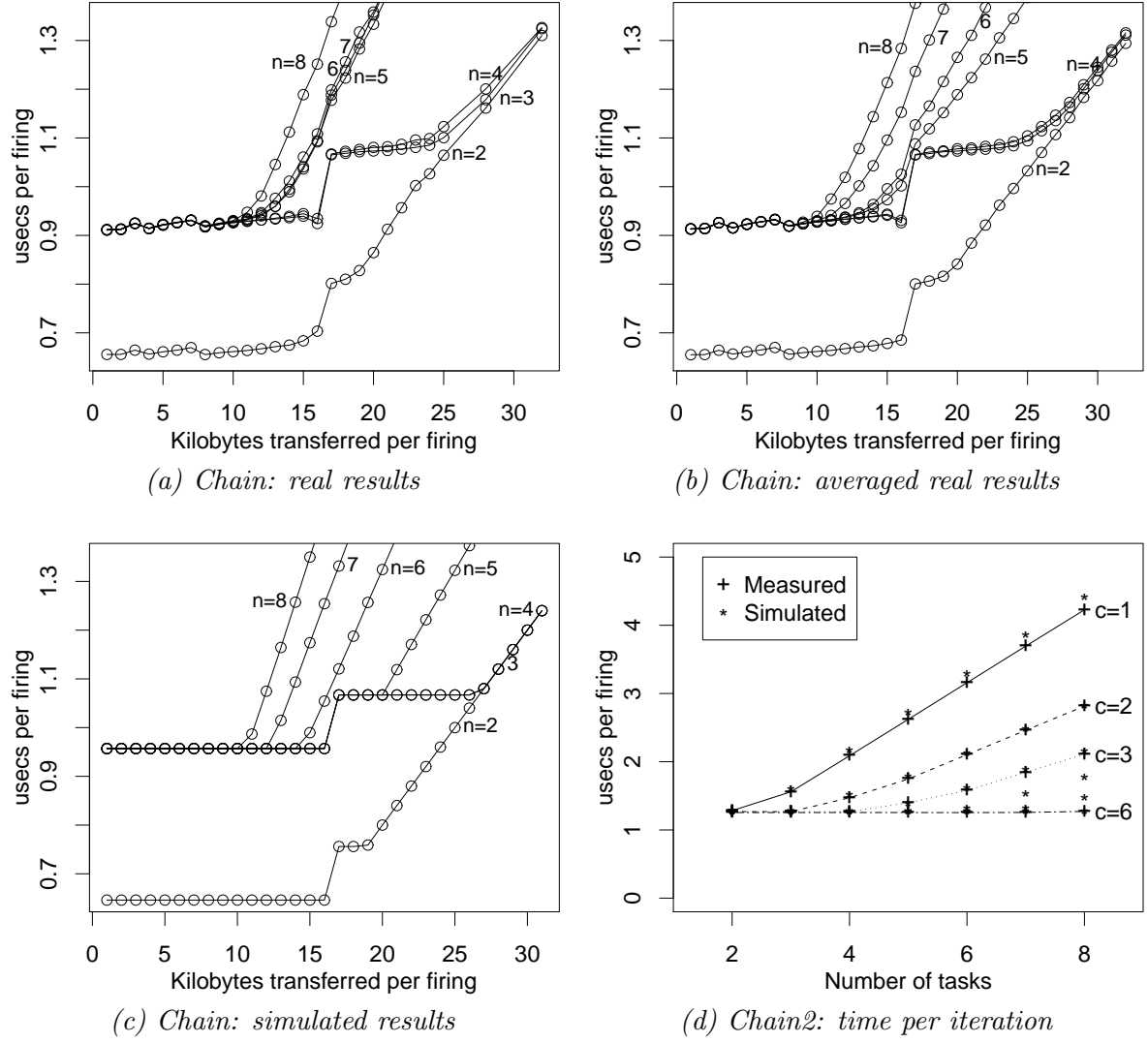


Figure 2.11: Time per iteration for the chain and chain2 benchmarks on Cell B.E.

is used to model latency and queue contention, and is a linear pipeline, similar to chain, but with an extra *cut* stream between the first and last tasks. The number of blocks in the consumer-side buffer on the cut stream is a parameter,  $c$ .

This section characterises the IBM QS20 blade, which was two Cell B.E.s (processors). Figure 2.10 shows the time per iteration for *producer-consumer*, as a function of  $b$ . The discontinuity at  $b = 16\text{KB}$  is due to the overhead of programming two DMA transfers. For  $b < 20.5\text{KB}$ , the bottleneck is the computation time of the producer task, as can be seen in Figure 2.12(a) and (b), which compares real and simulated traces for  $b = 8\text{K}$ . For  $b > 20.5\text{K}$ , the bottleneck is the interconnect, and the slope of the line is the reciprocal of the bandwidth:  $25.6\text{GB/s}$ . Figure 2.12(c) and (d) compares real and simulated traces for  $b = 24\text{K}$ . The maximum relative error for  $0 < b < 32\text{KB}$  is 3.1%.

Figure 2.11 shows the time per iteration for *chain*, as a function of  $n$ , the number of tasks, and  $b$ , the block size. Figure 2.11(a) shows the measured performance on the IBM QS20 blade, when tasks are allocated to SPEs in increasing numerical order. The EIB (Element Interconnect Bus) on the Cell processor consists of two clockwise and two anticlockwise rings, each supporting up to three simultaneous transfers provided that they do not overlap. The drop in real, measured, performance from  $n = 4$  to  $n = 5$  and from  $n = 7$  to  $n = 8$  is due to contention on certain hops of the EIB, which the ASM does not attempt to model. As described in Section 2.2, the ASM models an interconnect as a set of parallel buses. Figure 2.11(b) shows the average of the measured performance of three random permutations of the SPEs. The simulated results in Figure 2.11(c) are hence close to the expected results, in a probabilistic sense, when the physical ordering of the SPEs is not known.

Figure 2.11(d) shows the time per iteration for *chain2*, as a function of the number of tasks,  $n$ , and the size of the consumer-side buffer of the *shortcut* stream between the first and last tasks, denoted  $c$ . The bottleneck is either the computation time of the first task ( $1.27\mu\text{s}$  per iteration) or is due to the latency of the chain being exposed due to the finite length of the queue on the shortcut stream. Figure 2.12(e) and (f) shows real and simulated traces for the latter case, with  $n = 7$  and  $c = 2$ .

## 2.5 Validation of the ASM

This section describes the validation work using the ACOTES GNU radio benchmark, which is based on the FM stereo demodulator in GNU Radio [GNU]. Table 2.1(a) shows the computation time and multiplicity per kernel, the latter being the number of times it is executed per pair of  $l$  and  $r$  output elements. Four of the kernels, being FIR filters, peek backwards in the input stream, requiring history as indicated in the table. Other than this, all kernels are stateless.

Table 2.1 shows two mappings of the GNU radio benchmark onto the Cell B.E. The first allocates one task per kernel, using seven of the eight SPEs. Based on the resource utilisation, the *Carrier* kernel was split into two worker tasks and the remaining kernels were partitioned onto two other SPEs. This gives 79% utilisation of four processors, and approximately twice the throughput of the unoptimised mapping, at  $7.71\text{ms}$  per iteration, rather than  $14.73\text{ms}$  per iteration. The throughput and latency from the simulator are within 0.5% and 2% respectively.

## 2. ABSTRACT STREAMING MACHINE

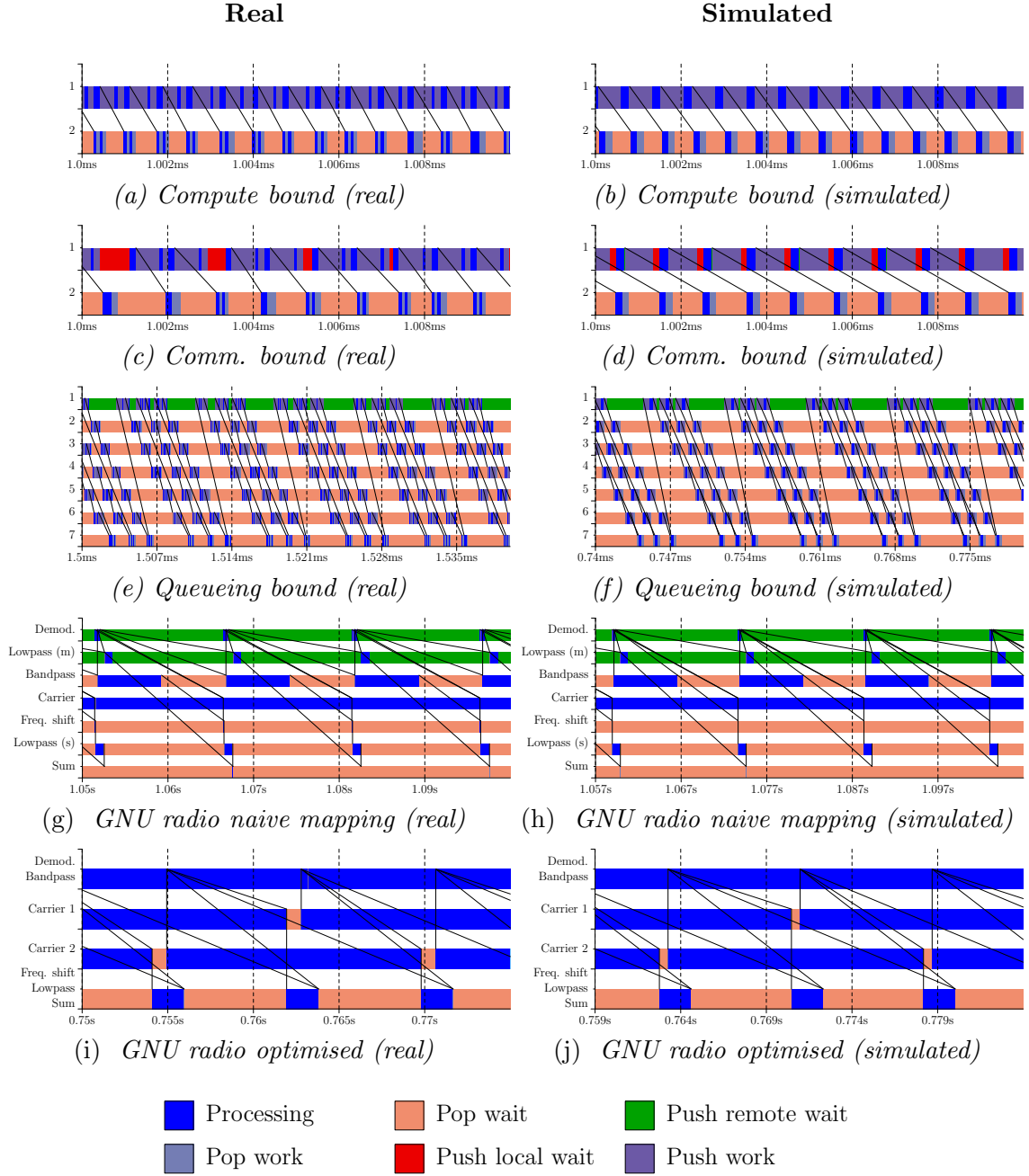


Figure 2.12: Comparison of real and simulated traces

Kernel	Multiplicity	History buffer	Time per firing (us)	% of total load
Demodulation	8	n/a	398	1.7%
Lowpass (middle)	1	1.6K	7,220	3.8%
Bandpass	8	1.6K	7,246	30.4%
Carrier	8	3.2K	14,351	60.2%
Frequency shift	8	n/a	12	0.1%
Lowpass (side)	1	1.6K	7,361	3.9%
Sum	1	n/a	13	0.0%

(a) *Kernels*

Task	Kernel	Blocking factor	Task	Kernel	Blocking factor
1	Demodulation	512	1	Demodulation	1024
2	Lowpass (middle)	128		Bandpass	1024
3	Bandpass	1024	2	Carrier (even)	1024
4	Carrier	1024	3	Carrier (odd)	1024
5	Frequency shift	1024		Lowpass (middle)	128
6	Lowpass (side)	128	4	Frequency shift	1024
7	Sum	128		Lowpass (side)	128
				Sum	128

(b) *Naive mapping*(c) *Optimised mapping*

Table 2.1: Kernels and mappings of the GNU radio benchmark

## 2. ABSTRACT STREAMING MACHINE

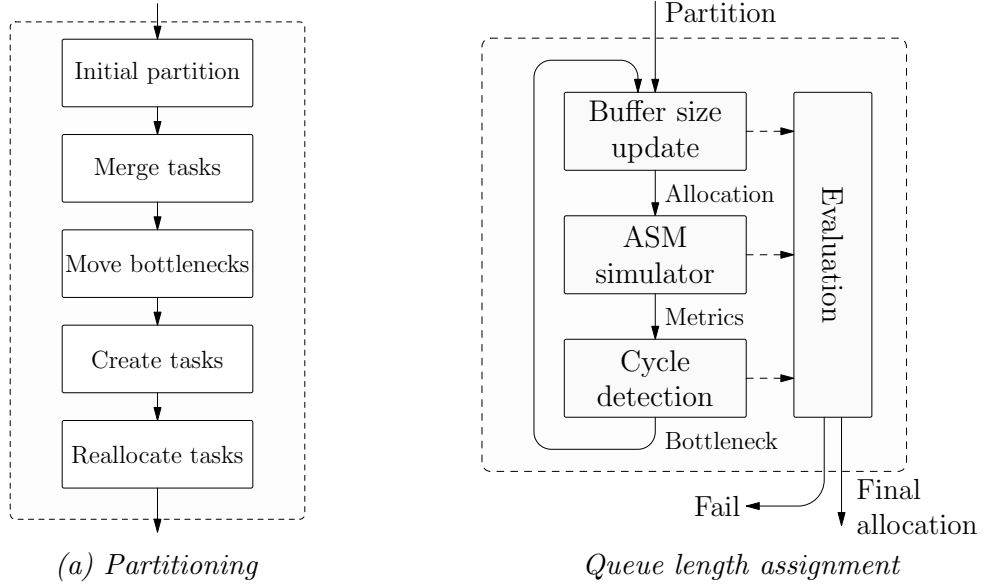


Figure 2.13: Detail on the main phases in the search algorithm

## 2.6 Using the ASM

This section explains how the ACOTES stream compiler uses the ASM machine description and simulator.

### 2.6.1 Static partitioning

The partitioning phase in Section 3.2 decides how to fuse kernels into tasks, and allocates these tasks to processors [CRA09a]. It represents the target as an undirected bipartite graph,  $H = (V_H, E_H)$ , taken directly from the ASM. The weight of processor  $p$ , denoted  $w^p$  is its clock rate in GHz, and the weight of interconnect  $u$ , denoted  $w^u$  is its bandwidth in GB/s. The static routing table is determined using minimum distance routing, respecting the `interfaceRouting` parameters. We didn't find it necessary to store the routing table explicitly in the ASM.

Figure 2.13(a) shows the main stages in the partitioning phase. An initial partition is constructed by recursively subdividing the target and program. The partition is then improved using several optimisation passes.

The partitioning phase uses *connectivity sets* [CRA09a] to constrain the mapping to make sure the compiler can support it. In particular, the ACOTES compiler can only fuse kernels that are lexicographically adjacent in the same basic block. Each connectivity set is therefore a pair of adjacent kernels in the same basic block. In a more advanced compiler, we would expect the connectivity sets to be as illustrated in Figure 2.7(b).

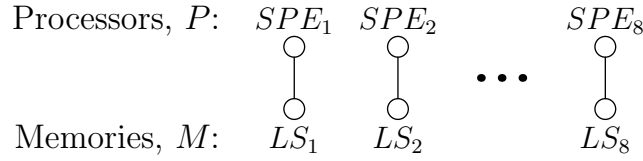


Figure 2.14: Memory constraint graph for the Cell Broadband Engine

### 2.6.2 Static buffer sizing

The queue length assignment phase in Section 3.3 allocates memory for stream buffers, subject to memory constraints, and taking account of variable computation times and task multiplicities [CRA10b]. The objectives are to maximise throughput and minimise latency.

This phase is an iterative algorithm, which uses the ASM simulator to find the throughput, utilisation, and latency, given the candidate buffer sizes. As mentioned in Section 1.3, simulation is used because a mathematical model is unlikely to capture the real behaviour.

Figure 2.13(b) shows the main stages in the queue length assignment phase. A *cycle detection algorithm* uses statistics from the ASM simulator to find the bottleneck. There are two cycle detection algorithms: the *baseline* algorithm uses only the total wait time on each primitive on each stream, and the *token* algorithm tracks dependencies through tasks. The *buffer size update algorithm* chooses the initial buffer sizes, and adjusts them to resolve the bottleneck. The *evaluation algorithm* monitors progress and decides when to stop, choosing the buffer sizes that achieved the best performance-latency tradeoff.

The inputs to the queue length assignment phase are the stream program, minimum buffer sizes, and the *memory constraint graph*. The minimum buffer sizes can be one block, because an SPM stream program is acyclic. The memory constraint graph is a bipartite graph,  $\mathcal{H} = (R_{\mathcal{H}}, E_{\mathcal{H}})$ , where the vertices are the processors and memories, and the edges connect processors to their local memories. Figure 2.14 shows a memory constraint graph for the Cell B.E.

The memory constraint graph is generated from the `addressSpace` parameter for each processor. The remaining capacities are taken from the `size` parameters of the memories, *minus* the sizes of any code and data already in them.

## 2.7 Related work

Recent work on stream programming languages, most notably StreamIt [TKA02] and Synchronous Data Flow (SDF) [LM87], has demonstrated how a compiler may potentially match the performance of hand-tuned sequential or multi-threaded code [GR05].

Most work on machine description languages for retargetable compilers has focused on describing the ISA and micro-architecture of a single processor. Among others, the languages ISP, LISA, and ADL may be used for simulation, and CODEGEN, BEG, BURG, nML [FVPF95], EXPRESSION [HGG<sup>+</sup>99], Maril and GCC's .md machine description are intended for code generation (see; e.g. [RDF98]). The ASM describes the behaviour of the system in terms of that of its parts, and is designed to co-exist with these lower-level models.

The Stream Virtual Machine (SVM) is an intermediate representation of a stream program, which forms a common language between a high-level and low-level compiler [LMT<sup>+</sup>04;

## 2. ABSTRACT STREAMING MACHINE

---

[MTHV04](#)]. Each kernel is given a linear computation cost function, comprised of a fixed overhead and a cost per stream element consumed. There is no model of irregular dataflow. The SVM architecture model is specific to graphics processors (GPUs), and characterises the platform using a few parameters such as the bandwidth between local and global memory. The PCA Machine Model [[Mat04](#)], by the Morphware Forum, is an XML definition of a reconfigurable computing device, in terms of *resources*, which may be processors, DMA engines, memories and network links. The reconfigurable behaviour of a target is described using *ingredients* and *morphs*. Unlike the ASM, the PCA Machine Model describes the entire target, including low-level information about each processor’s functional units and number of registers.

ORAS is a retargetable simulator for design-space exploration of stream-based dataflow architectures [[Kie99](#)]. The target is defined by the *architecture instance*, which defines the hardware as a graph of architecture elements, similar to the resources of the ASM. The purpose is performance analysis rather than compilation, and the system is specified to a greater level of detail than the ASM.

Gordon et al. present a compiler for the StreamIt language targeting the Raw Architecture Workstation, and applying similar transformations to those discussed in this chapter and the next [[GTA06](#)]. As the target is Raw, there is no general machine model similar to the ASM. The compiler uses simulated annealing to minimise the length, in cycles, of the critical path. Our approach has higher computational complexity in the compiler’s cost model, but provides retargetability and greater flexibility in the program model.

Gedae [[LBS](#)] is a proprietary stream-based graphical programming environment for signal processing applications in the defence industry. A version of Gedae has been released for the Cell processor. The developer specifies the mapping of the stream program onto the target, and the compiler generates the executable implementation. There is no compiler search algorithm or cost model.



## Chapter 3

# Compile-time Decisions

The stream compiler takes the human-readable source code, written in a stream programming language, and generates an executable that runs efficiently on the target machine. The compiler applies optimising transformations, including those described in this chapter.

The first high-level transformation enabled by stream programming is unrolling. Unrolling batches up work to amortise overheads in computation and communication. It also enables vectorisation [LA00] and data reuse. Automatically determining the unroll factors was not considered part of this thesis. MacroSS [HCW<sup>+</sup>10] is one technique that unrolls kernels to enable SIMD vectorisation. The unroll factors for the benchmarks in this chapter and Chapter 4 were set manually.

The next transformation is static partitioning, which decides which kernels should be fused together, and on which processors they should be executed. Clearly a good partitioning algorithm is crucial to high performance. A bad partition may be poorly balanced, loading most of the work onto one processor; or it may be well balanced but imply an excessive amount of communication back and forth between the processors. In which case, the communication links could become the bottleneck.

The final step is to statically allocate the buffers. This is an important problem, because it affects performance, as explained in Section 3.1, especially when computation times and communication rates are variable. Some platforms with distributed memory provide each core with very little addressable memory. For example, the Cell B.E. has just 256KB of local store per SPE, which must contain all code and data. On such platforms, it is important to allocate memory carefully.

### 3.1 Motivation

The stream program is built from kernels, which communicate through one-way channels known as streams. The programmer should be encouraged to write many small kernels, knowing that kernels on the same processor will be fused together, and that performance would be as good as if the programmer had done it.

The job of the partitioning algorithm is to decide which kernels should be fused together, and on which processors they should be executed. The goal is performance, taking account of constraints from the compiler. The partitioning algorithm optimises performance by trying to balance the load equally among the processors.

### 3. COMPILE-TIME DECISIONS

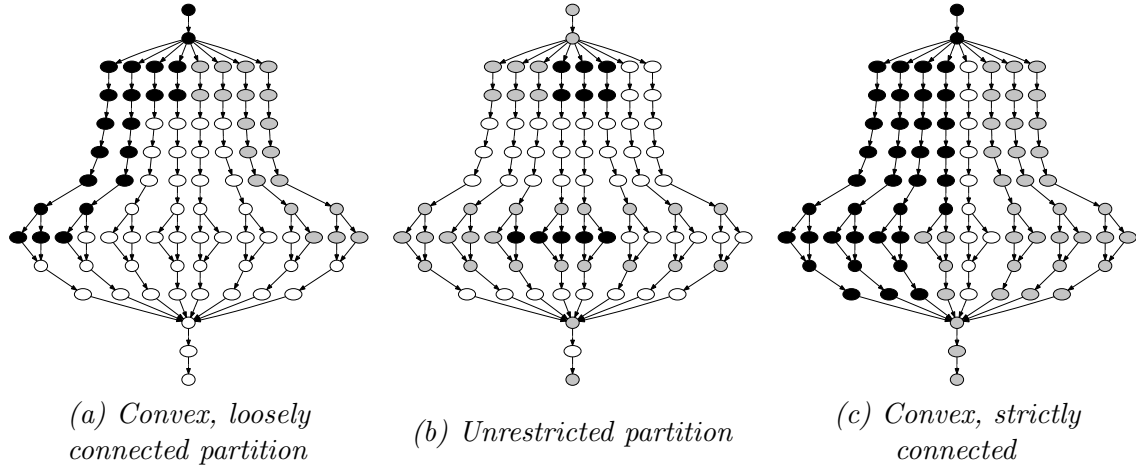


Figure 3.1: Example partitions of the StreamIt filterbank benchmark onto a 3-core SMP. Each node is a kernel, each colour is a task, and each edge is a stream

The choice of partition also affects performance indirectly through its effect on software pipelining and buffer allocation. Regarding the former, the partition may imply an excessively long software pipeline. Regarding the latter, in the worst case there may not be enough memory in local stores for the buffers implied by the partition, so the partition may not be realisable. Or there may only be space for small buffers, which are not sufficient to cover latencies and short-term variation.

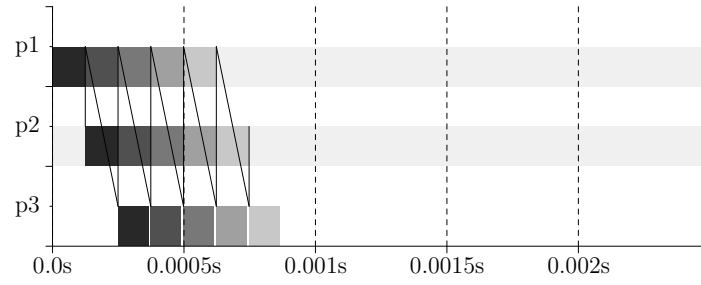
Figure 3.1 shows three partitions for the StreamIt *filterbank* benchmark on a 3-core SMP. Each processor has a single task containing the kernels of its colour. Figure 3.1(a) is the partition generated by the heuristic in Section 3.2. Data flow is from processor  $p_1$  (black) to  $p_2$  (grey) and  $p_3$  (white), and from  $p_2$  to  $p_3$ —an acyclic graph. Figure 3.2(a) shows an execution trace, with shades of grey corresponding to five iterations.

Figure 3.1(b) shows a partition that would be optimal, ignoring the cost of software pipelining. This partition requires software pipelining, since otherwise, as shown in the trace in Figure 3.2(b), there are many stalls where dependencies prevent computation from being overlapped; throughput is 53% lower than before. Figure 3.2(c) is pipelined using the stage assignment phase from the SGMS algorithm [KM08]. It has 0.2% higher throughput than the convex partition, but due to startup overhead would break even only after 8,000 iterations.

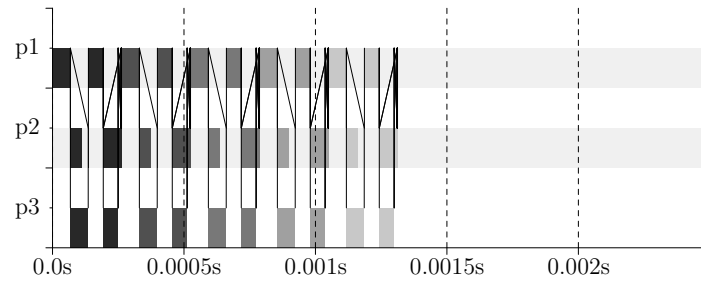
The partitioning problem, even excluding its indirect effects and communications costs, is  $\mathcal{NP}$ -hard, so it can only be solved using heuristics. Section 3.2 describes a new heuristic for the partitioning problem. It considers the loads on the processors and buses, considers its effect on downstream passes, and models the compiler’s ability to fuse kernels.

#### 3.1.1 Convexity

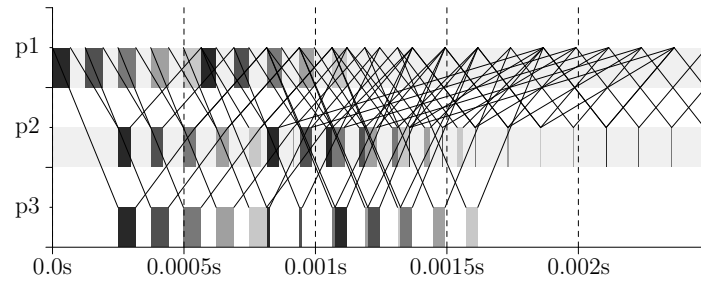
A partition is convex if the graph of dependencies between tasks is acyclic. Equivalently, every directed path between two kernels in the same task is internal to that task. The convexity constraint is intended to avoid long software pipelines. As illustrated in the previous section, a partitioning algorithm unaware of the cost of pipelining may require long pipelines for a small increase in throughput. The optimal unrestricted partition for the StreamIt 2.1.1



(a) *Convex partition from heuristic (time: 1.00)*



(b) *Unrestricted without pipelining (time: 2.11)*



(c) *Unrestricted partition with pipelining (time: 1.00)*

Figure 3.2: Traces for five iterations of filterbank, scheduled using SGMS. Iterations are identified using shades of grey.

### 3. COMPILE-TIME DECISIONS

---

*serpent* benchmark [GTA06] on two Cell B.E.s is 10% faster than the optimal convex partition, but it requires 209 pipeline stages rather than 31. We did not obtain the CPLEX Solver to evaluate StreamRoller, but since it uses ILP to solve a similar problem, its result should be similar. This translates into higher memory use, which may simply not fit, as well as startup overhead and latency. Table 3.1 shows that partitions from our algorithm seldom require pipelining at all, and performance is, on average, within 5% of optimum.

When the benefit from software pipelining is above some threshold, the algorithm relaxes connectedness and convexity. Section 3.2.3 shows the partition of *vocoder*, which benefits from software pipelining. The result is close to optimal performance using a short pipeline.

#### 3.1.2 Connectivity

The connectedness constraint is primarily to help code generation, since it is easier to fuse adjacent kernels, whose relative frequencies are known via the stream between them. Figure 3.3(a) shows a program using the SPM (Section 1.5.2). Kernels *read* and *write* perform IO, and *update* manages the automaton and sends only accepting states. The macros `NEW_STATE` and `ACCEPT_STATE` manage the automaton, and their precise behaviour is irrelevant to the discussion. Consider the case where the partition merges *read* and *write* into task 1, with *update* in task 2. This partition is not convex, so pipelining is required. Task 1 is not connected, so the compiler requires the relative frequencies of *read* and *write*.

This can be solved using dynamic scheduling inside the task, by switching between kernels when a push or a pop starts to wait. Dynamic scheduling may not be supported by the runtime, and it adds overhead and unpredictability, which are undesirable in real-time embedded systems. Chapter 4 addresses dynamic scheduling of stream programs. In the absence of a runtime dynamic scheduler, this example requires either an extra stream carrying the condition, as in Figure 3.3(b), or duplicating the calculation of the condition, plus the state on which it is based, which would duplicate the whole *update* kernel.

The general case requires duplicating state or creating a dependence cycle. Figure 3.4 shows an example, not using the SPM, where each push and pop is guarded by a condition; e.g. each time  $k_1$  fires, it pushes on the stream to  $k_3$  whenever  $a$  is true and pushes on the stream to  $k_4$  whenever  $b$  is true. The relationship between the firing rates of any two kernels depends on all conditions on the path between them. If  $k_2$  and  $k_3$  are fused into one task, then the entire graph must be fused. This is because the task containing  $k_2$  and  $k_3$  must know the relationship between their firing rates. It therefore requires some function of  $e$  and  $f$  to be sent from  $k_4$ , and this creates a directed cycle.

A naïve definition of connectivity, *strict connectivity*, considers a partition to be connected when each processor has a weakly connected subgraph. Unfortunately, wide split-joins, as in *filterbank*, do not usually have good partitions subject to this constraint. In Figure 3.1(a),  $p_2$  (grey) is not strictly connected, so our strict heuristic produces the partition in sub-figure (c), which has performance 28% worse than (a). In general, strict connectedness allows only the processors containing the split or the join kernel to have kernels from more than one branch.

We generalise connectivity by providing to the partitioning algorithm a set of *basic connected sets* [MB06], each of which specifies kernels that the compiler can pairwise merge. For strict connectivity, there is a basic connected set for each pair of communicating kernels.

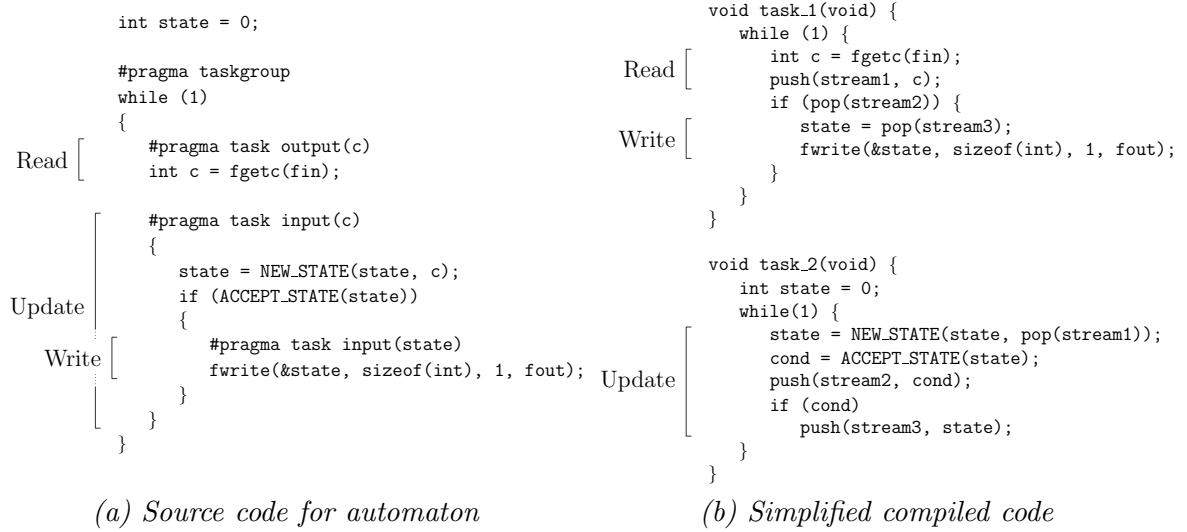


Figure 3.3: Motivation of connectivity: example programs with data dependent pushes and pops

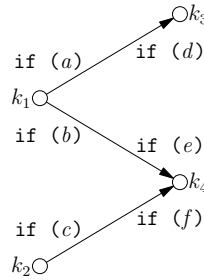


Figure 3.4: If  $k_2$  and  $k_3$  are fused into one task, then the entire graph must be fused

This allows the partitioning algorithm to be adapted to the compiler and source language(s). If the compiler understands StreamIt [TKA02] splitters and joiners, there should be a basic connected set for each splitter (joiner) containing its successors (predecessors), which solves the problem outlined above. Similarly, there may also be a basic connected set covering each region of the program graph that is internally SDF.

### 3.1.3 Queue sizes

The second problem considered in this chapter is static queue sizing. Double buffering is a well-known technique to overlap communication and computation. There are two situations, however, when a stream ought to be allocated more than two buffers. The first is when a stream covers a long latency or, equivalently, crosses more than one pipeline stage boundary. The second is when there are short-duration load imbalances due to variable computation times or communication rates.

The *chain8* benchmark illustrates the first situation, and is shown in the upper part of Figure 3.5. It has eight tasks in a pipeline, with streams between consecutive tasks, and

### 3. COMPILE-TIME DECISIONS

---

another stream between the first and last tasks. Figure 3.5(a) shows the progress of the first and last tasks relative to the stream between them. The vertical axis is time, and the horizontal axis is the position in the stream. At any given time the producer is working on some interval of the stream, which it owns. It starts at the top left of the plot, at the beginning of both the stream and time, moving to the right when it sends data to the consumer, and continually downward through time. The figure also shows the progress of the consumer. The progress of the consumer is also shown.

The periodic pattern of waiting is caused by the interaction between two dependencies. First, the consumer must wait for its data to arrive, which means that it waits for the producer, plus the latency of the pipeline. This gives a vertical dependency from producer to consumer. Second, the producer must wait for an empty consumer-side buffer in which to send its data, and this gives a horizontal dependency from consumer to producer. The interaction between these dependencies causes the periodic pattern of waiting.

Figure 3.5(b) is for six consumer-side buffers, which increases throughput by 73%, and is sufficient for the producer to be always busy. This shows that double buffering was not sufficient, but also that the number of buffers can be less than one plus the difference in pipeline stage, which is the number of buffers allocated by StreamRoller [KM08] and SPIR [CLC<sup>+</sup>09]; in this case eight.

The second situation is illustrated using the *producer-consumer* example in the lower part of Figure 3.5. If the producer and consumer both have fixed computation times and communication rates, then double buffering is sufficient. Sometimes, single buffering at one or other end will be enough, even with good load balancing. Figure 3.5(c) shows the progress of this example, using double buffering, when computation times are normally distributed. Increasing the number of consumer buffers to five, as shown in Figure 3.5(d), increases throughput by 20%.

The queue sizing algorithm is based on i) the stream program that has been mapped onto processors, ii) feedback from an earlier execution, and iii) the memory constraints. The algorithm exposes a trade-off between throughput and latency. It is general, in that it applies to stream programs with unstructured stream graphs, and it supports variable execution times and communication rates.

The inputs to the algorithm are the *mapped stream program*, a *program trace* and the *machine description*, giving the target topology and memory budgets. A simple model of computation times and communication rates, such as independent normal distributions and Poisson arrivals, may be misleading, so the only options are simulation and real execution. The experimental results use coarse-grain simulation, but real execution could be used instead. The output is the buffer size for the producer and consumer on each stream, which may be different.

The performance of the queue length assignment algorithm is quantified using the *utilisation*, which is the percentage of time that the most heavily loaded processor or bus is busy. Utilisation is proportional to throughput. If the stream graph is acyclic, at least one resource ought to be 100% busy. If any resource has utilisation less than 100%, it must be due to insufficient buffering.

The tradeoff between utilisation and the number of consumer buffers is illustrated in Figure 3.6. Chain has linearly increasing utilisation until it reaches 100%. Producer-consumer

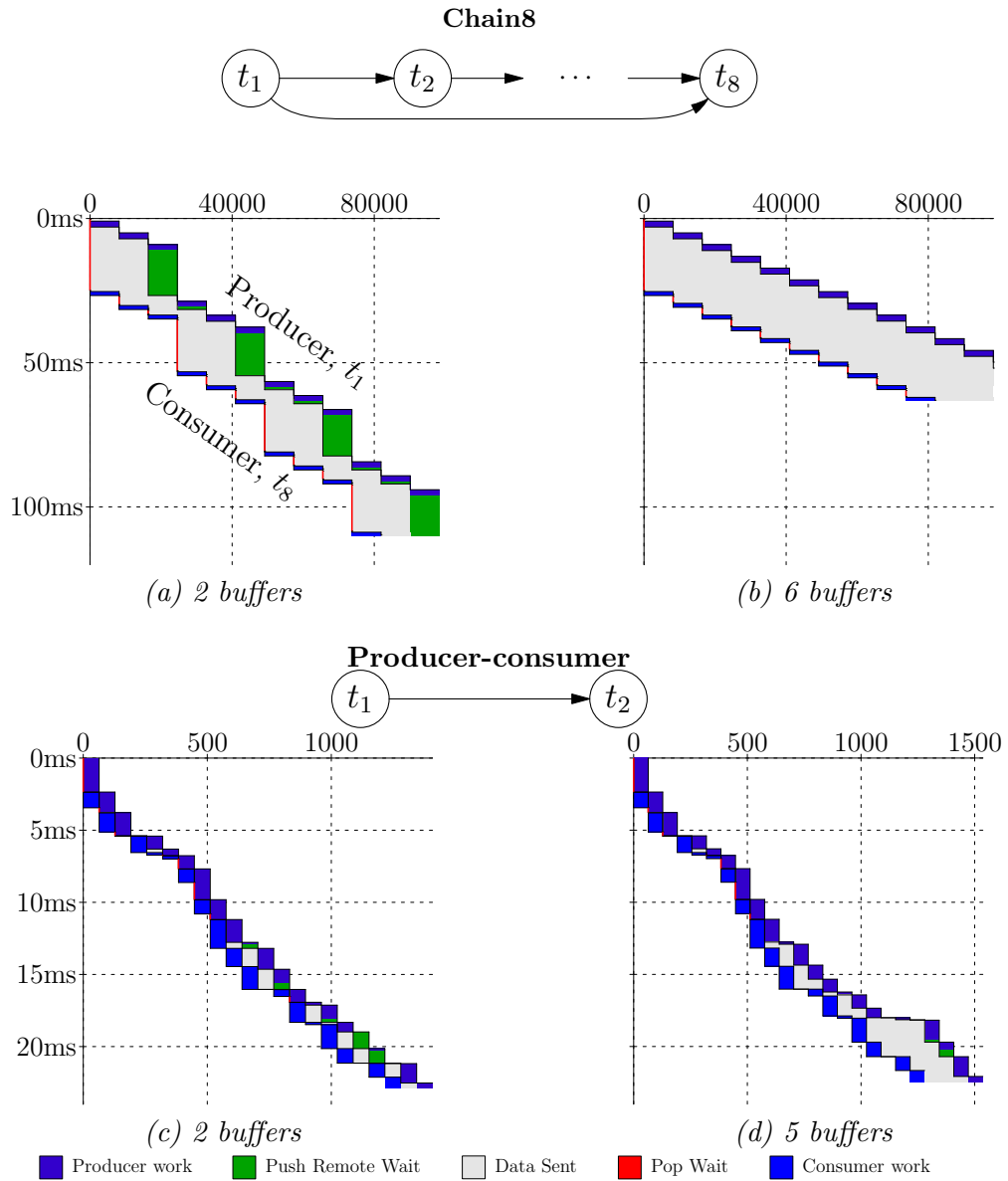


Figure 3.5: Effect of consumer queue length on chain8 and producer-consumer

### 3. COMPILE-TIME DECISIONS

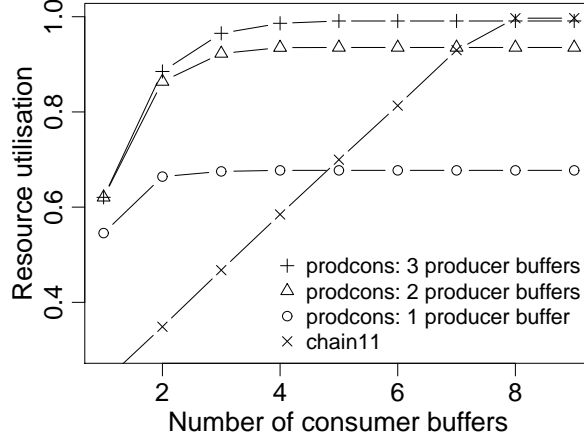


Figure 3.6: Memory-performance tradeoff

achieves 99% utilisation with 3 producer and 4 consumer buffers, and additional buffering yields diminishing returns.

The SPM and StreamIt languages eliminate deadlock, so the objective function depends only on performance and latency. The interaction between bounded memory in process networks and deadlock, but not performance, has been explored in depth [Par95; Buc93; GB03], and these techniques can determine the *minimum* buffer sizes.

The queue length assignment algorithm is iterative, and consists of a coarse-grain simulator, a *cycle detection algorithm*, a *buffer size update algorithm*, and an *evaluation algorithm*. The cycle detection algorithm analyses metrics from the simulator, and finds a bottleneck cycle. The buffer update algorithm chooses the initial buffer allocation, and adjusts buffer sizes to resolve the bottleneck. The evaluation algorithm monitors progress and decides when to stop, choosing the buffer allocation that achieved the best performance-latency tradeoff.

## 3.2 Static partitioning

### 3.2.1 The partitioning problem

The target is represented as an undirected bipartite graph  $H = (V, E)$ , where  $V = P \cup I$  is the set of vertices, a disjoint union of processors,  $P$ , and interconnects,  $I$ ; and  $E$  is the set of edges. Each processor,  $p$ , has weight,  $w^p$ , equal to its clock speed in GHz, and each interconnect,  $u$ , has weight,  $w^u$ , equal to its bandwidth in GB/s. The static route between processors  $p$  and  $q$  is represented by  $r_{pq}^u = 1$  if it uses interconnect  $u$ , and 0 otherwise. In general,  $r_{pq}^u \neq r_{qp}^u$ ; e.g. dimension-order routing on a mesh. Figure 3.8 shows the topology of our example targets, omitting the edge and vertex weights and the routing table. This representation is a simplified form of the Abstract Streaming Machine (ASM).

The program is represented as a directed acyclic graph,  $G = (K, S)$ , where  $K$  is the set of kernels, and  $S$  is the set of streams. If the program is cyclic, then each strongly connected component is contracted into a single vertex. The load of kernel  $i$  on processor  $p$ , denoted  $c^{ip}$ , is the mean number of gigacycles in some fixed time period  $\tau$ . Similarly, the load of stream  $ij$ , denoted  $c^{ij}$  is the mean number of gigabytes transferred in time  $\tau$ .



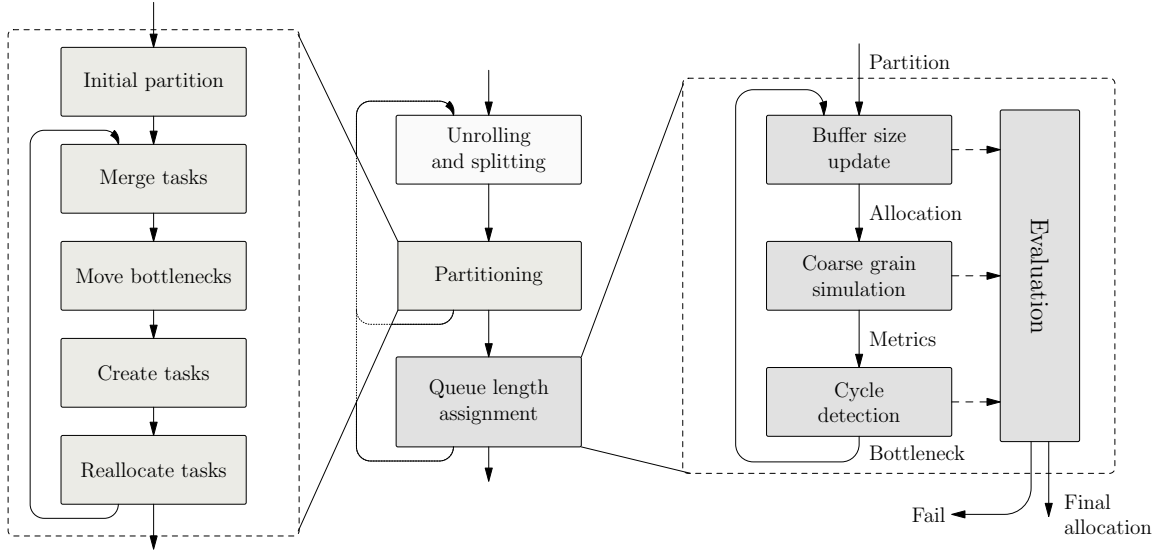


Figure 3.7: The mapping phase of the ACOTES compiler, showing the partitioning and queue sizing algorithms

The basic connected sets are a collection,  $\mathcal{C} = \{C_j\}$ , of subsets of  $K$ , where each  $C_j$  is a set of pairwise connected kernels. A subset  $L \subseteq K$  is connected if, for any pair of kernels  $k, k' \in L$ , there is a sequence  $k = k_1, k_2, \dots, k_n = k'$ , with each  $k_i \in L$  and each pair of consecutive kernels,  $k_i$  and  $k_{i+1}$ , connected by being members of some  $C_j$ . The whole set of kernels,  $K$ , should be connected.

The output of the algorithm is two map functions. Firstly,  $T$  maps kernels onto tasks, and secondly,  $P$  maps tasks onto processors. The partition implied by  $T$  must be convex, so the graph of dependencies between tasks is acyclic.

Let  $T_p = P^{-1}(p)$  be the tasks on processor  $p$ , and  $K_t = T^{-1}(t)$ ,  $K_p = \bigcup_{t \in T_p} K_t$  be the kernels on task  $t$  or processor  $p$ . The graph of  $t$  is the induced graph,  $G_t = G(K_t)$ , containing the kernels in  $t$  and internal streams. The task dependence graph  $G^T$  is the result of contracting each task in  $G$  into a single vertex.

The cost on processor  $p$  or interconnect  $u$  is

$$C^p = \sum_{i \in K_p} \frac{c^{ip}}{w^p}$$

$$C^u = \sum_{p, q \in P} r_{pq}^u \sum_{i \in K_p, j \in K_q} \frac{c^{ij}}{w^u}.$$

The goal is to find the allocation  $(T, P)$ , which minimises the maximum values of all the  $C^p$  and  $C^u$ , subject to the convexity and connectedness constraints.

### Predicting memory use of tasks

When multiple kernels are fused into one task, the algorithm needs to predict the memory use of the task, given the memory use and composition of each kernel. Finding the minimum

### 3. COMPILE-TIME DECISIONS

---

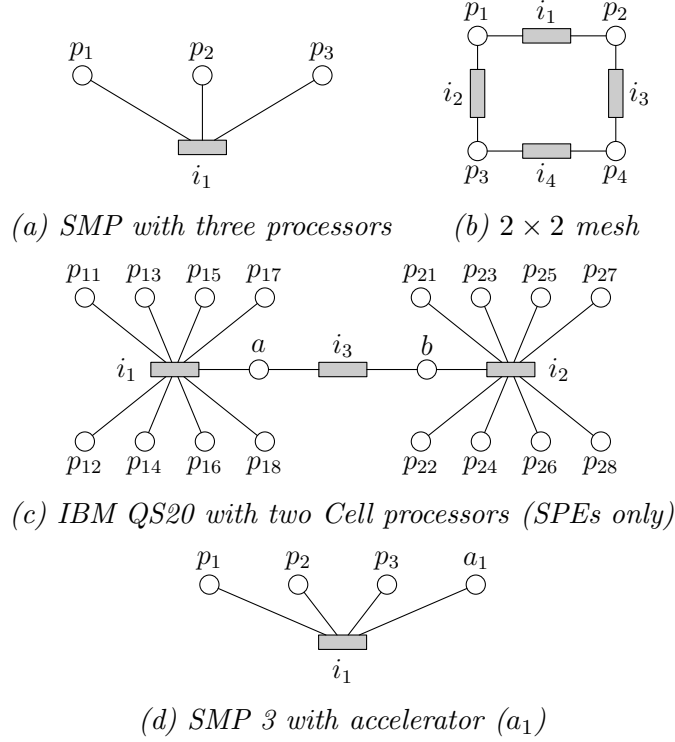


Figure 3.8: Topology of the targets used in this section (interconnects are shown as shaded rectangles)

memory use is an  $\mathcal{NP}$ -complete problem [BML96], even ignoring the possibility to overlap the buffers for two or more streams. Not only that, but the partitioning algorithm needs to predict, or at least bound, the memory use from the actual compiler, which is unlikely to be the theoretical minimum.

This algorithm assumes that the combined code size is the sum of the code sizes for the kernels, and that the total memory size is one block, plus the length of the history, for each internal stream, and the original memory size for each external stream. This calculation is orthogonal to the rest of the algorithm.

#### 3.2.2 The partitioning algorithm

The partitioning algorithm is split into two phases. The first phase produces an initial partition that is both convex and connected, with at most one task per processor. The second phase, refinement, improves the initial partition, and has some ability to escape from local minima; it can also create multiple tasks per processor.

The first phase could produce a trivial initial partition, which has all kernels in a single task, assuming enough memory. Our results show that the refinement phase still finds a good partition. A good initial partition, however, decreases the total time of the mapping algorithm, since it requires fewer passes of the refinement phase.

The refinement phase uses several algorithms based on Kernighan and Lin’s graph partitioning algorithm [KL70], and is repeated until there is no further improvement. The main

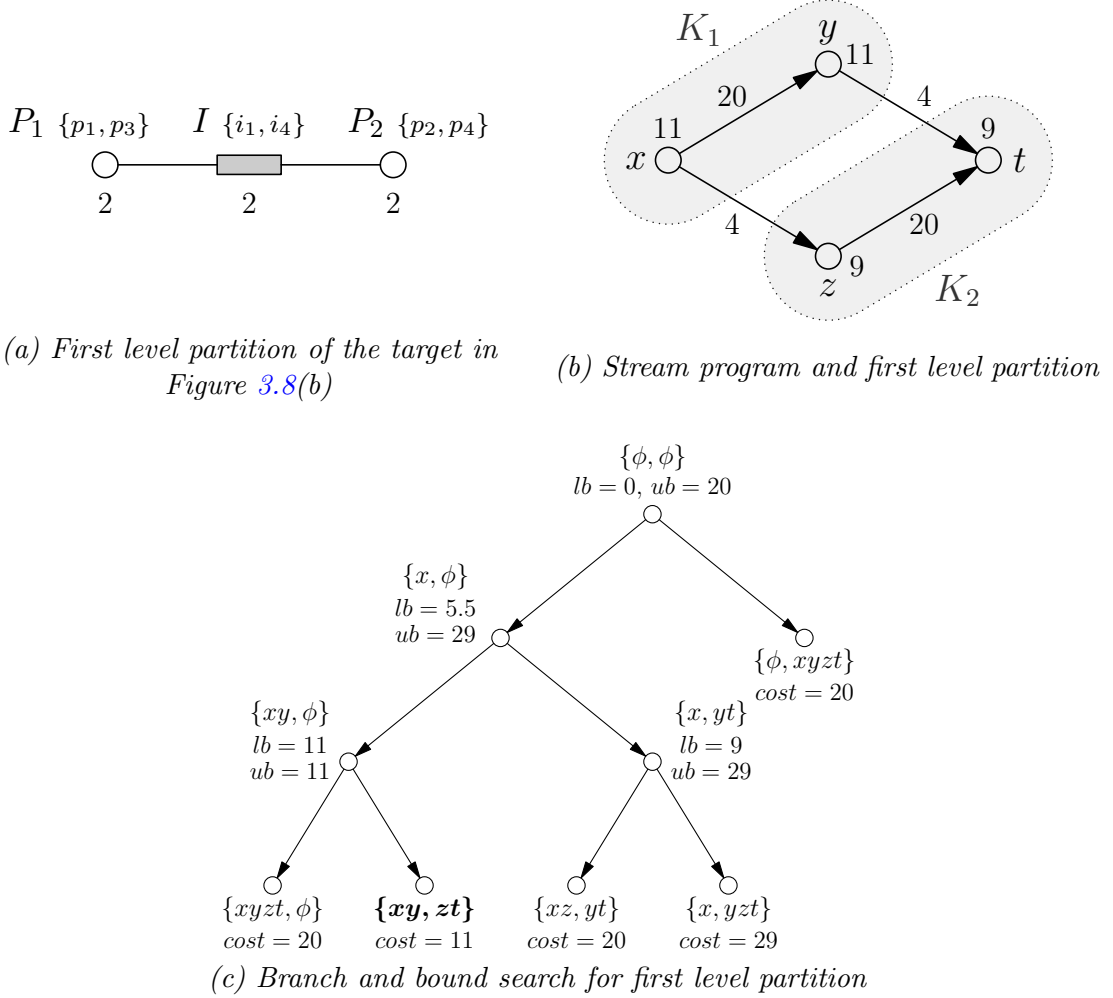


Figure 3.9: First level partition in the initial partition algorithm

step offloads kernels from bottleneck processors, while maintaining connectedness and convexity. If this produces no benefit, then one additional task is created, if enabled, and the new partition is kept if the improvement is larger than some threshold (currently 5%).

### Initial partition

The initial partition is generated by recursively subdividing the target and program graphs into halves, mapping each half separately. This continues until there is either a single kernel, which is mapped to some processor, or a single processor, which executes all kernels.

**Partitioning the target** The algorithm first divides the target into two subgraphs,  $P_1$  and  $P_2$ , and an aggregate interconnect,  $I$ , balancing two objectives: the subgraphs should have roughly equal total CPU performance, and the aggregate interconnect bandwidth between them should be low. Figure 3.9(a) shows the result of dividing the mesh target from Figure 3.8(b).

### 3. COMPILE-TIME DECISIONS

---

The optimal target partition is found as follows. First, the communications bottleneck for uniformly random traffic between  $P_1$  and  $P_2$  is given by

$$C = \max_{u \in I} \sum_{p \in P_1, q \in P_2} \frac{r_{pq}^u + r_{qp}^u}{w^u}. \quad (3.1)$$

The target is divided into halves to maximise  $\alpha$ , the product of  $C$  with the total performance of the less powerful of  $P_1$  or  $P_2$

$$\alpha = C \min \left( \sum_{p \in P_1} w^p, \sum_{q \in P_2} w^q \right). \quad (3.2)$$

An approximate solution is found using an adaptation of the Kernighan and Lin partitioning algorithm.

**Partitioning the program** The program (sub)graph is given edge and vertex weights. The edge weight for stream  $ij$ , denoted  $c_{ij}$  is the cost in cycles in time  $\tau$ , if assigned to the aggregate interconnect, rather than internal to  $P_1$  or  $P_2$ . The vertex weight for kernel  $i$  is a pair  $(c^{iP_1}, c^{iP_2})$ , the cost of assigning it to  $P_1$  or  $P_2$ , respectively. The goal is to find a two-way partition  $\{K_1, K_2\}$  to minimise the bottleneck given by

$$c = \max \left( \sum_{i \in K_1} c^{iP_1}, \sum_{j \in K_2} c^{jP_2}, \sum_{i \in K_1, j \in K_2} c_{ij} \right). \quad (3.3)$$

The partitioning algorithm is a branch and bound search. Each node in the search tree inherits a partial partition  $(K_1, K_2)$ , and unassigned vertices  $X$ ; at the root  $K_1 = K_2 = \phi$  and  $X = K$ . It chooses some kernel  $v \in X$ , adjacent to  $K_1$  with  $K_1 \cup \{v\}$  convex and connected (or any  $v$  if  $K_1$  is empty) then switches on either adding  $v$  and its ancestors to  $K_1$ , or  $v$  and its descendants to  $K_2$ . If adding vertices to  $K_1$  would cause  $K_2 \cup X$  to become disconnected, then the subtree contains no connected partitions, so is pruned.

Figure 3.9(b) and (c) show a program and its branch and bound search, with each node labelled by its sets  $K_1$  and  $K_2$ . The minimal cost,  $c^{K_1 K_2}$ , for all partitions in the subtree rooted at node  $(K_1, K_2)$  is at least as large as the partial sum on the vertices already assigned:

$$c^{K_1 K_2} \geq lb = \max \left( \sum_{i \in K_1} c^{iA}, \sum_{i \in K_2} c^{iB}, \sum_{i \in K_1, j \in K_2} c_{ij} \right). \quad (3.4)$$

Any valid partition in the subtree gives an upper bound on the optimal cost in that subtree. Since  $(K_1, X \cup K_2)$  is always valid:

$$c^{K_1 K_2} \leq ub = \max \left( \sum_{i \in K_1} c^{iA}, \sum_{i \notin K_1} c^{iB}, \sum_{i \in K_1, j \notin K_1} c_{ij} \right). \quad (3.5)$$

In Figure 3.9(c), the node marked  $\{x, \phi\}$  has  $K_1 = \{x\}$  and  $K_2 = \phi$ . The known cost on  $P_1$  is 5.5, being the cost of kernel  $x$  divided by the performance of  $P_1$ . The known costs on  $P_2$  and the interconnect are both zero. Hence by Equation (3.4),  $lb = \max(5.5, 0, 0) = 5.5$ . Similarly, by Equation (3.5), the upper bound on the optimum is the cost of partition  $\{x, yzt\}$ , so  $ub = 29$ .

The search algorithm tries to quickly find a good partition, so that more of the search tree is pruned by having its lower bound greater than some upper bound. It uses a depth-first search, and chooses vertex  $v$  adjacent to  $K_1$  (as it must) with the highest cost on whichever processor currently has the greatest load, then first considers adding it to the other processor.

### Refinement of the partition

The refinement stage starts with a valid initial partition, and improves it by applying the optimisation passes described below. As shown in Figure 3.7, these steps are applied in sequence, and iterated until no further improvement is seen. The optimisation passes are:

**Merge tasks** A greedy algorithm merges low cost tasks and has the effect of freeing processors and reducing communications

**Move bottlenecks** The main optimisation pass moves kernels from bottleneck processors

**Create tasks** Create a new task to relax the connectedness and convexity constraints, and keep the new partition if the benefit is larger than some threshold

**Reallocate tasks** A greedy algorithm improves the allocation of tasks to processors

The passes are described in detail below.

**Merge tasks** This step uses a greedy algorithm to merge tasks whose union is convex and connected, as long as it does not cause a new bottleneck. This pass often reduces bus traffic, and frees up processors so they can accept kernels without restriction. Since there are usually far fewer tasks than kernels, define the basic connected sets of tasks:  $D_j = \{T(k) : k \in C_j\}$ , where  $T(k)$  was defined earlier as the task containing kernel  $k$ , and set  $\mathcal{D} = \{D_j : |D_j| \geq 2\}$ . In this case, the union of  $T_1$  and  $T_2$  is connected if  $\{T_1, T_2\} \subseteq D_j$ , some  $D_j \in \mathcal{D}$ .

Section 3.2.1 defined the task dependence graph,  $G^T$ , as the directed acyclic graph on the tasks. Define  $d(T_1, T_2) = 1$  if there is a path from  $T_1$  to  $T_2$  of length two or more, and 0 otherwise. This can be calculated in time  $O(|T|^2)$ , using a topological sort. The greedy algorithm finds, using a branch and bound search, the connected pair of tasks  $T_1$  and  $T_2$  with minimum total cost on either of their current processors, such that  $d(T_1, T_2) = 0$ . If the bottleneck cost after merging is no greater than the current bottleneck cost, then the tasks are merged and allocated to the processor on which they have the minimum total cost. The algorithm continues until no more tasks can be merged.

**Move bottlenecks** This pass identifies a bottleneck processor,  $p_1$ , then considers moving a set  $M_1$  of kernels from some task on  $p_1$  to a task on another processor,  $q_1$ , without violating convexity or connectedness. The cost metric to minimise is the maximum of the costs on  $p_1$ ,  $q_1$  and all interconnects, after the move:

$$C = \max(C^{p_1}, C^{q_1}, \max_{u \in I} C^u). \quad (3.6)$$

This metric excludes the other processors, otherwise if some other processor had the same cost as  $p_1$ , its contribution would hide the benefit of any move.

Some kernels must be moved, even if doing so has a negative benefit—hence the algorithm has some ability to escape from local minima. After tentatively moving set  $M_1$ , record the bottleneck cost and identify the new bottleneck processor,  $p_2$ , which may still be  $p_1$ , and tentatively move a set  $M_2$  to another processor. It continues moving kernels, with the constraint that no kernel can be moved back to a processor that it has previously been

### 3. COMPILE-TIME DECISIONS

---

allocated to. For instance, none of the kernels in  $M_1$  may be tentatively moved back to  $p_1$ , but they may be moved a second time to another processor. This process continues until either there are no remaining valid moves, or a fixed limit, currently 50 moves, is reached. The final partition is that of the intermediate point in the algorithm with the maximum overall performance.

Any kernel,  $k$ , can potentially be moved to any task on a different processor if there is a kernel  $k'$  on the new task that shares a basic connected set with  $k$ . There are three additional requirements. Firstly, if  $k$  is neither a source nor a sink in its current task,  $T$ , then  $T - \{k\}$  cannot be convex. It is always necessary to move either  $k$  and all its ancestors in  $T$ , or  $k$  and all its descendants in  $T$ . Secondly, it is necessary to check, using a breadth-first or depth-first search on the basic connectivity sets, whether the remainder of the old task is still connected. Thirdly, there are several ways that the move can create a cycle in the task dependency graph, and this can be checked using a topological sort.

**Create tasks** This pass moves a kernel  $k$  on a bottleneck processor onto another processor, creating new tasks as necessary to become convex and connected. It then runs the Move Bottlenecks pass, with the restriction that the kernel cannot be moved from its new processor. The new partition is kept if the performance is improved by more than some threshold, currently 5%.

The most expensive kernel, on its current bottleneck processor, is considered first. This kernel may be moved to any processor in use for which the cost of the kernel is less than the current bottleneck cost. There is no advantage in moving a kernel to an unused processor, since that is the first thing that the Move Bottlenecks pass would do. If there are no valid choices, then the second most expensive kernel on the same processor is considered, and so on.

Kernel  $k$  is placed on some other processor in order to minimise the sum of the weights of the *large kernels*, including  $k$ , on the new processor; the large kernels are those of weight at least half that of the kernel being moved. The reason for ignoring lightweight kernels is that these are most likely to be able to be easily offloaded onto other processors.

**Reallocate tasks** The reallocation pass decreases communications traffic by permuting the loads on the processors. This pass is executed even if the bottleneck is on one of the processors. It only moves tasks between similar processors attached to different buses. Similar processors are those for which all kernel computation times are identical. For instance, it can permute the four processors on the  $2 \times 2$  mesh target, or SPEs on different processors on the two-Cell QS20 target.

The algorithm is similar to Kernighan & Lin, in that it swaps similar processors in a greedy manner to minimise the maximum load on the buses, even if doing so makes the bottleneck worse. After swapping the loads on two processors, they are fixed for the rest of the pass. The algorithm continues until there are no processors left, and outputs the best partition seen.

#### 3.2.3 Evaluation

This section uses the StreamIt 2.1.1 benchmarks [GTA06] to evaluate our heuristic algorithm and convex connected partitions in general. The StreamIt benchmarks have the two-terminal

### 3.2 Static partitioning

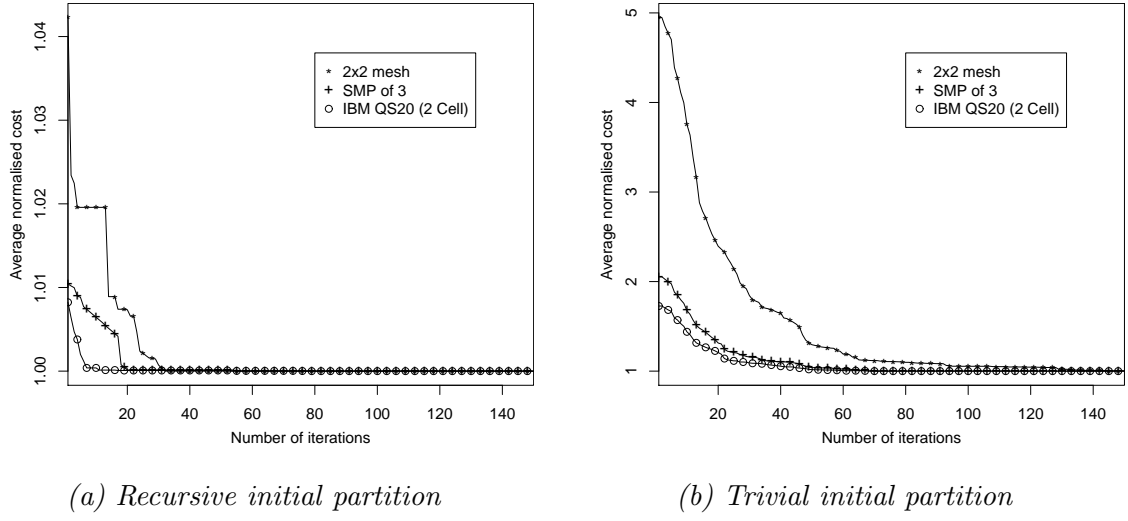


Figure 3.10: Convergence of the refinement phase as a function of the number of iterations

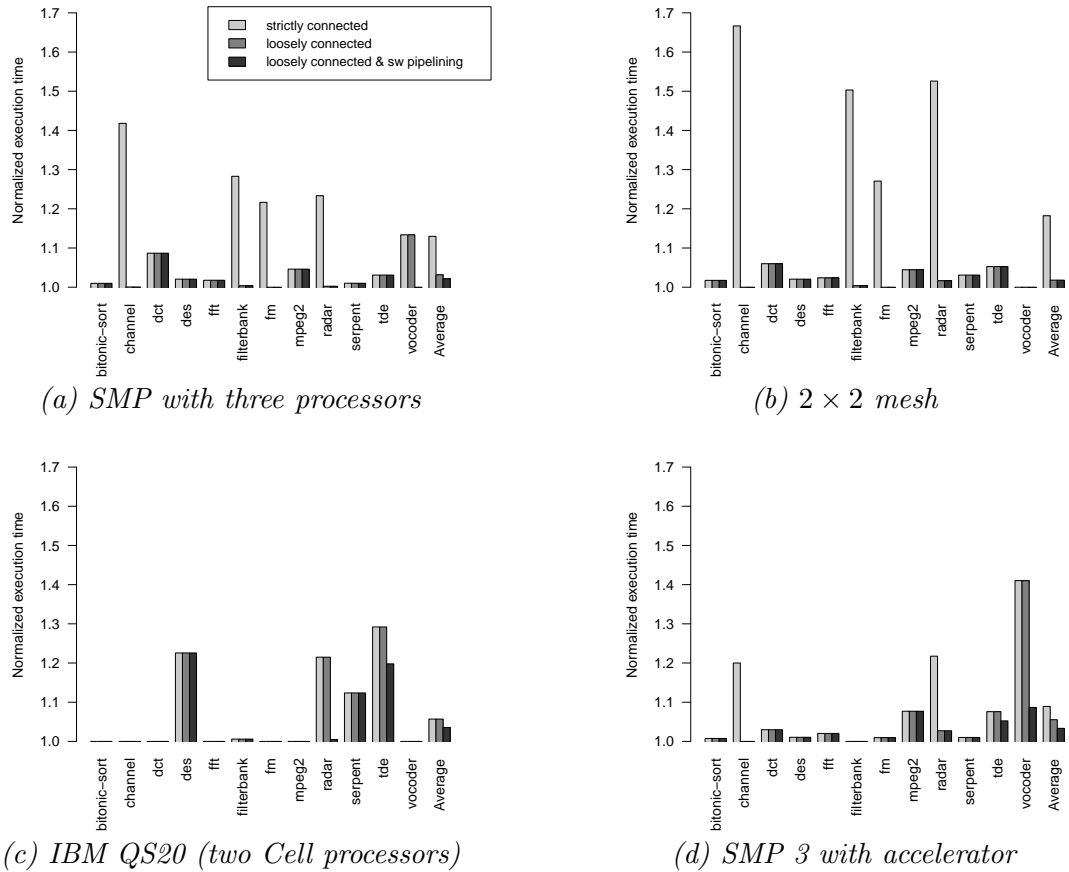


Figure 3.11: Normalised execution time for the StreamIt 2.1.1 benchmarks for the three variants of the heuristic algorithm. The unrestricted partition has execution time 1.0, and larger bars are slower.

### 3. COMPILE-TIME DECISIONS

Benchmark	Num. kernels	Width	Number of pipeline stages for <u>un</u> restricted & <u>h</u> euristic							
			SMP 3		2×2 Mesh		QS20 (2 Cell)		SMP 3 + Acc.	
			u	h	u	h	u	h	u	h
bitonic-sort	40	4	27	(5)	35	(7)	35	(19)	9	(7)
channel	55	17	11	(5)	9	(7)	9	(7)	9	(5)
dct	8	1	9	(5)	9	(7)	9	(9)	13	(5)
des	53	3					77	(31)		
fft	17	1	13	(5)	27	(7)	33	(23)	17	(7)
filterbank	85	16	19	(5)	19	(7)	23	(9)	19	(5)
fm	43	12	9	(5)	17	(5)	21	(9)	17	(5)
mpeg2	23	5	23	(5)	19	(7)	29	(19)	15	(7)
radar	57	12	13	(5)	9	(5)	19	13	19	(5)
serpent	120	2	143	(5)	163	(7)	209	(31)		
tde	29	1	39	(5)	41	(7)	57	33	23	(9)
vocoder	114	17	29	7	29	(7)	29	(7)	45	11
Average ratio			5.9		5.0		2.4		2.8	

Table 3.1: The number of pipeline stages for the optimal unrestricted partitions and the partitions generated by our heuristic (loosely connected with pipelining). Partitions that do not need pipelining are in parentheses.

series-parallel structure of StreamIt, but are the most widely used streaming benchmarks. The program graph, work estimates and data rates were taken from the StreamIt 2.1.1 compiler. The StreamIt compiler modifies the stream program graph before calculating the work estimates, so our kernel counts differ from those of the source program. The number of kernels ranges from 8 to 120, and has average 54.

Figure 3.10 shows performance vs. iteration in the refinement phase. At each point is plotted the minimum of all partitions seen so far. This graph shows that the refinement algorithm quickly converges to a good solution, even from a trivial initial partition. It also shows that the initial partition is on average within 4% of the performance of the final partition, although the worst case is 33% slower. The sub-figures have very different scales on the vertical axes. The graphs show the number of iterations rather than time, since the algorithm was implemented in unoptimised Python. Nevertheless, the per benchmark partitioning time on a 2GHz Intel MacBook is average 10.0 seconds and maximum of 58.4 seconds.

Figure 3.11 shows the normalised execution time for the partitions found by the heuristic, using strict and loose connectivity, against the optimal unrestricted partition, which has time 1.0. The strictly connected partitions for *channel*, *filterbank*, *fm* and *radar* have bad performance because of the wide split joins. The third column of Table 3.1 gives the width of each benchmark, which is the maximum size of a subset of kernels with no paths between any pair of them (an anti-chain). For example, the *filterbank* benchmark, illustrated in Figure 3.1, has a width of 16. The benchmarks with poor performance using strict connectivity tend to be the benchmarks with the largest width, although this is a great simplification.

Figure 3.11 also shows the bottleneck cost for the loosely connected partition generated by our heuristic when software pipelining is enabled. Software pipelining is most beneficial for the *vocoder* benchmark on SMP3 with accelerator, and *radar* on IBM QS20. Figure 3.12 shows the partition of *vocoder* on SMP with accelerator found using the heuristic, which uses eleven pipeline stages if scheduled using the stage assignment phase of the SGMS algorithm [KM08]: six for computation and five for DMA. The main improvement in throughput



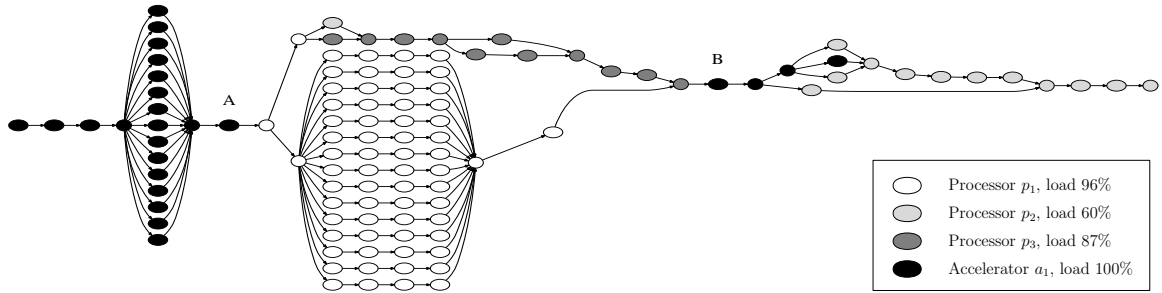


Figure 3.12: Vocoder benchmark on SMP 3 with accelerator using software pipelining.

comes from splitting the workload on  $a_1$  into two tasks. This benchmark has two heavyweight kernels that should run on the accelerator, but the cost of the smallest convex task containing both of them is very large. The optimal unrestricted partition is 9% faster, but it requires 45 pipeline stages.

The only bad results are *des*, *serpent*, and *tde* on IBM QS20. These three suffer since our fast algorithm gets stuck in a local optimum. These benchmarks suggest some limitations of the Create Task stage, which could be addressed in future work. The *des* and *serpent* benchmarks finish with two or more tasks of identical cost. The Create Task stage offloads a kernel from one of the bottleneck tasks, but since it does not reduce the cost of the other bottleneck(s), the move is rejected as not worthwhile. The *tde* benchmark finishes with the bottleneck task containing a heavyweight kernel. The Create Task stage tries to move the heavyweight kernel to another processor, but it cannot find a better solution. It may have been better to move some of the lighter kernels from the bottleneck task. Since the problem is  $\mathcal{NP}$ -hard, all heuristics will at times find suboptimal solutions.

Table 3.1 shows the pipeline lengths for the unrestricted and heuristic partitions. The pipelines were generated using SGMS [KM08], so half of the pipeline stages perform computation. The partitions that do not require software pipelines are given in parentheses. Some of the cells for the unrestricted partitions are empty, since finding the true optimum is slow (Figure 3.11 uses a lower bound). The unrestricted partitions for *serpent* and *tde* on Cell have long software pipelines because the programs themselves are long pipelines, and unless it is bus bound, there is no incentive for short pipelines. There are four data points in Figure 3.11 where our heuristic algorithm used software pipelining.

### 3. COMPILE-TIME DECISIONS

---

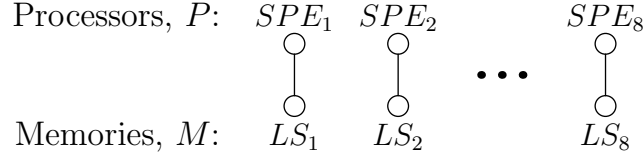


Figure 3.13: Memory constraint graph for the Cell Broadband Engine

## 3.3 Static buffer sizing

### 3.3.1 The buffer sizing problem

Queue length assignment seeks to find an optimal tradeoff, subject to memory constraints, between throughput and latency. We wish to find a close to Pareto optimal solution: that is, neither latency nor throughput can be improved without making the other one worse. Memory use is kept within the constraints, rather than being minimised.

The stream program is represented as a connected, not necessarily acyclic, digraph,  $\mathcal{P} = (T, S)$ , where  $T$  is the set of vertices (tasks), and  $S$  is the set of edges (streams). Each stream  $s$  has a producer and consumer buffer size in bytes,  $b_p(s)$  and  $b_c(s)$ , and a minimum number of buffers, sufficient to hold the working set and avoid deadlocks. If  $P$  is acyclic, as for ACOTES, deadlock is impossible; otherwise minimum sizes can be found using algorithms in the literature [Par95; Buc93; GB03]. The algorithm determines the actual number of buffers,  $n_p(s)$  and  $n_c(s)$ .

Each task has a trace, which is an alternating sequence of computation times and primitives. There are four communications primitives given in Section 2.3: `ProducerAcquire` and `ConsumerAcquire` obtain a buffer to write or read data. `ProducerSend` and `ConsumerDiscard` send or discard data once complete.

The traces are interpreted using the ASM coarse-grain simulator, which takes a machine description that defines the target. Queue length assignment needs only the memory constraints, which are represented using a bipartite graph,  $\mathcal{H} = (R, E)$ . The set of vertices,  $R = P \cup M$ , is a disjoint union of processors  $P$  and memories  $M$ , and the edges,  $E$ , connect processors to their local memories. Each memory has weight equal to the amount of memory available, in bytes, for stream buffers. Figure 3.13 shows the memory constraint graph for the Cell Broadband Engine; the memory weights depend on how much memory is already being used.

The evaluation algorithm in Figure 3.7 and experimental results in Section 3.3.3 both require an estimate of latency. Since it is orthogonal to the rest of this section, and only differences in latency matter, we propose the following scheme.

Define  $f_t(n)$  to be the time of firing,  $n = 0, 1, \dots, M_t - 1$  of task  $t$ , taken from the fire primitive. Since each task contributes to a common amount of real-world progress, normalise  $n$  to the interval  $0 \leq x < 1$  by dividing it by  $M_t$ . Then  $g_t(x) = f_t(\lfloor M_t x \rfloor)$  gives the time that task  $t$  was proportion  $x \in [0, 1)$  through the calculation. The latency,  $L(x)$ , is the difference between the largest  $g_t(x)$  for a sink and the smallest  $g_t(x)$  for a source, which can, unfortunately, be negative when multiplicities are variable. The *latency* is the average value of  $L(x)$ .

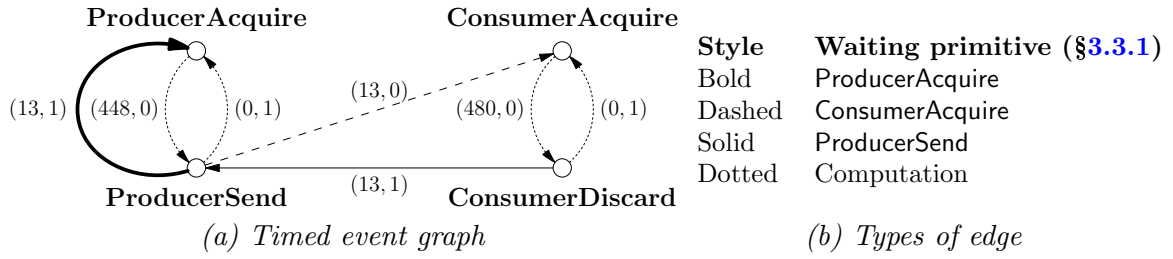


Figure 3.14: Example timed event graph used by the critical cycle algorithm

### 3.3.2 The buffer sizing algorithm

This section describes several algorithms for cycle detection and buffer size update. It first reviews the standard critical cycle detection algorithm, and explains when it is applicable. It then introduces the *baseline* algorithm, which finds the bottleneck cycle by analysing the time each task is blocked on each stream. This data is easy to obtain, and the algorithm is quite effective. It then gives an example that the baseline algorithm gets wrong, and proposes the *token* algorithm, which requires extra bookkeeping but achieves better results. Finally, it describes several variants on the buffer update algorithm, which have different tradeoffs between speed of convergence and latency.

#### Cycle detection algorithms

**Critical cycle algorithm** The *critical cycle* algorithm [IP95; DG98; GG93] solves the cycle detection problem for homogeneous Synchronous Data Flow (SDF) [LM87] with constant computation times and communications latencies. In homogeneous SDF, every time a producer or consumer fires, it pushes or pops a single buffer on each stream. All tasks therefore fire at the same rate. The algorithm can be extended to SDF, where each producer or consumer pushes or pops any fixed number of buffers, but it requires expanding the graph, which can make it much bigger [Lee86].

Figure 3.14(a) shows how producer-consumer, assuming a single buffer at each end, is represented by this algorithm. Each vertex is the return from a communications primitive. The edges are distinguished, for the diagram but not the algorithm, using the convention in Figure 3.14(b), which refers to the primitives in Section 2.3. Each edge has *weight*, which is its fixed computation time or communications latency, and *height*, which is the fixed difference between the firing number, which counts the number of times a task has fired, at its two ends.

For example, at the producer side, the dotted line from **ProducerAcquire** to **ProducerSend**, of weight 448 and height 0, represents computation inside a single iteration. The bold line in the reverse direction, of weight 13 and height 1, is because the producer cannot reuse its single buffer in the current firing until the previous DMA has completed.

Throughput is constrained by the *critical cycle*, which is a cycle with maximum ratio of total weight divided by total height. There are several algorithms to find such a cycle, many based on Karp's Theorem [Kar78], in time  $O(|S|^2|T|)$  or so [DG98], using the terminology of Section 3.3.1.

### 3. COMPILE-TIME DECISIONS

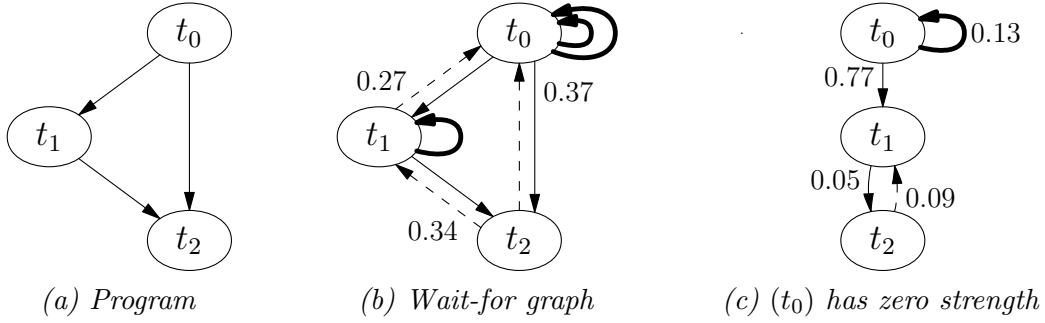


Figure 3.15: Example weighted wait-for graphs

**Baseline algorithm** Our *baseline* algorithm is more general, because it supports variable data rates, computation times, and communication latencies. It finds the bottleneck by analysing wait times in a real execution or simulation.

Figure 3.15 shows how the stream program and wait times are represented by the algorithm. Figure 3.15(a) is an example stream graph with three tasks in a triangle. Figure 3.15(b) is the *wait-for* graph, which has the same three edges per stream as the timed event graph. Following convention for wait-for graphs, the arrows point in the opposite direction, *from* the waiting task. The weight of an edge is the proportion of the total time that the task at the initial vertex, or tail, spent waiting in its communications primitive. The diagram shows three of the edge weights; the other weights will not be important in the discussion.

As for the critical cycle algorithm, performance is constrained by dependence cycles in the wait-for graph. The algorithm uses two bounds, one local and one global, on the maximum increase in performance from *relaxing* a cycle; i.e. increasing buffering on one of the streams in the cycle that gets full.

Consider the potential benefit from relaxing cycle  $C_1 = (t_0 t_2 t_1)$ . This can only be done by increasing buffering on the stream from  $t_0$  to  $t_2$ . Since  $t_1$  waits for 27% of the time, during the **ConsumerAcquire** primitive in this cycle, we could reduce the execution time of  $t_1$  by *at most* 27%, before the cycle disappears. Since all tasks execute for nearly the same amount of wallclock time, any change in throughput will cause all vertices to have their *total* waiting time, not just on the edges of this cycle, reduced by the same amount. It is therefore likely that the edge in the cycle that disappears first is its weakest edge.

The local bound is the *weight* of cycle  $C$ , denoted  $w(C)$ , which is the minimum weight of its edges. If there is no cycle with non-zero weight, then utilisation is already 100%. This is because every directed *acyclic* graph has a vertex with no outgoing edge, which corresponds to a task that never has to wait.

Figure 3.15(c) is the motivation for the global bound. The maximum weight cycle is the loop on  $t_0$ , of weight 0.13, which we will call  $C_2$ . A moment's reflection, however, shows that  $C_2$  cannot really be a bottleneck since neither  $t_1$  nor  $t_2$  ever wait for  $t_0$ , even indirectly. If we reduced the time  $t_0$  spent waiting on this loop, it cannot make  $t_1$  or  $t_2$  go any faster. Since throughput would be unchanged,  $t_0$  must spend the same total amount of time waiting, so the waiting time would move from **ProducerAcquire** to **ProducerSend** (see Figure 3.14(b)).

The global bound is the *strength* of the cycle, denoted  $s(C)$ , which is the lowest value of the maximum flow *through a single path* to the cycle, starting from any other vertex. Since

there is no path at all from  $t_1$  to  $C_2$  in Figure 3.15, the cycle has zero strength:  $s(C_2) = 0$ . In contrast, the cycle  $(t_1 t_2)$  has strength 0.77, because this is the weight of the only path from the only other vertex,  $t_0$ . Increasing the performance of  $t_1$  and  $t_2$  by any means could reduce execution time of the program as a whole by 77%. This cycle is the bottleneck, and it has weight 0.05. The requirement that flow be through a single path makes little difference in practice, but it reduces considerably the algorithmic complexity.

It is possible for the wait-for graph to be disconnected; e.g. when tasks wait for each other only through bus contention. This happens rarely, but it causes all strengths to be zero. Therefore, when all strengths are zero but the utilisation is below some threshold (currently 100%), the strengths are ignored. Since it almost never happens, there is little reason to be more sophisticated.

The strength of each vertex is found by computing the *all-pairs bottleneck paths* [Pol60]. This finds, for every pair of vertices, the value of the maximum flow through a single path from the first vertex to the second. It is solved using a variant of Dijkstra’s algorithm, running Dijkstra for each vertex to find the maximum flow paths into it. The strength of that vertex is given by the path with the lowest flow. The total execution time is  $O(|S||T| + |T|^2 \log |T|)$ , using a Fibonacci heap [FT87b; VWY07], with the terminology of Section 3.3.1.

The algorithm finds a cycle with the maximum value of the minimum of the local and global bounds. It is straightforward to show that we can take account of both simply by replacing the weight of every edge  $e = (a, b)$  by a new weight,  $w'(e) = \min(w(e), s(a))$ . A maximum weight cycle, according to  $w'$ , can be found in time  $O(|S| \log |S|)$ , where  $S$  is the set of streams. To find out whether there is a cycle of weight  $\geq W$ , for some  $W$ , just check whether there is any cycle if you ignore all edges of weight  $< W$ . This can be done in time  $O(|S|)$  by attempting to perform a topological sort. To find a maximum weight cycle, first sort the edge weights, and perturb them so that no two are exactly the same. Then use bisection on the sorted edge weights.

The baseline algorithm uses data that is easy to obtain, and is usually quite effective, but it has one limitation. Since each task is represented by a single vertex, it cannot “see” what is happening inside them.

Figure 3.16(a) shows the wait-for graph for an example where the baseline algorithm makes a bad decision. The maximum weight cycle is  $(t_1 t_0 t_2)$ , which has weight 0.50. Whether or not this is a bottleneck depends on the internal behaviour of tasks  $t_1$  and  $t_2$ . The order of operations per firing of task  $t_1$  is shown in Figure 3.16(b). If it is also known that task  $t_1$  *always* waits in step 5, then reducing the waiting time in step 1 will simply result in a longer waiting time in step 5. It can never advance the push in step 6, so the critical cycle cannot be  $(t_1 t_0 t_2)$ . The next section introduces the *token* algorithm, which addresses this problem, and describes the *indirect wait-for graph* in Figure 3.16(c).

**Token algorithm** The *token* algorithm addresses this problem by tracking dependencies through tasks. This is somewhat similar to causal chains [BH01], except that the aim is to resolve performance bottlenecks rather than artificial deadlocks. Their algorithm fixes a deadlock after it happens, when all tasks have got stuck, but we cannot expect all tasks in a cycle to ever be waiting simultaneously.

During the simulation, or at runtime in a dynamic scheme, each task  $t$  has a *current token*,  $S_t$ , which is the stream that most recently made  $t$  wait, directly or indirectly, because

### 3. COMPILE-TIME DECISIONS

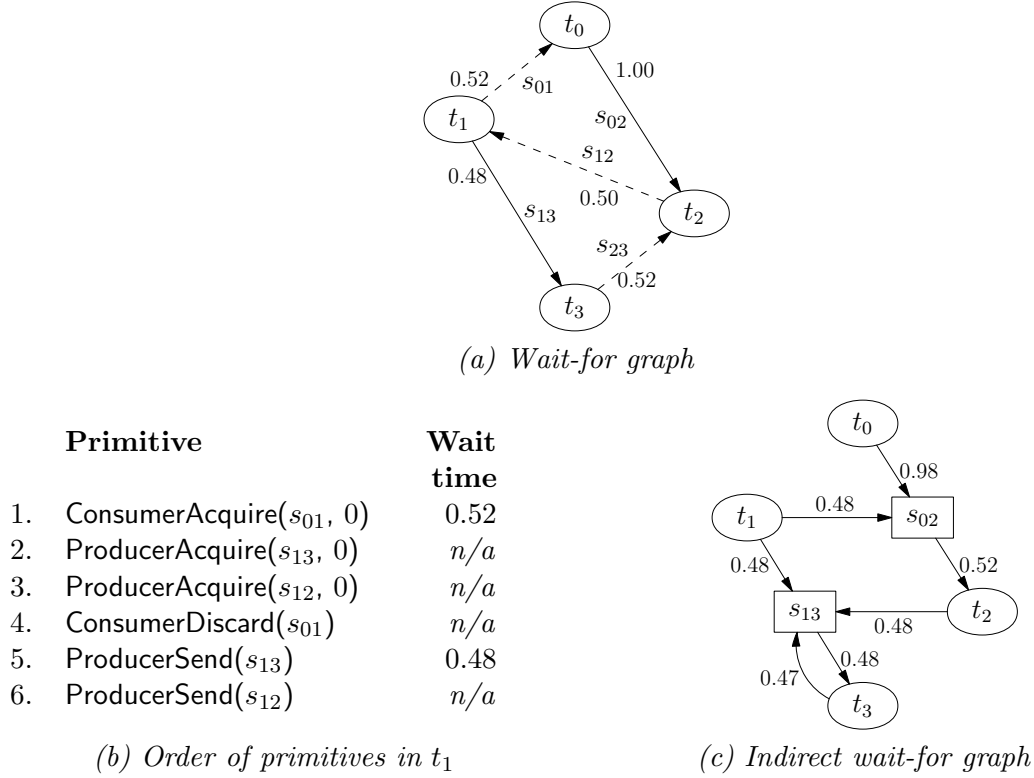


Figure 3.16: Example where baseline fails

it got full. It has a *current waiting time*,  $W_t$ , which measures how much the task has already had to wait, so that only increases in waiting times are charged to streams. It also has a *waiting vector*,  $(V_t)_s$ , which gives the total waiting time for each stream in the whole program. Each consumer buffer  $c$  has a *current token*,  $S_c$ , and *current waiting time*,  $W_c$ , which together record the producer's problem at the time the block in that buffer was sent.

When task  $p$  blocks for time  $\tau$  because output stream  $s$  is full, it sets  $S_p \leftarrow s$  and increases both  $W_p$  and  $V_p[s]$  by  $\tau$ . When task  $p$  sends a block using buffer  $c$  on output stream  $s$ , it records a copy of its current state:  $S_c \leftarrow S_p$  and  $W_c \leftarrow W_p$ . When a task  $q$  blocks for time  $\tau$  because input stream  $s$  is empty, it also, after the data arrives, reads  $S_c$  and  $W_c$ , from the consumer buffer  $c$  containing the end of the data. It then updates its current token  $S_q \leftarrow S_c$  to indicate that it had to wait, indirectly, for whichever stream the producer had to wait for, and calculates the increase in current waiting time  $\Delta W_q \leftarrow \min(\tau, W_c - W_q)$ , which can be either positive or negative. If it is positive, then  $V_q[S_q]$  is increased by  $\Delta W_q$ . In either case, the current waiting time is then updated using  $W_q \leftarrow W_q + \Delta W_q$ .

The waiting vectors are used to construct an *indirect wait-for graph*, as shown in Figure 3.16(c). If  $V_t[s] > 0$ , there is an edge from task  $t$  to stream  $s$  with weight  $V_t[s]/L$ , where  $L$  is the total execution time of the run, in the same units. Each stream  $s$  also produces an edge from  $s$  to its consumer  $q$ . The weight of this edge is  $s(q)$ , the *strength* of  $q$ , as defined for the baseline algorithm.

This is effectively viewing each stream as an actor in its own right, which is always blocked waiting for the consumer to discard its data. This is the most convenient place to

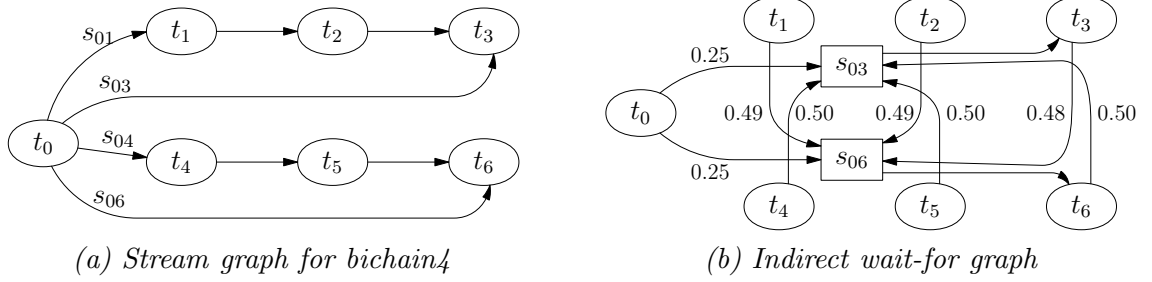


Figure 3.17: Token algorithm: bichain4 example

take account of the strengths, which are still relevant by the same argument as before. The token algorithm finds the maximum weight cycle in the same way as the baseline algorithm.

Figure 3.17 shows a second example which clarifies the need for the cycle-based algorithm outlined above. In the stream program of Figure 3.17(a), task  $t_0$  pushes the outputs in the cyclic order  $(s_{01} s_{03} s_{04} s_{06})$ , waiting only in `ProducerSend` for streams  $s_{03}$  and  $s_{06}$  due to their longer latency.

When it pushes on stream  $s_{04}$  of the right branch, the most recent wait was due to stream  $s_{03}$  being full, so it sends the token for  $s_{03}$ . Similarly, it sends the token for stream  $s_{06}$  to stream  $s_{01}$  of the left branch. The indirect wait-for graph is shown in Figure 3.17(b), with cycle  $(t_3 s_{06} t_6 s_{03})$  going through both streams.

### Buffer size update algorithms

The cycle detection algorithm returns a set of edges in the wait-for graph that cause a bottleneck cycle by becoming full. Relaxing the cycle involves increasing memory on one or more of these edges. The purpose of the buffer size update algorithm is to determine *which* edges to enlarge, and by *how many* buffers.

Our simplest algorithm is *miserly*, meaning that it starts at the minimum number of buffers, mentioned in Section 3.3.1, and each iteration increases the allocation of a single buffer by one. The other algorithms speculatively assign spare memory, and only take it away if it is needed elsewhere. For all these algorithms, each stream  $s$  *demand*s some number  $d_s$  of buffers, as for the miserly algorithm, and *request*s another  $r_s$  to be granted out of unused memory, if there is any. When there is not enough memory to grant all requests within some memory, we used the following algorithm. The total request in bytes is  $R = \sum r_s b_c(s)$ , where  $b_c(s)$  is the size in bytes of a single consumer buffer for stream  $s$ . If  $M$  bytes are left after granting all demands, so  $R > M$ , then each stream is initially granted  $\lfloor r_j M / R \rfloor$  extra buffers, then possibly one more, if it fits.

In our first alternative, *double*, each edge requests an extra buffer if it is currently allocated only one. In our second alternative, *exponential*, the request is for some multiple,  $f - 1$ , of the number of buffers demanded. It still uses a greedy update algorithm, so that when the number of buffers is increased, the edge demands, on the next iteration, one more buffer than it was given in total last time. The results use  $f = 2$ , so an edge will demand  $2^k - 1$  buffers, and request an equal number, for  $k = 1, 2, \dots$ , until it is given fewer buffers than it wants.



### 3. COMPILE-TIME DECISIONS

---

The third alternative, *level*, uses the *top level*, the length of the longest path from a source node, and *bottom level*, the length of the longest path to a sink node. The algorithm the same as *exponential*, except that the request is the maximum of a)  $f - 1$  times the number of buffers demanded, b) twice the difference in top level, and c) twice the difference in bottom level. This tries to give a high initial allocation to streams that cross a high latency.

#### 3.3.3 Evaluation

This section uses the StreamIt 2.1.1 benchmarks [GTA06], random graphs, and sixteen examples, including *chain8*, *producer-consumer*, *bad-baseline*, and *bichain4*. For the StreamIt benchmarks, the program graph, work estimates and communications rates were generated by the StreamIt compiler. The algorithm in Section 3.2 was used to produce partitions for an IBM QS20 blade, which has two Cell BEs.

**Buffer size update** The first three rows of Figure 3.18 compare the buffer update algorithms from Section 3.3.2. These plots also contain results for Basten and Hoogerbrugge (B&H) [BH01] and modified StreamRoller [KM08], which will be discussed in Section 3.4. The left column shows as a function of the iteration number, the utilisation, which is proportional to throughput, as remarked at the end of Section 3.1.3. The right column shows the tradeoff between latency and utilisation. Any points that cannot be Pareto optimal, because they are beaten on both utilisation and latency by some other point to the top-left, have been removed.

The first row is for random stochastic graphs with 32 tasks and 50 streams. The graphs are connected and acyclic, but otherwise unstructured. The computation time of each task is normally distributed with a random mean and variance (clamped above zero). Notice that B&H has poor performance and, since it increases buffering where it isn't necessary, high latency.

The upper bound on utilisation was found using an exhaustive search over all allocations of the buffers on the processor,  $p$ , whose memory bound caused the level algorithm to terminate. All other queues on other processors were set to their maximum possible size, assuming that all other queues in the same memory had their minimum size. Since this tends to allow a task near the beginning of the stream graph to work flat out filling downstream buffers, the steady state utilisation would be known only after many firings. Instead, we took the utilisation of the task on  $p$ , and scaled by the ratio of the long-term processing times of the most heavily loaded processor and of  $p$ .

The second row shows the StreamIt 2.1.1 benchmarks, with an unroll factor of 100. The third row shows the *stochastic* StreamIt benchmarks, which have normally-distributed computation times, and are intended to show how the algorithms fare for realistic program graphs.

The left column shows that the level algorithm always provides the fastest convergence. The modified StreamRoller algorithm is similar to the first iteration of the level algorithm, and B&H is considerably worse. The level heuristic initial allocation is within 15% of the upper bound on optimal performance, and is increased to within 3% of optimal after four iterations.

**Cycle detection** This section evaluates the cycle detection algorithms only, using greedy buffer update without memory constraints. When task execution times and communications rates are constant, and bus contention is negligible, the critical cycle algorithm of Sec-



tion 3.3.2 is optimal. The last row of Figure 3.18 shows the utilisation and latency for an average of six random graphs with stochastic computation times. The poor performance of the critical cycle algorithm (about 60% utilisation), is because it is unable to detect cycles that arise from execution time variability. The baseline and token algorithms achieve similar performance, although the token algorithm achieves slightly lower latency.

We also evaluated the cycle detection algorithms when there is high bus utilisation. The critical cycle algorithm cannot model increased communication latency due to contention [HP07, §E.5]. For a benchmark with a single producer task connected to two consumers, and bus usage close to 100%, the critical cycle algorithm achieves about 70% utilisation. The baseline and token algorithms measure waiting times directly, and consistently achieve 100% utilisation.

### 3.4 Related work

**Partitioning** There has been a great deal of work in automatically mapping stream programs onto multiprocessor systems. The Ptolemy II software environment [EJL<sup>+</sup>03] is an actor-based model for real-time embedded systems that supports several models of computation, including Synchronous Dataflow (SDF) and Kahn Process Networks (KPN). Related work from the Ptolemy project explores the more theoretical aspects of partitioning and scheduling data flow graphs for multiprocessors [HL91].

The Stream Graph Modulo Scheduling (SGMS) algorithm is part of StreamRoller [KM08], a StreamIt compiler for the Cell Architecture. This algorithm splits stateless kernels, partitions the graph, and statically schedules. The splitting and partitioning problem is translated into an Integer Linear Programming (ILP) problem, which is solved using CPLEX [ILO]. This approach uses mature technology to solve the ILP problem; it also applies kernel splitting in the same step, rather than using the iterative approach we follow.

Their partitioning algorithm considers only CPU loads, and ignores communications bandwidth. This may be sufficient for a single Cell processor, which has a high-bandwidth on-chip bus, but it is inappropriate when communication is off-chip, as in the Cell QS20 target, or when a bottleneck may appear in part of an on-chip network, such as a large mesh.

The StreamRoller ILP formulation does not attempt to find a partition that minimises the memory, latency and startup overheads introduced by software pipelining. Since it uses an ILP solver to find a (close to) optimal solution to a problem with similar objective and constraints to our unrestricted partition, the resulting pipeline length should be similar. StreamRoller does not have any concept similar to our connectivity constraint. We believe that when the program is written using an unrestricted programming language, the partitioning algorithm requires some mechanism to model which kernels can be statically scheduled by the compiler. They do not restrict the memory footprint on each processor, although it appears that their ILP formulation could be extended to do so.

Flexstream [HCK<sup>+</sup>09] uses a related algorithm, *adaptive stream graph modulo scheduling* to map a stream program to the Cell Architecture. It is a hybrid static-dynamic approach which is able to adapt to changes in resource availability. The static work partitioning algorithm uses ILP to map the program to the most powerful target; e.g. the full machine. The dynamic partition refinement stage is a heuristic that adapts the partition at run time

### 3. COMPILE-TIME DECISIONS

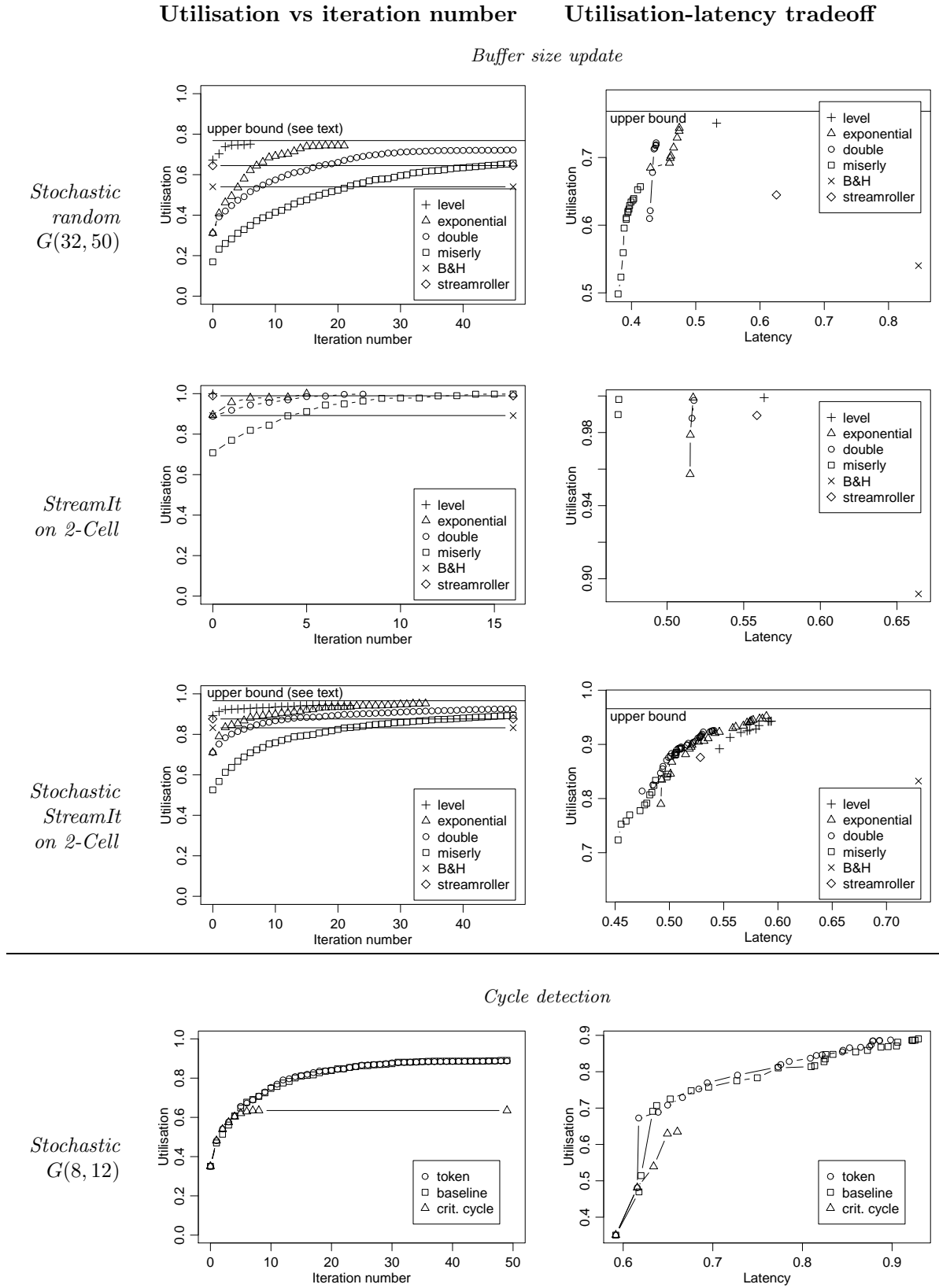


Figure 3.18: Comparison of the buffer size update and cycle detection algorithms

to take account of the resources that are actually available; e.g. if another application is running.

The StreamIt compiler [GMA<sup>+</sup>02; GTA06] targets the Raw Microprocessor [WTS<sup>+</sup>97], symmetric multicore architectures, and clusters of workstations. This is a long running project with a publicly available compiler and benchmark suite. The StreamIt source language imposes a structure on the stream program graph, where each kernel has a single input and a single output, and kernels are composed in pipelines, split-joins, and feedback loops. Since the kernels have static data rates, the compiler can fuse any set of kernels. The default partitioner uses dynamic programming. Our model of the source program is more general, since we target unstructured program graphs with variable data rates, and we use the connectedness constraint to reason about the capabilities of the compiler. Our model of the target system is also more general, since we can target a heterogeneous multiprocessor system with any communications topology.

Liao et al. [LDWL06] use affine partitioning to map regular multidimensional programs written using the Brook language [Buc03] onto a four-processor SMP. The R-Stream compiler ([www.reservoir.com/r-stream.php](http://www.reservoir.com/r-stream.php)) is a proprietary high level compiler for stream programs, which uses a polyhedral model to partition code and data to a parametric parallel machine. Gedae [LBS] is a proprietary GUI tool for mapping data flow graphs to a heterogeneous multiprocessor system. The transformations are under user control, and the partition is not automatically found by the compiler.

Decoupled Software Pipelining (DSWP) [RVVA04] is a technique to tolerate variable latency instructions in loops. It breaks a loop into strongly connected components, which execute on different threads. The threads communicate using a synchronization array, a hardware structure that provides low overhead blocking queues between threads. DSWP can also exploit fine-grained parallelism, by mapping the strongly connected components to different cores.

**Queue sizes** Basten and Hoogerbrugge (B&H) [BH01] is the only other work that also targets unstructured graphs with variable multiplicities and computation times. Their algorithm sets each FIFO buffer size to be proportional to the amount of data streaming through it. This gives a relative size for each buffer, but it is not motivated by the underlying problems discussed in Section 3.1.3, and has poor performance in Figure 3.18. We interpreted B&H to mean double buffering on the producer side, with all the remaining memory allocated to consumer buffers, rounding the number of buffers *up* to an integer. If rounding up causes the buffer allocation to not fit, we reduced the target memory use until it did fit. The *chain8* example in Figure 3.5 shows the problem with this heuristic. If all data rates are the same and there is enough memory on  $t_n$  for ten buffers, Basten and Hoogerbrugge allocates five buffers to each stream for 70% utilisation, while our heuristic allocates eight to  $(t_1, t_n)$  and two to  $(t_{n-1}, t_n)$  for 100% utilisation.

The SDF tool [SGB06] uses an exhaustive search to find all Pareto-optimal buffer allocations for an SDF graph. It requires exponentially many steps, and only supports constant computation times and data rates. For an  $n$ -way split or join where each stream needs  $b$  buffers, their algorithm requires  $n^b$  steps, while our level algorithm requires  $O(n \log_2 b)$  steps to find a single solution.

StreamRoller [KM08] performs buffer allocation as part of software pipelining, but it is restricted to graphs with fixed multiplicities and computation times. The algorithm is similar

### 3. COMPILE-TIME DECISIONS

---

to the first iteration of the *level* algorithm, in that the number of buffers allocated to a stream is always one plus the difference in pipeline stage. The *chain8* example in Section 3.1.3 shows that this is conservative, even when there is no variability. Hence the StreamRoller algorithm can require more memory than necessary; if there is insufficient memory, it fails.

Due to the unrolling factor we used, StreamRoller failed on at least one benchmark for all of the graphs in Figure 3.18. This is true even for the StreamIt benchmarks, for which our algorithm achieves 100% utilisation on at least one processor. We modified StreamRoller to use our arbitration scheme described in Subsection 3.3.2, and obtained the results shown in Figure 3.18. Even with this modification, however, our iterative algorithm has about 13% higher performance for the stochastic random graphs and stochastic StreamIt benchmarks.

The SPIR compiler [CLC<sup>+</sup>09] extends StreamRoller to find a partition and software pipeline subject to memory and latency constraints. Unlike our approach, computation times and communication rates are constant. As for StreamRoller, the number of buffers allocated to a stream is one plus the difference in pipeline stage. Since the problem cannot be solved exactly using ILP, it is a heuristic which uses two passes of the commercial CPLEX ILP solver. Our algorithm could be used to improve the buffer allocation of a partition produced by SPIR.

## 3.5 Conclusions

This chapter introduced a new partitioning heuristic for stream programs. Unlike previous work, it takes account of the partition’s effect on software pipelining and buffer allocation. The algorithm controls the length of the pipeline using the convexity constraint. Unlike previous work, it has a flexible mechanism to take account of the compiler’s ability to fuse kernels.

This chapter also introduced a queue sizing algorithm, which allocates the memory in local stores to streams. Unlike several previous algorithms, it supports streaming programs with non-constant multiplicities and variable computation times. It also achieves higher performance and lower latency than previous algorithms.

## Chapter 4

# Run-time Decisions

The previous chapter introduced two new algorithms for the stream compiler. The stream compiler transforms the source code into an executable, performing optimisations using the information known before the program starts running. This chapter focuses instead on the decisions taken at run time. In particular, this chapter is concerned with dynamic scheduling of stream programs.

A program can be scheduled either statically, by the compiler, or dynamically, by the run-time system. Chapter 3 was concerned with static scheduling. If the program is scheduled statically, each task becomes a thread containing a loop, each iteration of which executes its kernels' work functions in sequence. Tasks communicate using communications primitives, which they call as and when they need them. This implies a relatively simple run-time system, similar to *acolib*. The main advantages of static scheduling are low overhead and low memory use.

If the program is scheduled dynamically, each task is broken up into self-contained, non-blocking tasks. The dynamic scheduler decides in which order to run these tasks, constrained by dependencies between them: a task cannot start if it needs an output from a task that hasn't yet finished. Tasks consume input when they start and produce output when they complete, rather than communicating as required using communications primitives.

Dynamic scheduling is the only choice when the program's behaviour is unpredictable. It also supports stream graphs that are not known at compile time, because kernels and streams are introduced dynamically. As illustrated in Figure 4.1, a dynamic scheduler adds overhead, but it can be better overall when the partitioned stream program is unbalanced. The discussion in Section 4.2.2 gives an example.

A dynamically scheduled program still benefits from unrolling and partitioning transformations, since they coarsen the granularity, and bigger tasks have smaller total overhead. In addition, unrolling enables vectorisation, and task fusion enables data reuse and polyhedral transformations. A partitioning transformation for a dynamic scheduler, unlike the algorithm in Chapter 3, does not need to map tasks to processors.

Section 4.1 defines the dynamic scheduling problem. Section 4.2 describes the previously known scheduling algorithms, and gives a worst case example for each. Section 4.3 develops an adaptive scheduler for stream-like applications. There are two variants of this scheduler, the *apriority* stream scheduler requires a one-dimensional stream program, and it needs

## 4. RUN-TIME DECISIONS

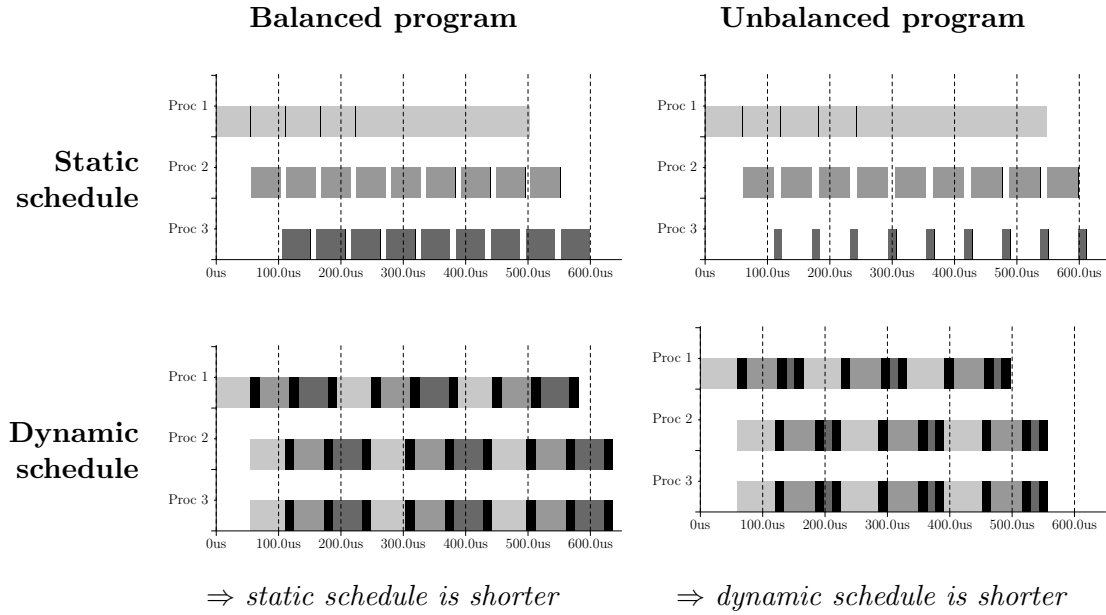


Figure 4.1: Example traces with static and dynamic scheduling (black is scheduling overhead and shades distinguish kernels)

annotations from the compiler. The *gprior* general-purpose scheduler is more general, and it does not need any additional information. Section 4.4 is the experimental evaluation.

### 4.1 The dynamic scheduling problem

#### 4.1.1 Interface to the dynamic scheduler

The stream program starts executing on one processor, in a sequential master thread. Whenever the master thread reaches a function marked as a task, the function does not execute right away. Instead, an instance of that task is *created*, and added to the partially-generated dependency graph (PDG). The PDG holds tasks until they have finished executing. Each vertex is a task, and each directed edge represents a dependency between tasks. When the predecessors of a task, if any, have all finished, the task becomes *ready*. Ready tasks are held in a data structure known as the *ready queue*, until they are issued, in an order determined by the dynamic scheduler. Figure 4.2 illustrates this process, showing the dynamic scheduler in context.

The dynamic scheduler supports the three operations labelled in Figure 4.2:

```

void create-task(Task task-id, Task-set preds)
Task issue-task(Worker worker)
void complete-task(Task task-id)
  
```

The *create-task* operation adds a task to the partial dependency graph (PDG). Its arguments identify the task, and the set of tasks whose outputs it consumes. The *issue-task* operation is called by a worker when it is idle; it waits for and returns the next ready task to issue. The *complete-task* operation is called by a worker when it finishes executing its task; it

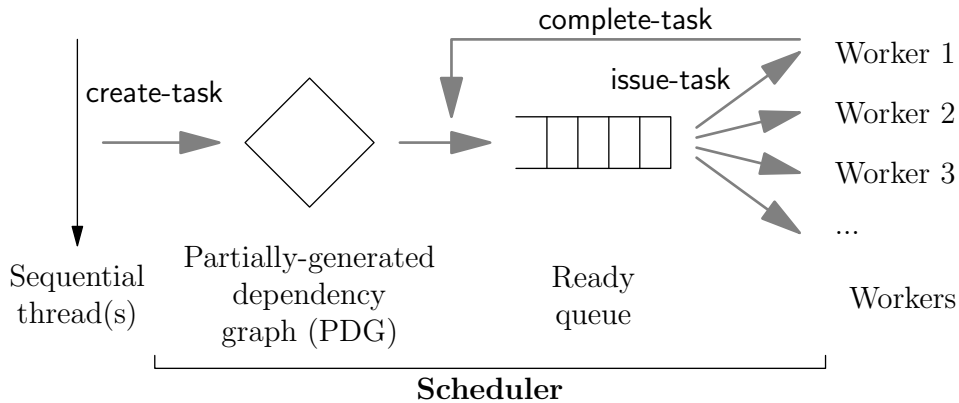


Figure 4.2: Dynamic scheduler in context

notifies the scheduler that the outputs from that task are now ready. A complete scheduler would provide some additional operations, to support initialisation, barriers, and so on, but they are not relevant here. This interface is similar to the interface between Nanos++ [Nana] and its scheduler plugin.

This construction has some direct implications. First, tasks are **non-preemptive**, so once issued, they run to completion. Second, the scheduler is **non-clairvoyant**, meaning that task execution times are not known until they complete. Third, the dependency graph is built by a sequential thread, and is not known all at once.

#### 4.1.2 Throttle policy

Another aspect of scheduling is the *throttle policy*. The program may create a large number of tasks in the course of its execution, so the main thread should be stalled if the PDG contains too many tasks. On Nanos++, this is done by inlining the next task, effectively making the rest of the master thread a new task dependent on the newly created one. This is a reasonable way to prune a recursive computation such as quicksort, which would otherwise create a large number of very small tasks. On CellSs, the main program on the PPE simply stalls, and the processor does not become an extra worker, since all the workers are SPEs.

The decision to stall task creation is taken by the throttle policy. In Nanos++, the default throttle policy is based on the number of ready tasks per processor, inlining tasks if the number of ready tasks is greater than some threshold.

For one-dimensional stream programs, the number of ready tasks is usually confined to a relatively narrow region, as shown in the histograms in Figure 4.3. These histograms show the distribution of the number of ready tasks, sampled at a constant rate. If the threshold is greater than ten, it is never exceeded, and the size of the PDG will grow without limit. If it is less than ten, the main thread will stall too frequently. A better throttle policy is one based on the number of tasks in the PDG. The main thread should be stalled when the PDG contains a certain number of iterations' worth of tasks.

## 4. RUN-TIME DECISIONS

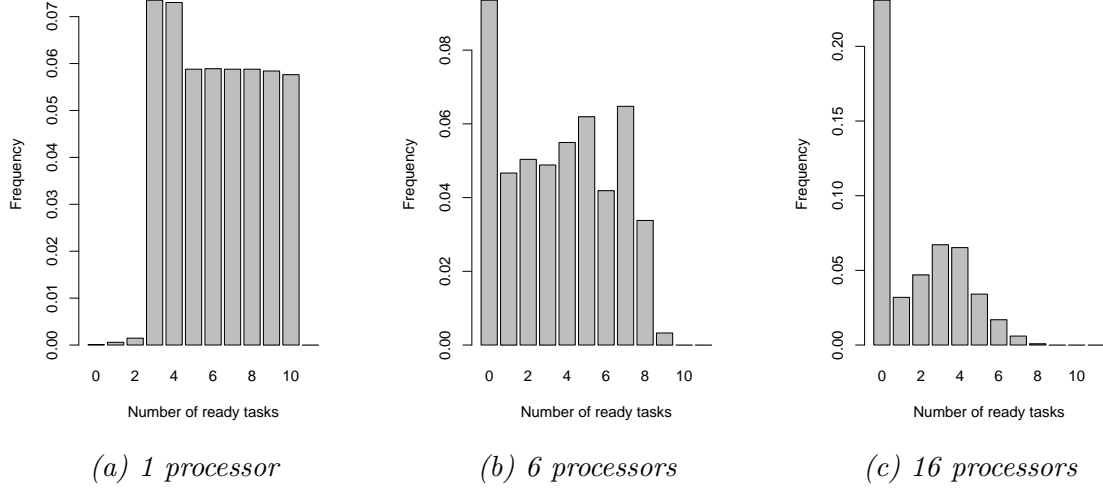


Figure 4.3: Number of ready tasks, excluding those running: `channelvocoder`, using oldest-first

### 4.1.3 Objective function: comparing schedulers

The precise goal of the scheduler depends on the context. An *offline* stream program, such as video transcoding or a database query, should be finished as soon as possible. In standard terminology, the scheduler’s objective would be to minimise the makespan, written  $C_{\max}$ , the time that the last task finishes.

Other stream programs, such as video playback, operate in real time. This suggests that an additional goal should be to minimise latency, so that an output frame isn’t delayed by the processing of too many tasks from future frames.

An additional constraint on the scheduler is memory footprint. If the memory footprint is too large, either performance will suffer through paging to disk, or the program will fail due to insufficient memory.

## 4.2 Survey of DAG scheduling techniques

The scheduling problem is  $\mathcal{NP}$ -hard, because it includes a well-known  $\mathcal{NP}$ -hard problem as a special case. This corresponds to the case where the dependency graph is known up-front, a problem known as  $p|prec|C_{\max}$  [LK78; UI175].<sup>1</sup> The scheduler therefore has to use a *heuristic*, which tries to find a good solution that might not be optimal. Many heuristics have been proposed, for various related scheduling problems, and a selection is given in Table 4.1. Many of these heuristics are unsuitable because they require the task execution times, they need to schedule the whole program in advance, or they are too slow.

This section investigates the performance of the scheduling algorithms listed in Table 4.2. These policies are non-preemptive, non-clairvoyant, online, and fast. The next subsection

<sup>1</sup>That is,  $p$  processors, with precedence relations, and objective to minimise the maximum completion time.



## 4.2 Survey of DAG scheduling techniques

Abbrev.	Full name	Year	Online?	Hetero- geneous?	Clair- voyant	Note
HLFNET [ACD74]	Highest Levels First with No Estimated Finish Times	1974	Offline	Homo	No	Same as botlev
SCFNET [ACD74]	Smallest Co-levels First with No Estimated Finish Times	1974	Online	Homo	No	Same as toplev
DSH [KL88]	Duplication Scheduling Heuristic	1988	Offline	Homo	Yes	$O(n^4)$ is too slow
ETF [HCAL89]	Earliest Task First	1989	Online	Homo	No	Schedules task that would start first
MCP [WG90]	Modified Critical Path	1990	Offline	Homo	Yes	Like botlev, but takes account of descendants' bottom levels
MH [ERL90]	Mapping Heuristic	1990	Offline	Hetero	Yes	$O(n^2p^3)$ is too slow
GDL [SL93]	Generalised Dynamic Level	1993	Offline	Hetero	Yes	
DSC [YG94]	Dominant Sequence Clustering	1994	Offline	Homo	Yes	
BIL [OH96]	Best Imaginary Level	1996	Offline	Hetero	Yes	
HEFT [THW99]	Heterogeneous Earliest Finish Time	1999	Offline	Hetero	Yes	
MCT [MAS+02]	Minimum Completion Time	1999	Online	Hetero	Yes	
MET [MAS+02]	Minimum Execution Time	1999	Online	Hetero	Yes	
PCT [MAS+02]	Partial Completion Time	1999	Online	Hetero	Yes	
OLB [MAS+02]	Opportunistic Load Balancing	1999	Online	Hetero	Yes	
CPOP [THW99]	Critical Path on a Processor	1999	Offline	Hetero	Yes	
k-DLA [WY02]	k-Depth Lookahead	2002	Offline	Hetero	Yes	
HCPT [HJ03]	Heterogeneous Critical Parent Trees	2003	Offline	Hetero	Yes	

Table 4.1: Known DAG scheduling heuristics, related to the problem in this chapter






		Top level	Bottom level	Criticality	#Children	#Descendants
A		0	3	3	3	4
B		1	2	3	1	2
C		1	1	2	1	1
D		2	1	3	1	1
E		3	0	3	0	0

Figure 4.4: Metrics used by the scheduling policies

describes the policies in detail. Subsection 4.2.2 gives a simple realistic worst-case example for each policy. Subsection 4.4 is an empirical analysis on a 16-core machine.

### 4.2.1 The online scheduling policies

Table 4.2 lists the scheduling policies investigated in this section. All policies work by issuing a ready task with the highest *priority*, where the priorities may be adjusted dynamically, and they differ, of course, between the policies. It is possible to construct examples that cannot be scheduled optimally by any priority-based scheduler [Koh75], since the optimal schedule may require *unforced idle time*: a processor waiting idle for some time, even though there are some ready tasks. Section 4.2.2 gives an example stream program, observed in practice, which has poor performance for this reason, when using the oldest-first scheduler.

The **fifo** (first in–first out) policy schedules the task that became ready *first*. Conversely, the **lifo** (last in–last out) policy schedules the one that became ready *last*. For recursively generated computations, such as quicksort, fifo tends to open up parallelism, because it

#### 4. RUN-TIME DECISIONS

Policy	Description	Priority known when	Priority at ready time	Ready queue	Ties possible	Per-task complexity
fifo	FIFO (ready-first)	ready	known	circular buffer	yes	$O(1)$
lifo	LIFO (ready-last)	ready	known	stack	yes	$O(1)$
oldest	Earliest created task first	created	known	priority queue	no	$O(\log R)$
toplev	Minimum top level (depth)	created	known	priority queue	yes	$O(\log R)$
botlev	Maximum bottom level	issued	lower bound	priority queue + map	yes	$O(W)$
crit	Maximum top+bottom level	issued	lower bound	priority queue + map	yes	$O(W)$
mchild	Maximum number of children	issued	lower bound	priority queue + map	yes	$O(\log R)$
mdesc	Maximum num. descendants	issued	lower bound	priority queue + map	yes	$O(W)$
wave	Stream program wavefront	created	known	priority queue	yes	$O(\log R)$
apriority	From Section 4.3	ready	known	priority queue	yes	$O(\log R)$
gpriority	From Section 4.3	ready	known	priority queue	yes	$O(\log R)$

Table 4.2: The scheduling policies and cost, where  $W$  is the number of tasks in the PDG, and  $R$  is the number of ready tasks

schedules larger tasks near the top of the call tree first, doing the computation in breadth-first order. In contrast, **lifo** has lower memory use and better data locality, and would often be the best schedule for a single processor. The Cilk scheduler [BJK<sup>+</sup>95] uses LIFO for local queues and FIFO for work stealing.

The **oldest** policy schedules the task that was *created* first. This policy is similar to FIFO when all tasks are independent.

The remaining policies use the metrics illustrated in Figure 4.4. The **toplev** policy schedules a task with the minimum top level; the top level is the number of edges in a longest path from a source vertex to the task. Similarly, the **botlev** policy schedules a task with the *maximum* bottom level, which is the number of edges in a longest path from the task to a sink vertex. The **toplev** and **botlev** policies are also called SCFNET and HLFNET, respectively [ACD74]. The **botlev** policy was used for task scheduling in 1969 [Bow69], and later, when taking account of known computation times, it was known as HLFET and the Critical Path Method.

There is an important difference between the top and bottom levels. The top level is easily calculated in task creation, and is then known for certain, but the bottom level must be calculated bottom up. Adding a task to the PDG may change the bottom level of every other task. The average case will normally be less expensive, since the traversal can terminate whenever it reaches a task whose bottom level is not changed. The bottom levels may be updated in this way, either every time a task is created, or periodically.

The **crit** policy schedules a task with the maximum *criticality*, which is the sum of the top and bottom levels. The **crit** policy is equivalent to choosing a task with the least *slack* (sometimes called the mobility). The **mchild** policy schedules a task with the greatest number of successors. The aim is to try to open up parallelism, but the policy is short-sighted because it is local, and it is not adaptive, because it keeps opening up parallelism even if there is plenty enough already. The **mdesc** policy schedules a task with the greatest number of descendants. It is expensive and has poor performance.

The **wave** scheduler is specific to one dimensional stream programs. It schedules a kernel with lowest wave number, where the wave number of a kernel is equal to the iteration number

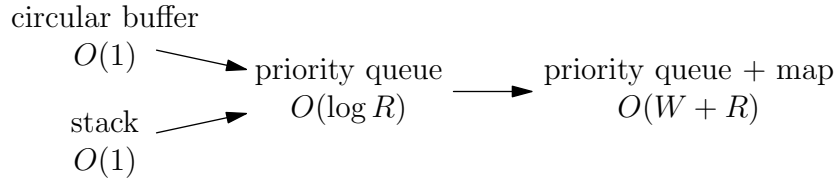


Figure 4.5: Lattice of ready queues for a heuristic

plus the kernel’s top level within the iteration. It is equal to the top level when all kernels have multiplicity one.

The **apriority** and **gpriority** policies are the adaptive schedulers that will be introduced in Section 4.3. The results are presented in this section so they can be more easily compared with the existing schedulers.

All policies except oldest-first may give several tasks the same priority. Often the tie will be broken in some arbitrary way, but this could be done using a second scheduling policy. Continuing like this, a *heuristic* is a list of policies; e.g. (fif, oldest) is a FIFO scheduler, but if several tasks became ready at the same time, they are scheduled in increasing order of their creation time.

### Scheduler complexity

Table 4.2 summarises the overall amortised per task time complexity of the policies. The complexity takes account of the cost of assigning priorities, updating priorities when other tasks are created, and pushing and popping tasks to the ready “queue”.

The table shows the ready “queue” data structure required by each policy. The fif policy, on its own, simply uses a circular buffer for the ready queue. There is no need to explicitly track priorities, because the priority of a task is known when it becomes ready, and is implied by its position in the circular buffer. Similarly, lifo uses a stack. For fif and lifo, insertion and removal are both constant time:  $O(1)$ .

The oldest, toplev and wave policies use a priority queue, because the task’s priority is known at creation time, and therefore known at ready time, when the task is inserted into the queue. Similarly for apriority and gpriority. Both insertion and removal have cost  $O(\log r)$ , using a heap or self-balancing binary tree, where  $r$  is the current size of the ready queue. The average value of  $\log r$  is written  $\overline{\log R}$ . In both cases, calculating the priority itself is constant time:  $O(1)$ , so overall the cost per task is  $\overline{\log R}$ .

For the remaining policies, task priorities are not known until issue time, but a lower bound is known at ready time. For example, the mchild policy schedules the task with the greatest number of children. When a task becomes ready, some of its children may already have been created, but not always all of them. Each time another task is created, all of its parents require their priority to be increased by one. The number of ready parents is at most the number of function arguments, so the extra number of updates per task is  $O(1)$ .

These policies can be implemented using a priority queue, since priority queues typically provide an operation that increases an element’s priority. It also requires a map from the task ID to its position in the priority queue; e.g. the node in a tree, and this map must be updated as the priority queue is manipulated. Task removal has cost  $O(\log r)$ , as before, but

## 4. RUN-TIME DECISIONS

Heuristic	Throughput (compared with optimal)	
	Exhaustion	Back pressure
Optimal static, without fission	$\frac{1}{p}$	$n/a$
Optimal static, with fission	$\frac{p+1}{2p}$	$n/a$
(oldest)	$\frac{p}{2p-1}$	$\frac{3p-4}{2p-2}$
(fifo, *)	$\frac{p}{2p-1}$	$\frac{2p}{3p-1}$
(lifo, *)	$n/a$	$\frac{p}{2p-1}$
(toplev, *)	$\frac{p}{2p-1}$	$\frac{p}{2p-1-f^*}$
(botlev, *)	$\frac{p}{2p-1}$	$\frac{p}{2p-1-f^*}$
(crit, *)	$\frac{p}{2p-1}$	$\frac{p}{2p-1}$
(mchild, *)	$\frac{p}{2p-1}$	$\frac{p}{2p-1}$
(mdesc, *)	$\frac{p}{2p-1}$	$\frac{p}{2p-1-f^\dagger}$
(wave, *)	$n/a$	$n/a$

Table 4.3: Throughput for one dimensional stream programs, compared with optimal, where  $p$  is the number of processors,  $f$  is the unroll factor,  $f^* = \frac{p-1}{f}$ ,  $f^\dagger = \frac{p-1}{f+1}$

task creation requires some of the priorities of the tasks in the ready queue to be updated. Using a balanced tree or heap, updating the priority of a single task in the ready queue is  $O(\log r)$ ; using a Fibonacci heap, it is  $O(1)$  amortised time [FT87a].

For botlev, crit, and mdesc, adding a single task may change the priority of every other task in the PDG. There are simple, common, examples, for each of the three policies, that exhibit this worse case.

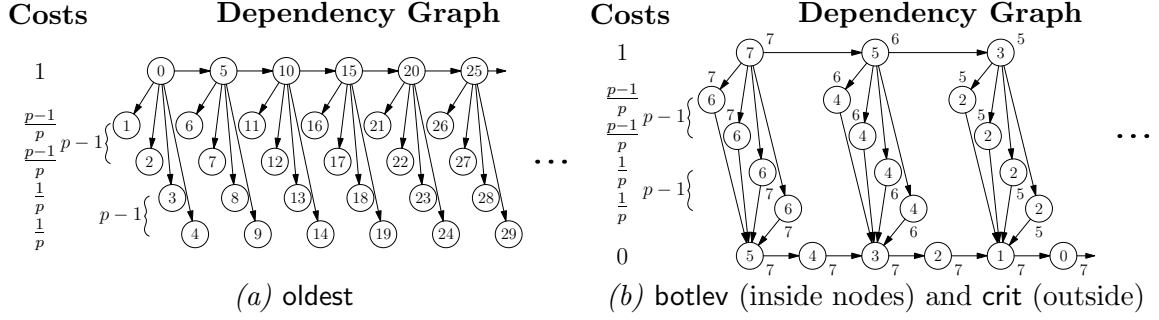
A heuristic is a list of policies, and the ready queue for a heuristic is the *most general* of the ready queues of its policies; i.e. the one that is furthest to the right of Figure 4.5.

### 4.2.2 Theoretical evaluation

This section exhibits simple, realistic worst case examples for the scheduling heuristics. Many of these examples occur in the results in Section 4.4. All the examples are one-dimensional “streaming” programs, constructed by a single for loop containing calls to several *kernels*. Each such call generates a task. The multiplicity, or number of tasks per kernel per iteration, varies between kernels.

These streaming examples divide into two groups, which correspond to the two main ways that the schedule can be bad. The first group suffers from *exhaustion*, meaning that the scheduler frequently runs out of parallelism throughout the execution. This happens when the scheduler tends to complete all the kernels from one loop iteration before starting the next iteration. This is typical of *oldest*, and it is inappropriate when the next iteration begins with a long kernel, on which all other kernels depend.

The second group suffers from poor *back pressure*, meaning that some kernels execute the loop iterations at a faster rate than others. Poor back pressure implies poor temporal data locality, which, if the amount of data transferred between kernels is large, will reduce


 Figure 4.6: Worst case examples for exhaustion, illustrated for  $p = 3$  processors

performance significantly. If the amount of data transferred is small, when all iterations of the faster kernels have been done, there may not be enough parallelism left to keep all of the processors busy. Back pressure can be helped by reducing the window size, but it still uses more memory than it should; exhaustion cannot.

Table 4.3 shows the worst case throughput, as a multiple of the optimal throughput. These figures assume that the number of iterations are large, so they exclude the overhead at the start and end of the program. All examples have an optimal schedule with all processors 100% busy during the steady state.

Many examples have worst case ratio close to  $\frac{p}{2p-1}$ , so for large numbers of processors, on average they use only half of the number of processors as the optimal schedule. Graham's bound [Gra71] shows that any demand scheduler; i.e. one using priorities or a list schedule, has execution time at most  $\frac{2p-1}{p}$  times optimal; so this bound is indeed the worst case.

These examples are synchronous dataflow (SDF) [LM87] streaming programs, so they could also be scheduled statically, at compile time. Table 4.3 shows the worst case for *optimal* static schedulers with and without support for *kernel fission*, which splits stateless kernels (a kernel is stateless if there is no dependency path from one iteration of the kernel to another).

If the static scheduler does *not* support fission, the worst case is when most of the work is in a single stateless kernel, since the static scheduler allocates that kernel to one of the processors, while any dynamic scheduler would run it on them all. *With* kernel fission, the worst case is a pipeline of  $p + 1$  stateful kernels. One processor must be assigned two of them, giving utilisation  $\frac{p+1}{2p}$ , which approaches one half for large  $p$ .

## Exhaustion

Figure 4.6(a) is an example where the **oldest** policy suffers from exhaustion. The iterations of the loop run from left to the right. Each row is a kernel, all of whose tasks have the same execution time, shown to the left of the diagram. The vertex labels, for this one diagram, number the tasks in the order they were *created*. The other diagrams will highlight whichever metric is used by the scheduler. The diagram is illustrated for  $p = 3$  processors.

An optimal schedule allocates one processor to the top row, at cost 1 per iteration. The remaining  $p - 1$  processors are each allocated to one row of cost  $\frac{p-1}{p}$  and one row of cost  $\frac{1}{p}$ , for a total cost per processor also equal to 1 per iteration. The schedule is a pipeline of length two, and the loads are balanced, so steady state utilisation is 100%.

#### 4. RUN-TIME DECISIONS

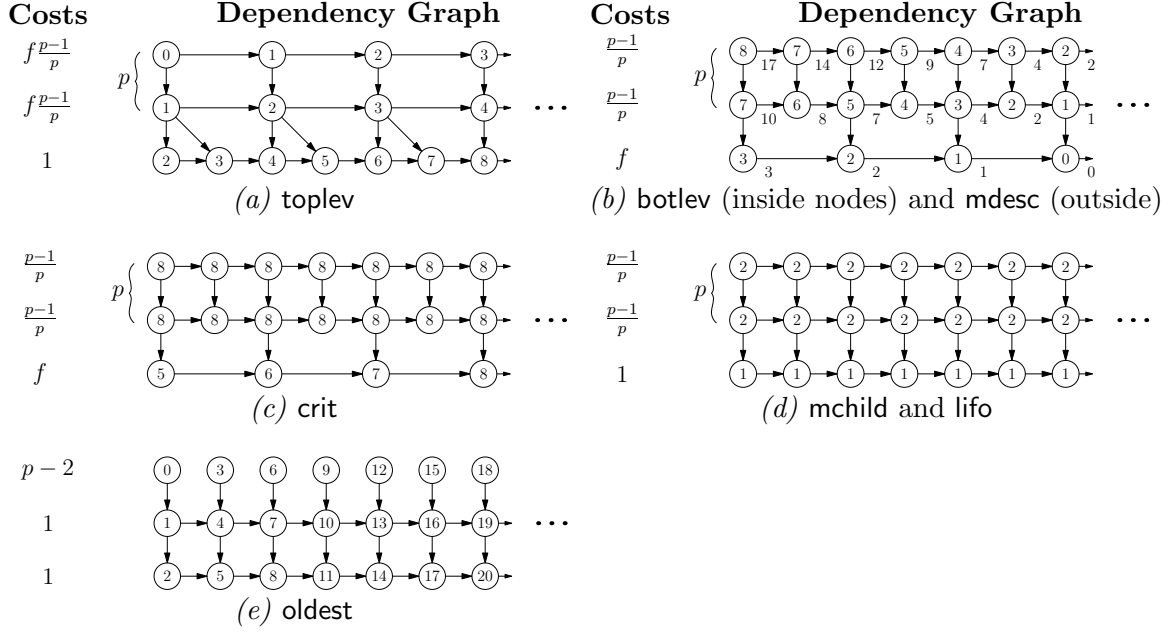


Figure 4.7: Worst case examples for back pressure, illustrated for  $p = 2$  processors ( $p = 3$  for subfigure (e)) and unroll factor  $f = 2$

The **oldest** scheduler will first issue task 0 from the top row. Once that has completed, all other tasks in the same iteration (1, 2, 3, and 4 in the picture) become ready. The next task from the top row, 5, also becomes ready, but it will not be issued until the tasks from the same iteration have been done. The kernel execution times have been chosen so that all processors finish the first iteration at the same time. At this point, the first task, 5, in the next iteration can start on one processor, but the remaining processors will be idle.<sup>1</sup> Each iteration requires time 1 for the task from the top row, plus time  $\frac{p-1}{p}$  for the remaining tasks; hence the result in Table 4.3.

This example is a perfectly ordinary stream program, and several StreamIt benchmarks suffer a similar fate. The **fm** benchmark illustrated in is a good example.

This example can be adapted for some of the other heuristics. It works as it stands for (**fifo**, **oldest**) and (**lifo**, **oldest**), because in the above description, all tasks in the ready queue became ready at the same time. To produce a general example for **fifo** and **toplev**, add a zero-cost task between each consecutive pair of tasks in the top row; i.e. between tasks 0 and 5, and 5 and 10; and so on.

Figure 4.6(b) shows a similar example for **botlev** and **crit**. The labels inside the tasks are the bottom level, and the labels outside are the criticality. The analysis is the same as before, and this example also turns up in the empirical analysis. Similar examples exist for **mchild** and **mdesc**. A possibility for the latter is to unroll all but the first kernel and make the branches of the split stateful; this is one of many ways to ensure that the branches of the split have more descendants than the next task in the top row.

<sup>1</sup>An example execution trace for this example is shown in Figure 4.9(b).

### Back pressure

Figure 4.7(a) is an example where **toplev** suffers from poor back pressure. Each task is labelled with the top level. There are  $p + 1$  kernels, the first  $p$  of them with multiplicity one, and the last with multiplicity  $f$ . The diagram is illustrated for  $p = 2$  processors, and  $f = 2$ . The first  $p$  rows have tasks of cost  $f(p - 1)/p$ , after unrolling, and the final row has tasks of cost 1.

An optimal dynamic schedule uses  $p - 1$  processors for the top  $p$  rows, and the remaining processor for the bottom row. The bottom row has cost 1 per iteration of the original loop. The tasks in the top  $p$  rows each have cost  $f(p - 1)/p$ , but they each cover  $f$  iterations of the original loop. Hence the total cost for  $f$  iterations of the top  $p$  rows is  $f(p - 1)$ , which is also 1 per processor per original iteration. Ignoring the first few and last few iterations, the load is balanced perfectly, and can be scheduled as a pipeline, so utilisation is 100%.

The **toplev** scheduler proceeds by top level, with 100% utilisation until the last task in the top row has been completed. At that point, the top  $p$  rows are all close to complete, but the bottom row is about  $1/f$  complete. The rest of the bottom row will be serialised.

For  $n$  complete iterations of the original loop, the total busy CPU time, for either scheduler, is  $np$ . The **toplev** scheduler has idle time equal to

$$t = (p - 1)n \left( 1 - \frac{1}{f} \right),$$

since  $p - 1$  processors wait for the completion of the bottom row.

The ratio of total time is therefore

$$\begin{aligned} \frac{np + t}{np} &= \frac{p + (p - 1)(1 - \frac{1}{f})}{p} \\ &= \frac{2p - 1 - f^*}{p}, \quad \text{where } f^* = \frac{p - 1}{f}. \end{aligned}$$

For large unroll factors, the relative utilisation is close to  $\frac{p}{2p-1}$ , itself close to one half for large numbers of processors.

Figure 4.7(b) is a similar example for **botlev** and **mdesc**. The labels inside the tasks are the bottom level, and the labels outside are the number of descendants. The analysis is similar to the previous case, leaving much of the bottom row to be processed sequentially on one processor. For **botlev**, the bottom row will not be started until only  $\frac{1}{f}$  of the top row is left. At this point, the  $p + 1$  rows will be processed together, until just the bottom row is left. For **mdesc**, the bottom row will begin execution a little later, but the analysis is similar.

Figures 4.7(c) and (d) show worst-case examples for **crit**, and **mchild** and **lifo**. The labels inside the tasks are the priorities, given by the criticality and number of children, respectively. In both cases, none of the tasks from the bottom row will be issued until the top row is finished. For **crit**, this is true whatever the unroll factor, because removing just one task from the bottom row moves the bottom row off the critical path. For **mchild**, this is true because the tasks in the bottom row have one child, whereas all the other tasks have two.

For **lifo**, the first or second task in the bottom row will be delayed until the end of the execution, because it will always have been the first task put in the ready queue that has still to be done.



#### 4. RUN-TIME DECISIONS

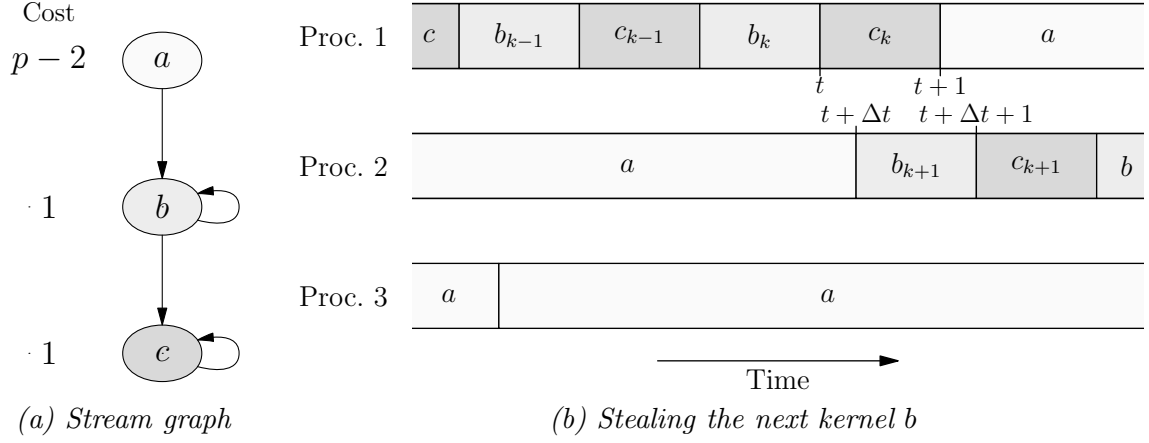


Figure 4.8: Illustration for oldest back pressure

For *fifo*, the utilisation under exhaustion is slightly higher, at  $\frac{2p}{3p-1}$ . An example is a stateful pipeline, similar to Figure 4.7(d), but with  $p(p-1)$  kernels of cost 1, followed by a single high-cost kernel of cost  $p+\epsilon$ , for small  $\epsilon$ . An optimal schedule has a perfectly balanced pipeline, processing one iteration every  $p$  time units.

The *fifo* schedule, however, runs the high cost kernel at one half the rate of the other kernels. The remaining iterations of this kernel will have to be processed on one processor. This is because while a high-cost task is executing, a complete wave of other tasks can complete: in time  $p$  on  $p-1$  remaining processors,  $p(p-1)$  tasks are completed. When the high-cost task has completed, there is a whole new wave of other tasks in the FIFO queue ahead of it, so a second wave completes before the next high-cost task starts. The utilisation, in Table 4.3, is like *topev* with  $f=2$ .

Surprisingly perhaps, the **oldest** scheduler can also suffer from poor back pressure. This is possible, even though it always assigns highest priority to the oldest task. An example is shown in Figure 4.7(e). This example is redrawn in Figure 4.8(a) as a stream graph, with the kernels labeled  $a$ ,  $b$ , and  $c$ .

The costs are chosen so that the optimal schedule should use, on average, one processor for kernel  $b$ , one processor for  $c$ , and the rest of the processors for kernel  $a$ .

In fact, the **oldest** scheduler often uses one processor for both  $b$  and  $c$ , and  $p-1$  instead of  $p-2$  processors for the rest, causing the back pressure problem. Figure 4.8(b) shows why. When processor 3 finishes a task, processor 1 is currently executing an iteration of kernel  $b$ , and no iterations of either kernel  $b$  or  $c$  are ready. It therefore has to execute another iteration of kernel  $a$ . When processor 2 finishes its task, processor 1 is currently executing kernel  $c$ , and the next iteration of kernel  $b$  is ready. This allows processor 2 to begin executing this kernel, opening up parallelism. However, when processor 1 finishes, the only ready tasks are again those from kernel  $a$ . The overlap is therefore short-lived, and only in exceptional runs are there a significant amount of concurrency between kernels  $b$  and  $c$ .



## 4.3 Adaptive schedulers

This section describes two adaptive schedulers for stream-like programs. The first, *adaptive priority* (apriority), is specific to one-dimensional stream programs. The second, *general adaptive priority* (gpriority), has slightly more overhead but it is more general.

Both policies are low overhead, and are based on the oldest-first scheduler. As will be seen in Section 4.4, oldest-first has the lowest memory use, and it usually has the best performance, but it sometimes suffers from exhaustion. The adaptive schedulers modify the oldest first scheduler so that if exhaustion occurs, the priorities are adjusted to try to stop it happening again. If exhaustion never happens, both policies remain the same as *oldest* throughout execution, retaining its low memory use.

The *oldest* scheduler issues the task with the smallest task number; that is, the priority is zero minus the task number. Both the adaptive schedulers modify this priority by adding an *adjustment*, which depends only on the kernel ID. The adjustments start at zero. Whenever a kernel is seen to be a bottleneck, the adjustments of that kernel and its ancestors are increased.

Both adaptive schedulers have complexity similar to that of oldest-first. They hold the ready tasks in a priority queue, which has complexity per task equal to the average value of the logarithm of the number of ready tasks. Ready tasks are inserted with known priority. The schedulers maintain a small table of metrics, described later, which is updated in constant time per task. The priority adjustments are updated infrequently, and the cost is linear in the number of kernels. In addition, the *gpriority* scheduler has some overhead at task creation time, which has a constant cost per dependency edge.

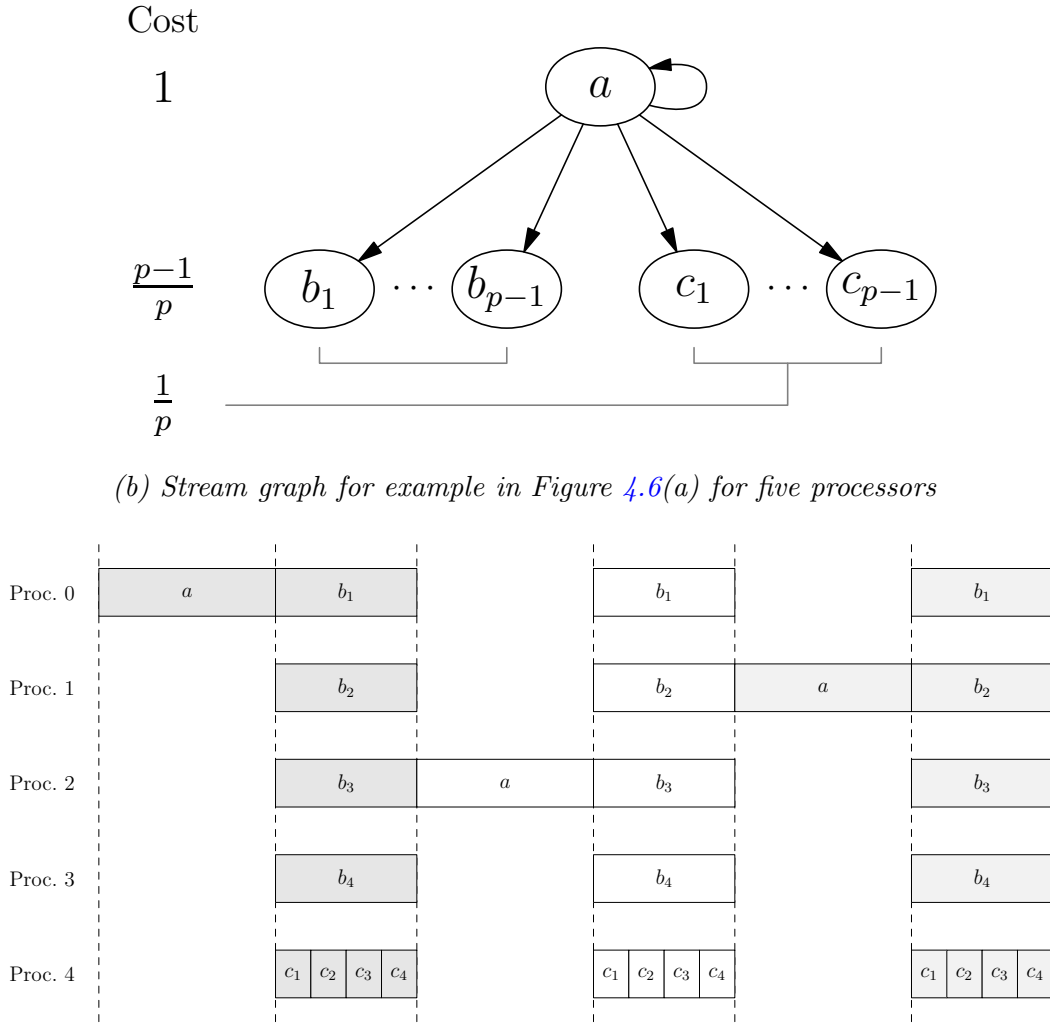
### 4.3.1 Intuition

Figure 4.9 shows the worst-case example for oldest-first, from Figure 4.6(a), redrawn as a stream graph, with the kernels given labels  $a$ ,  $b_1$ , and so on. An execution trace is shown in Figure 4.9(b). For comparison, an optimal execution trace is shown in Figure 4.9(c). As described in Section 4.2.2, this example suffers from exhaustion: processor utilisation is poor because the oldest-first scheduler finishes the tasks from one iteration before starting the next. The next iteration starts with a single expensive kernel  $a$ , and all other processors have to wait for it to finish.

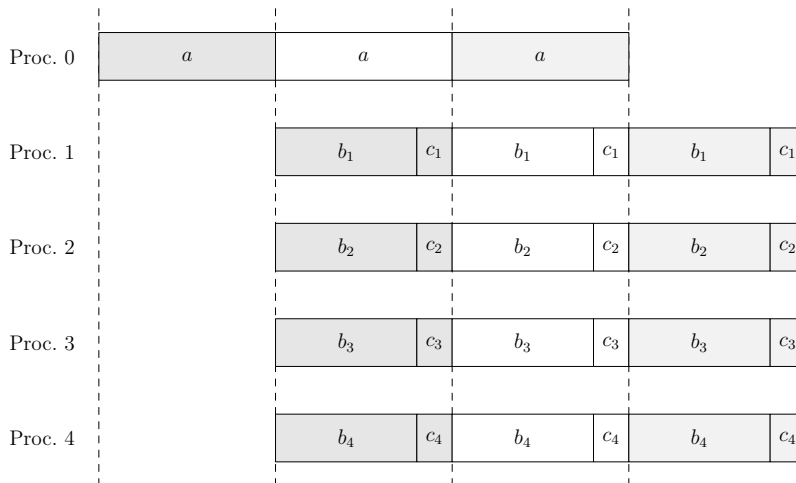
The problem is apparent in the trace—only one processor is busy just before kernel  $a$  finishes. To avoid this problem, each firing of kernel  $a$  should have started executing earlier. In contrast, every time kernel  $b_1$  finishes executing, roughly half of the processors are busy. The precise number depends on the order in which  $b_1$  through  $b_4$  and  $c_4$  finish, and on the latency in the run-time system between the time a task finishes and the thread is marked idle. Figure 4.10 is an example set of statistics after three iterations. It gives, for each kernel, the number of times that kernel has been done and the average number of threads that were busy just before it finished.

The average number of busy threads for kernel  $a$  is low, so  $a$  is probably the bottleneck. The problem can be alleviated or fixed by increasing kernel  $a$ 's priority. The following sections extend this idea into a working scheduler.

#### 4. RUN-TIME DECISIONS



(b) Example execution trace for five processors (shading distinguishes iterations)



(c) Optimal execution trace for five processors (shading distinguishes iterations)

Figure 4.9: Motivation for busyness statistic

Kernel	Times completed	Average busy threads
$a$	3	1.0 ←
$b_1$	3	2.3
$b_2$	3	4.3
$b_3$	3	4.3
$b_4$	3	3.7
$c_1$	3	5.0
$c_2$	3	5.0
$c_3$	3	5.0
$c_4$	3	3.0

Figure 4.10: Example statistics after three iterations of the program in Figure 4.9

Name	Notation	Initial value	Description
<i>For every thread</i>			
State	$S_j$	not started	State: not started, busy, or idle
<i>For every kernel</i>			
Adjustment	$a_k$	0	Priority adjustment for this kernel
Sum busy threads	$t_k$	0	Used to calculate average busy threads
Starved count	$s_k$	0	Number of times kernel was <i>starved</i>
Non-starved count	$c_k$	0	Number of times kernel has completed without being <i>starved</i>

Table 4.4: Statistics for adaptive schedulers

### 4.3.2 Monitoring

#### Updating statistics

Table 4.4 lists the small set of statistics that are maintained by the `apriority` and `gpriority` schedulers. Each worker thread starts in state `not started`, in which it remains, until the scheduler issues the first task to run on it. In order to avoid discriminating against kernels near the top of the stream graph during the first few iterations, processors are also considered busy if they have not executed any tasks yet. The scheduler keeps track of the number of busy threads, which is the number of threads in the `not started` or `busy` state.

Every time a task completes, the adaptive scheduler needs to update the corresponding kernel's statistics. This must be done before any successors of the task can become ready. If the next task for the same kernel has not yet been created, the kernel is said to be *starved*, and the master thread could be the bottleneck. If this happens too often, there would probably be little point in increasing that kernel's priority anyway. If the kernel is starved and there is at least one idle thread, the starved count for this kernel is increased. Otherwise, the kernel's non-starved count is increased by one, and the sum of busy threads is increased by the current number of busy threads.

## 4. RUN-TIME DECISIONS

---

### 4.3.3 Updating priorities for **apriority**

#### Update algorithm

The update algorithm for **apriority** is called every so often to modify the priority adjustments. It happens on a worker thread at the end of **complete-task**. More details on the update interval are given below.

The update algorithm works as follows. First, if more than 10% of tasks were starved; that is, if  $\sum_k s_k \geq \frac{1}{10} \sum_k c_k$ , the main thread is probably the bottleneck, and there is little the adaptive scheduler can do, except try to minimise overhead. In this case, the priority adjustments are left unchanged, and the interval to the next time the update algorithm is called is increased. The experiments in Section 4.4 simply use an interval of 100ms for the normal case and 500ms if more than 10% of tasks were starved. This behaviour could be made more sophisticated.

If fewer than 10% of tasks were starved, a *candidate* kernel is identified. The candidate has the lowest average number of busy threads. In case of ties, it is the kernel with the largest ID, which is the one nearest the bottom of the stream graph. The average number of busy threads for the candidate is compared with the average for all kernels. If it is less than 90% of the average for all kernels, then the candidate is treated as a the *bottleneck* kernel; otherwise, there is no clear bottleneck kernel.

The priority of the bottleneck kernel is increased by the number of kernels per iteration. In order to avoid priority inversion, if any of the candidate's predecessors now have a lower priority than it, their priorities are increased to be the same. This procedure applies to all ancestors of the candidate. After updating any kernel adjustments, all kernels have their statistics reset:  $t_k := 0$ ,  $s_k := 0$ , and  $c_k := 0$  for all  $k$ .

For the example in Figure 4.9, the candidate is kernel  $a$ , its average number of busy threads is 1.0, which is less than 90% of the average since the average is 3.7. Its priority adjustment is increased to eight, which correctly pipelines the program. All statistics are reset, no other priority adjustments are increased, and the program remains correctly pipelined for the rest of the execution.

It may seem sufficient, after updating a kernel's adjustment, to only reset the statistics for either that kernel or that kernel and its ancestors. Figure 4.11 is an counterexample to such a claim. The bottleneck is kernel  $a$ , exactly as before. After correctly increasing the priority adjustment for kernel  $a$ , kernel  $d$  will next appear to be the bottleneck. Unfortunately, updating the priority adjustment for kernel  $d$  makes all priority adjustments the same, which has the same effect as if they were all zero.

#### Updating ancestors

Figure 4.12 is an example that illustrates why the adaptive algorithms need to update the ancestors of the indicated kernel. Subfigure (a) is the stream graph. All kernels have multiplicity one, and cost shown outside the vertex, where  $p$  is the number of processors and  $\epsilon \ll 1$  is some small number much less than one. Since the only stateful kernel is the source,  $a$ , the critical path has cost 1 per iteration.

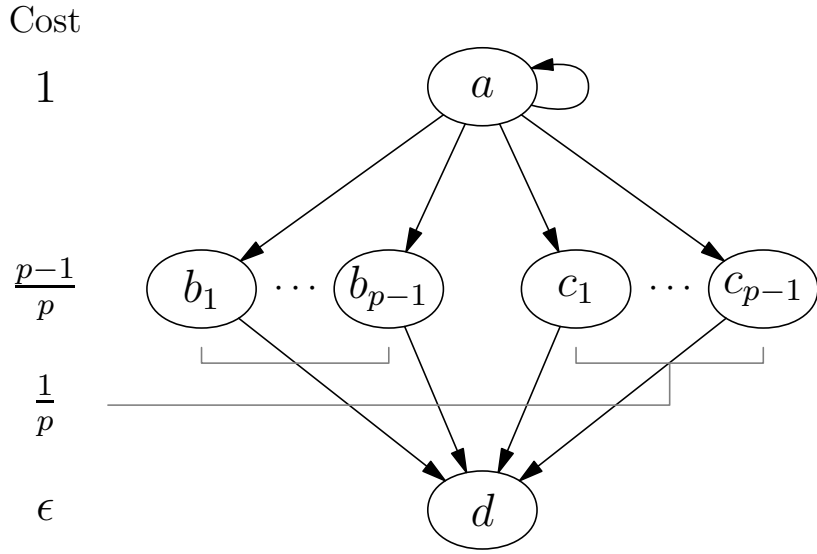


Figure 4.11: Example showing why all kernel statistics should be reset

The total work per iteration is

$$\begin{aligned}
 & 1 + \frac{2p-2}{p} + \epsilon + (p-1)\frac{p-2}{p} + 2(p-1)\epsilon \\
 &= p + (2p-1)\epsilon \\
 &\approx p,
 \end{aligned}$$

so an optimal schedule on  $p$  processors requires time 1 per iteration.

Subfigure (b) is an example trace of two iterations, using the oldest-first scheduler. This requires time  $\frac{2p-1}{p} + \frac{\epsilon}{2}$  per iteration.

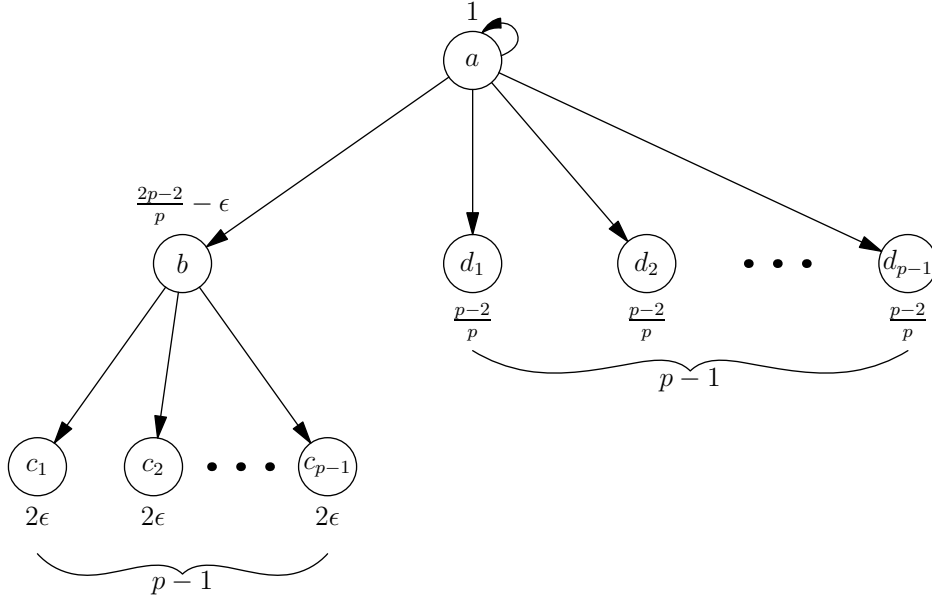
Subfigure (c) shows the busy statistics for the **apriority** scheduler, which is, for each kernel, the average number of busy processors just as it finishes, but before any of its predecessors wake up. The startup mechanism means that even on the first iteration, kernel  $a$  registers  $p$  busy processors. The results for  $b$  and  $c$  are clear. The value for kernel  $d$  is an average: the first kernel  $d$  that finishes sees  $p$  busy processors, the second sees  $p-1$  busy processors, and the last sees two busy processors. The average can be found by pairing off in pairs, starting with the first and last, and each pair has average  $\frac{p+2}{2}$  busy processors.

The **apriority** and **gpriority** schedulers will therefore identify kernel  $b$  as the critical path, and increase its priority. Unfortunately kernel  $b$  already always executes immediately after kernel  $a$  in the same iteration, so increasing its priority makes no difference—it is necessary also to increase the priority of kernel  $a$ .

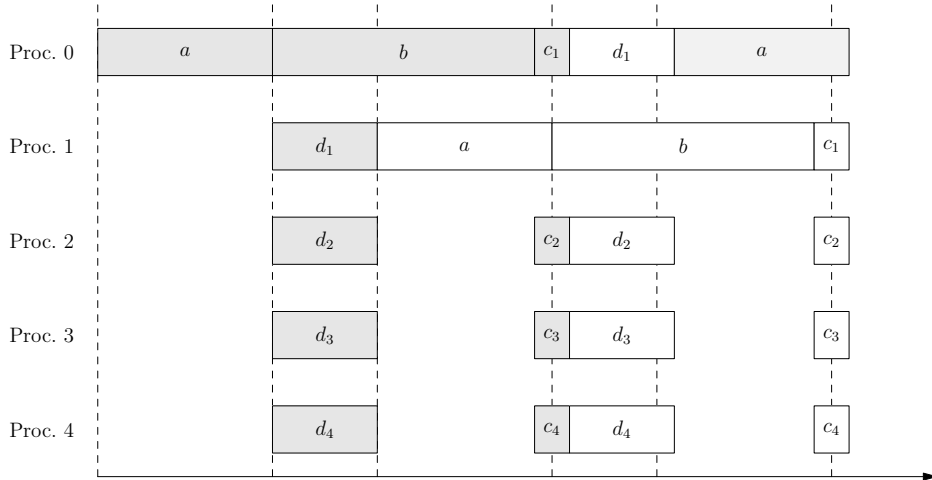
#### 4.3.4 Updating priorities for **gpriority**

The **gpriority** algorithm differs from **apriority** in the following ways. First, some decision has to be made about what constitutes a *kernel*. Second, the dependency graph between kernels is constructed at run-time, rather than being provided somehow by the compiler. Third,

#### 4. RUN-TIME DECISIONS



(a) Stream graph



(b) Example trace

Kernel	Average Busy threads
$a$	$p$
$b$	$2 \leftarrow$
$c$	$p$
$d$	$\geq \frac{p+2}{2}$

(c) Statistics for apriority

Figure 4.12: Example that shows the benefit from updating the ancestors, where  $p$  is the number of processors

this kernel dependency graph may be cyclic. Fourth, there is no concept of a steady-state iteration.

A reasonable definition of a kernel is the source line of the *caller* of the StarSs task. For a stream program, converted using `str2oss` or a similar tool, this definition is consistent with `apriority`. When the same kernel definition is instantiated several times in a stream graph, each instantiation is treated as a different kernel. Figures 5.15 and 5.16 show the translated code for a stream program: lines 95 and 96 correspond to two different kernels with the same function definition.

To construct the kernel dependency graph, in `create-task`, the scheduler has to map from the task to the kernel. This can be done either by explicitly adding tags in the compiler, or hashing the address of the work function. The latter works when the work function is actually a wrapper, which is the case using the Mercurium source to source compiler for OmpSs.

Figure 4.13 shows an example kernel dependency graph, constructed by `gpriority` for 25 frames of HD 1080p video through the H.264 decoder skeleton. There are four kernels, corresponding to the four stages in this H.264 decoder skeleton. This benchmark has streaming behaviour, but it is not one dimensional. Macroblock decode and Deblock iterate over a two-dimensional space.

To the left of each of kernel is shown the number of times it was called; e.g. `Scan` was called 25 times: once per frame. Each edge shows the number of times,  $n$ , that that dependency was seen in the PDG, and the *distance*, which is the average difference,  $D$ , in task number. For example, assuming that the tile width is two or more, macroblock decode usually depends on three of its neighbours: the tile to the left, with difference 1, the tile above, with difference 31, and the tile above and to the right, with difference 30.<sup>1</sup> The average of these values is 20.7, which is close to the value that was measured. The measured distance is a little different, because some of these dependencies are missing at the image borders.

The priority of a task is equal to its kernel's priority adjustment minus the task number, as for `apriority`. The priority adjustments for the kernels control the overlapping of kernels. Each kernel, however, is effectively scheduled using oldest-first, since all the tasks for the same kernel have the same adjustment. The `gpriority` scheduler uses the same statistics, shown in Figure 4.4, as `apriority`. Similarly, the bottleneck kernel is found in exactly the same manner as before.

Since there is no concept of steady-state iteration, the priority adjustment for the bottleneck kernel cannot be increased by the number of tasks per iteration. Instead, each kernel has a *delta* value, which is initialised to some small number. Whenever that kernel has its priority adjustment increased, it is increased by the delta, and the delta is doubled. This mechanism is rather crude, but it seems to work.

Whenever a kernel is identified as a bottleneck and its priority adjustment is increased, the adjustments of its ancestors are updated to avoid priority inversion. For example, if the adjustment of `Entropy Decode` is increased from zero to  $\Delta$ , the adjustment of `Scan` would be increased from zero to  $\Delta - 1$ . Unlike for `apriority`, this procedure takes account of the average difference in task number.

---

<sup>1</sup>These figures are for an image size of  $1920 \times 1080$  and a skewed tile of size  $4 \times 4$ . Per row, there are 29 complete tiles and two incomplete tiles.

## 4. RUN-TIME DECISIONS

---

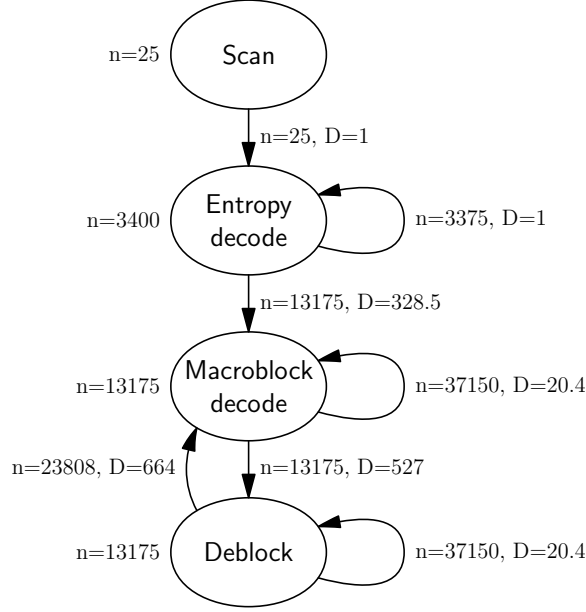


Figure 4.13: Kernel dependency graph for H.264 decoder skeleton

After increasing the priority adjustment for kernel  $k$  from  $a_k$  to  $a'_k = a_k + \Delta$ , each of that kernels predecessors,  $p$ , are visited in turn. The predecessor's priority adjustment is updated:

$$a'_p = \max(a_p, a_k + \Delta - D_{pk}) \quad \text{where } D_{pk} \text{ is the distance of the edge from } p \text{ to } k.$$

The intuition is that priority inversion happens when a task has higher priority than a predecessor. The difference in priorities of tasks is equal to the difference in priority adjustment minus the difference in task number.

## 4.4 Experimental evaluation

### Infrastructure and benchmarks

This section describes the experimental results for the schedulers described in Section 4.2.1, together with the adaptive schedulers described in Section 4.3. The implementation and results use OMP Superscalar (OmpSs), but they are not specific to it. The OmpSs compiler generates an executable that uses the Nanos++ runtime library.

The benchmarks are listed in Table 4.5. The StreamIt 2.1.1 benchmarks [GTA06] are one-dimensional stream programs written in the StreamIt language. They were converted to StarSs source code using a conversion tool, known as `str2oss`, described in Section 5.3.

At the time of writing, Nanos++ does not yet implement renaming, described in Section 1.4.1. If each task for a given filter were to write its output into the same temporary array, there would be output and anti-dependencies between consecutive iterations. For this reason, a command-line option was added to `str2oss` to expand the sizes of the temporary



## 4.4 Experimental evaluation

Benchmark(s)	Source	Parameters or input file
StreamIt 2.1.1 benchmarks	MIT	Twelve benchmarks in StreamIt language
Check LU factorisation	SMPs 2.1	Block size: $32 \times 32$ , matrix size: $16 \times 16$ blocks
Cholesky factorisation	BSC	Block size: $32 \times 32$ , matrix size: $16 \times 16$ blocks
H.264 decoder skeleton, <i>supporting 3D wave</i>	Self	100 frames of Pedestrian (1080p) from HDVideoBench [ASRV07]

Table 4.5: Benchmarks used in the evaluation

arrays by a given factor,  $n$ , and use the segments in round-robin order. Each unwanted dependency therefore crosses  $n$  iterations rather than one.

The experimental results include measurements of the peak memory use. This is the largest amount of memory that would be used for renaming. Since Nanos++ does not actually do renaming, the memory use was modelled by the application.

There are four additional benchmarks written in StarSs. Check LU factorises a non-symmetrical matrix into the product of a lower-triangular and an upper-triangular matrix, without a permutation matrix:  $A = LU$ . It was taken as-is from the SMP Superscalar 2.1 distribution, including a correctness check at the end, which is the sparse matrix multiplication of  $L$  and  $U$ . Cholesky uses LAPACK and BLAS routines to perform a Cholesky decomposition:  $A = LL^T$ .

The H.264 decoder skeleton has the same pattern of dependencies as an H.264 decoder using the 3D wave optimisation [MAJ<sup>+</sup>09]. The entropy decoder was unrolled by 60 macroblocks, and the macroblock decode and deblocking tasks were unrolled by four in each direction, using the standard tile shape.

### Scalability

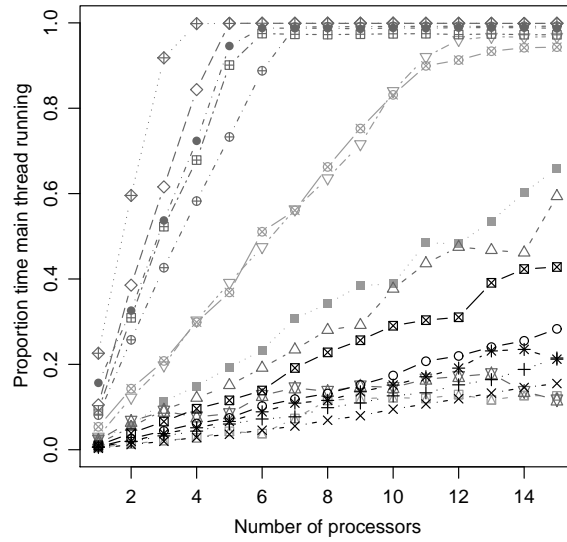
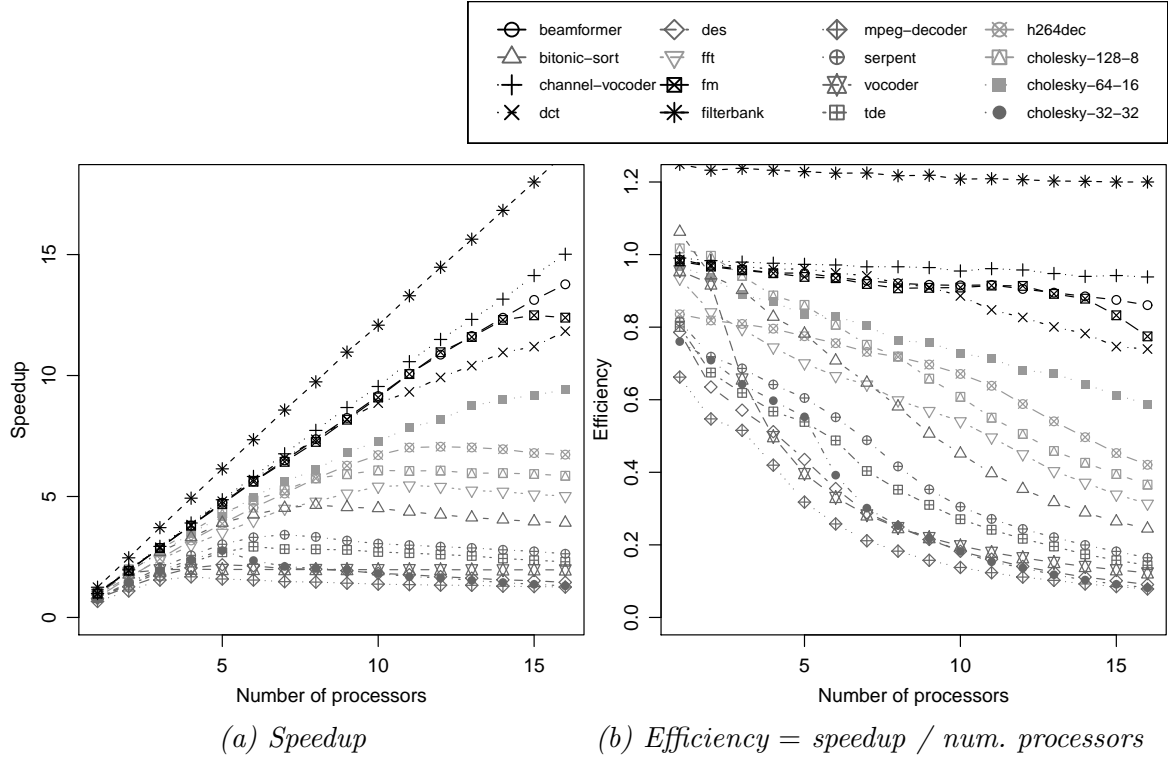
Figure 4.14 shows how well the benchmarks scale, as the number of processors varies between one and sixteen. The comparison is between the best dynamic scheduler and a serial implementation, compiled with GCC instead of the OmpSs compiler. The best scheduler is chosen independently for each data point, in order to estimate the scalability of the application itself.

It is clear that across the benchmarks there is a wide range of scalability: **filterbank** scales exceptionally well. Five benchmarks have 80% or higher efficiency to sixteen cores. Other benchmarks, especially **des**, have poor scalability. As can be seen from Figure 4.14(c), most of the benchmarks that scale poorly have the main thread running for a large proportion of time. This shows that for these benchmarks the main thread is the bottleneck.

The reason why some benchmarks have especially poor performance is that, although the benchmarks have been optimised using kernel unrolling, they have not benefited from kernel fusion. This is because the **str2oss** tool described in Section 5.3 supports unrolling but not fusion. This is evidence that a partitioning algorithm like that in Section 3.2 is still beneficial, even with dynamic scheduling.

Nevertheless, the wide range of scalability means that the results of this section are applicable both to benchmarks that scale well and benchmarks that scale poorly.

#### 4. RUN-TIME DECISIONS



(c) *Fraction of time main thread is running*

Figure 4.14: Scalability of the benchmarks

### Average and worst-case results

**Average** Figure 4.15(a) and (b) show, for each scheduler, the average utilisation and average peak memory use, where the average is taken across all streaming benchmarks. The non-streaming benchmarks were excluded, since they are not supported by *wave* and *apriority*. The memory use is the multiple of that required by the sequential version of the benchmark. It includes all data regions used by the benchmark and renamed by the run-time, but it excludes data structures used by the run-time itself; for example to store the PDG.

Three heuristics have consistently poor average performance, between 7% and 10% worse than *oldest* and the two adaptive policies. These are *botlev*, *crit*, and *mdesc*; all of which suffer due to the overhead of recalculating the bottom level or the number of descendants as the PDG is built.

The averaged peak memory footprint shows a wide range: *toplev* is particularly prone to poor back pressure, causing excessive average memory use. The *oldest*, *apriority* and *gpriority* schedulers have considerably lower memory use than the others, since these schedulers tend to overlap few stream program iterations.

**Robustness** Figure 4.15(c) and (d) compares the robustness of the schedulers. In Figure 4.15(c), each data point is the worst case ratio of efficiency, which is the efficiency of the scheduler, divided by the efficiency of the best scheduler, for that benchmark and number of processors. The worst case is found across the streaming benchmarks. A value of 1.0 on the y-axis therefore indicates that, for all benchmarks, the scheduler achieves the best performance seen. Figure 4.16 shows the same results, but magnified to show only the efficiency of the more robust schedulers.

Figure 4.15(d) is the worst case memory use, as a multiple of the sequential version. As before, the worst case is found across all streaming benchmarks.

As before, *botlev*, *crit* and *mdesc* are clearly the worst-performing schedulers, again due to the overhead of calculating the bottom level or number of descendants.

The only robust schedulers are the adaptive schedulers, *apriority* and *gpriority*. Of the non-adaptive schedulers, *oldest* is best performing overall, but its worst efficiency is up to 8% lower.

The worst case peak memory footprint shows a wider range: *toplev*, *fifo*, and *lifo* have excessive memory use for certain benchmarks. The remaining heuristics have intermediate memory use, which is considerably higher than that of *oldest*, *apriority*, and *gpriority*. Compared with *oldest*, the two adaptive schedulers use a negligible amount of additional memory.

### Detailed results

This section compares the efficiency and memory use of the six best schedulers. These are *fifo*, *oldest*, *toplev*, *wave*, and the two adaptive schedulers: *apriority* and *gpriority*. The other schedulers were seen to have poor robustness and average performance.

Figure 4.17 shows the experimental results for the StreamIt benchmarks. There are two plots for each scheduler. The first is the ratio of efficiency: the efficiency of that scheduler divided by the efficiency of the best scheduler for the same benchmark and number of pro-

#### 4. RUN-TIME DECISIONS

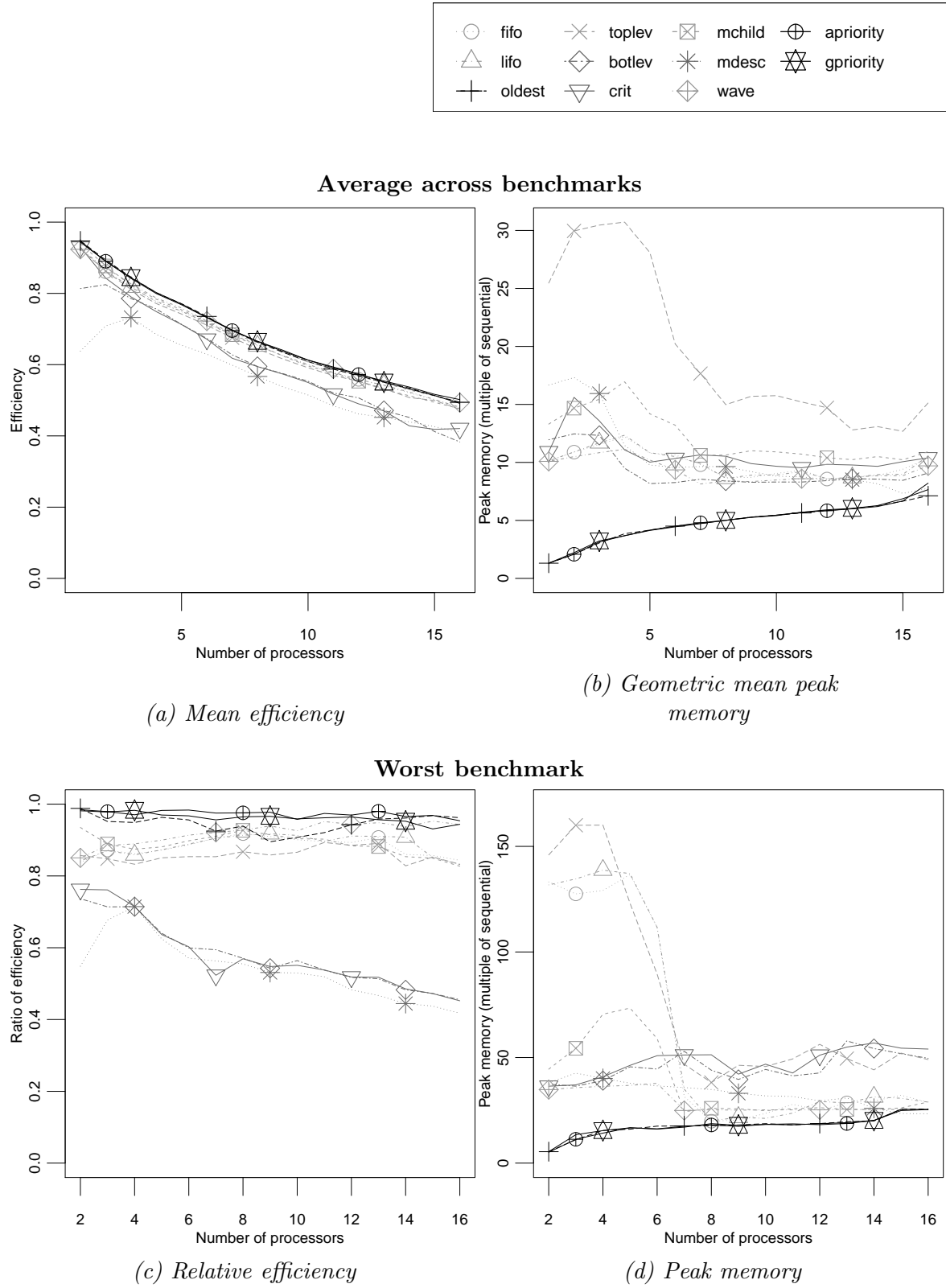


Figure 4.15: Average and worst case results

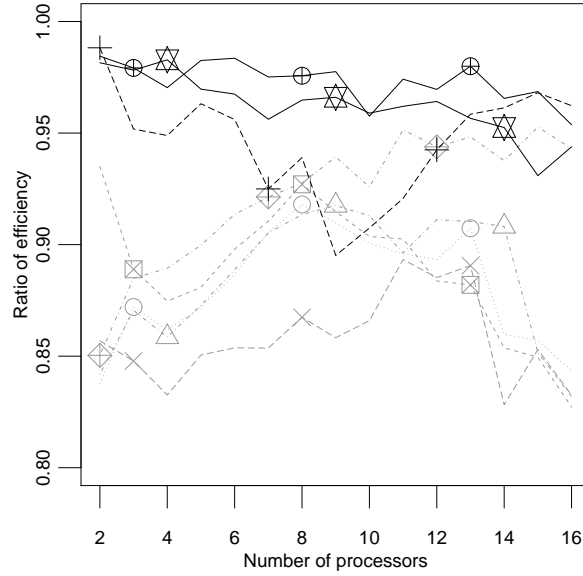


Figure 4.16: Worst case results: detail of efficiency of most robust schedulers

processors. The y axis has been expanded to show greater detail—no points were lost in doing so. The second plot is the peak memory use, as a multiple of the sequential version.

The **apriority** scheduler has the highest efficiency and lowest memory use. The **gprior** scheduler has slightly lower efficiency due to the greater overhead, and the same low memory use.

Figure 4.18 shows the experimental results for the non-StreamIt benchmarks. The **wave** and **apriority** schedulers were excluded, since they both require regular stream programs, so do not support these benchmarks. It was not possible to determine the memory footprint, because Nanos++ does not currently implement renaming. As described in Section 4.4, the renaming memory footprint was modelled by the application, and this was only implemented for the StreamIt benchmarks.

The **gprior** scheduler has the best efficiency and memory use overall.

## 4.5 Conclusions

This chapter introduced two new low-complexity adaptive dynamic scheduling algorithms for stream-like programs. Many existing scheduling algorithms, listed in Table 4.1, either schedule the whole program in advance, or they need to know the execution time of every task in advance, or they are too slow.

The scheduling algorithms in this chapter have better average and worst-case performance than all the scheduling algorithms in Table 4.2. The scheduling overhead at run-time is similar to that of oldest-first. They take advantage of the stream graph representation, by giving each kernel in the stream graph its own priority adjustment.

## 4. RUN-TIME DECISIONS

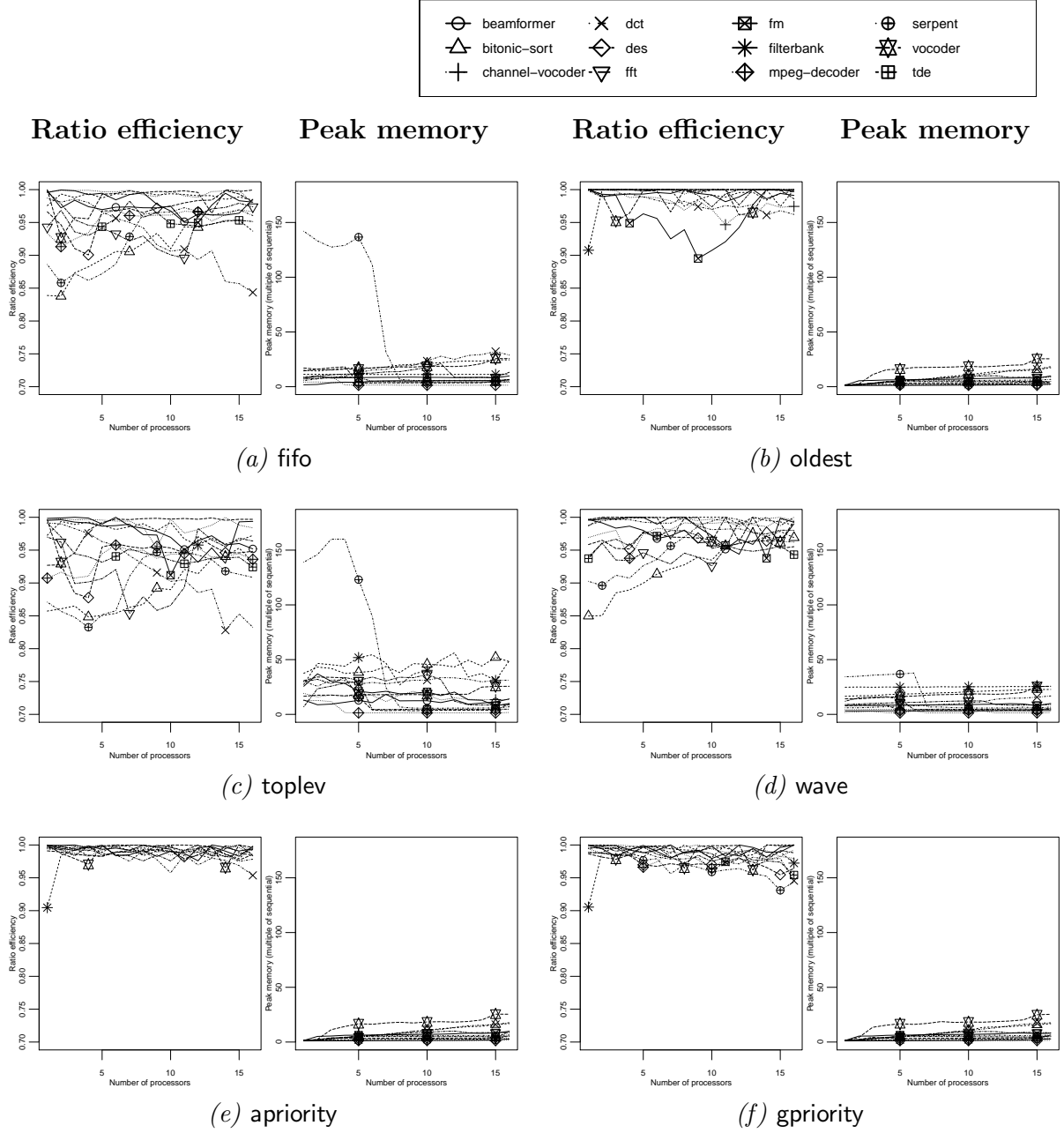


Figure 4.17: Comparison of the scheduling heuristics: StreamIt

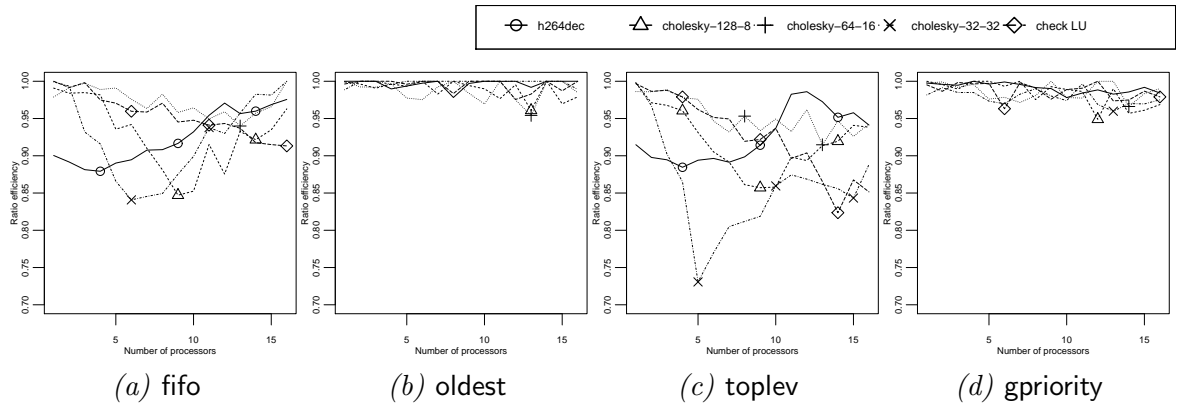


Figure 4.18: Comparison of the scheduling heuristics: non-StreamIt

#### 4. RUN-TIME DECISIONS

---



## Chapter 5

# Support Tools

This chapter describes the support tools that were developed in the course of this thesis. StarssCheck is a debugging tool, based on Valgrind [NS07], that finds bugs in StarSs programs. Paraver Animator (`prvanim`) is a simple tool that helps visualise the progress of a parallel application, by creating an animated GIF file. StreamIt to OmpSs (`str2oss`) is a tool to convert streaming programs in the StreamIt language into task based programs using StarSs.<sup>1</sup>

### 5.1 Debugging using StarssCheck

A programming language without debug tools may be a fine research vehicle, but it is unlikely to be widely adopted, as users become frustrated by bugs in their code, blame the compiler, and think that the language is hard to use. Moreover, parallel programs often harbour subtle intermittent bugs, which may not be noticed for months or years. If StarSs or OmpSs become mainstream, and if they are used for production code, then people will want as much evidence as possible that their code is free of hidden bugs.

This section describes StarssCheck, a debugging tool that finds bugs in StarSs programs. It was used, for example, to check the output of the StreamIt to StarSs converter described in Section 5.3. Similar ideas could be used in a tool supporting SPM. The tool was written for StarSs because StarSs is more mature, and it already has real users.

Section 1.4.1 explains StarSs in enough detail to understand StarssCheck. It suggested that the pragmas could sometimes have been written by the compiler instead of the programmer. StarssCheck is the converse. Although it is hard for the machine to work out the pragmas from scratch, it is relatively easy for the machine to check them.

StarssCheck checks that tasks only access memory that they are supposed to. A task can read any argument marked as `input`, but it mustn't change it. This could be enforced using `const` in C, assuming one never removes `const` using a cast. Adding `const` to existing code can be time-consuming, however, so is often avoided. A task can read or write any argument marked as `output` or `inout`, and modify the stack at or below its arguments. A task can always execute its code and read constant values. It mustn't access any other data

---

<sup>1</sup>Recall that StarSs is the language and OmpSs is the implementation—see Section 1.4.1

## 5. SUPPORT TOOLS

---

in memory<sup>1</sup>, something that is certainly hard for the compiler to check statically. If the task *does* try to read or write elsewhere in memory, it normally wouldn't be stopped from doing so; the program might seem to work most of the time, but have intermittent bugs—see Section 5.1.1.

StarssCheck uses an analysis tool that runs under Valgrind [NS07], a widely-used framework for binary instrumentation. The default Valgrind tool, *memcheck* [SN05], warns the user whenever the program's behaviour depends on invalid data; for example when a conditional branch depends on the contents of memory returned by `malloc`. StarssCheck is not specific to Valgrind. It requires a binary translation tool that has some mechanism similar to Valgrind's *client requests*, which are calls from the guest program into the analysis tool. The tool could, for example, have been written for Pin [LCM<sup>+</sup>05].

As mentioned in Section 1.4.1, a compiler that doesn't understand StarSs will ignore the pragmas, and compile a valid sequential program. StarssCheck runs the sequential version of the program under Valgrind, and checks that, for the supplied input, the pragma annotations are correct. StarssCheck can be adapted to any programming language for which a valid sequential program can be easily derived; e.g. by ignoring pragmas or keywords. It also requires the language to supply some restrictions on the regions of memory that can be accessed by a task; there is little point if a task can freely access shared memory.

### 5.1.1 Common StarSs errors

Figure 5.1 illustrates the main errors that can be found using StarssCheck. In Figure 5.1(a), the function accesses memory via a pointer embedded in a structure. This pointer, unlike the function arguments, will not have been tracked or renamed by the OmpSs runtime system. The read will not be seen as a dependency between tasks, and, on distributed memory, the data will not have been transferred via DMA. On CellSs, `p->ptr` is a pointer in the PPE's address space, but it is being dereferenced on an SPE.<sup>2</sup> Normal load and store operations on an SPE are not subject to memory protection, and addresses wrap modulo the size of the local store, so the function will read *some* value. The PPE and SPEs all use 32-bit pointers, which are not distinguished by the C type system.

In Figure 5.1(b), the function reads outside array `x`. This example also shows how StarSs supports variable length arrays: by specifying the length in the pragma rather than the C prototype. The tool must therefore handle array sizes that are not known until run time. Conversely, in Figure 5.1(c), if `isLong==0`, the array is declared larger than necessary, and neighbouring memory will be corrupted if the caller has allocated an array smaller than 20 elements.

Subfigure (d) is missing a `pragma wait` before the main program accesses array `x`. A wait is required even for write-after-read dependencies, because task creation returns immediately without taking a copy. The original contents of the array should remain unmodified until all tasks that read it have been allocated to SPEs, and outgoing DMAs have completed. All missing waits are race conditions, hence non-deterministic and notoriously hard to debug.

---

<sup>1</sup>In SMPSSs, the programmer can take advantage of shared memory, covertly sharing data through opaque pointers, and taking care of locking, memory consistency, and so on. StarssCheck could be extended to allow this, either by having the programmer tell StarssCheck what was going on, or by tracking opaque pointers.

<sup>2</sup>The pointer dereference could be made to work using the `_ea` attribute, but the dependency would still not be tracked by the OmpSs runtime system.

In Figure 5.1(e), the direction of data transfer is incorrect. This bug will often be easy to find because it is so blatant, but it can also be found by StarssCheck. In Figure 5.1(f), the author has not noticed that argument `x` can be `NULL`. While StarSs could be extended to allow `NULL` variable pointers, and pass them unmodified to the task, this is not the current behaviour. This example, and the example in Figure 5.1(c) may cause exceptions deep inside the run-time library, for which source may not be available.

StarssCheck finds all the mistakes in Figure 5.1. It is also a good place to check whether function arguments are correctly aligned for CellSs. This constraint is imposed by the Cell B.E. DMA engine [IBM09, §7.2.1].

### 5.1.2 How StarssCheck works

#### Overview

Figure 5.2 shows the structure of StarssCheck. A translation tool reads the StarSs annotations from the source code, generating a wrapper function for each task. It also translates the *finish* and *wait* pragmas into appropriate macros. Translation currently uses a Python script, but a more robust tool, based on Mercurium [BDG<sup>+</sup>04], would be better. The translated source code is compiled as normal, and executed under Valgrind, using the *Starssgrind* tool.

An alternative is to take an executable generated by the OmpSs compiler, and intercept all calls to the run-time library. The benefit would be that the program would not need recompilation, but only if you are using the same variant of OmpSs; and not, for example, CellSs (since Valgrind does not support the Cell B.E.). This approach would work, but it also would be specific to a certain version of the run-time API, which may change in future.

#### High-level interface to Starssgrind

StarssCheck uses Valgrind’s client request mechanism, which recognises a “magic” sequence of instructions in the binary. The instructions achieve nothing useful, unless the program is running under Valgrind, in which case they are recognised as a call into the analysis tool.

Table 5.1 lists the client request macros provided by Starssgrind. The precise semantics are given in Section 5.1.2, but the general idea is apparent in Figure 5.3, which shows the translated version of the *bmod* function (Figure 1.6).

Translation can be done using text substitution, without changing line numbers. Each task is given a wrapper function, which is a simple mechanism to define the region of stack it can touch, irrespective of the calling convention. The `PUSH_CONTEXT` macro enters a task. It sets up the task’s context, which initially allows access only to the `.text` section and the stack below the wrapper function’s frame pointer. The `INPUT_BLOCK`, `OUTPUT_BLOCK` and `INOUT_BLOCK` macros each declare a contiguous region of memory to be accessible by the task. The `POP_CONTEXT` macro leaves a task, and restores the master thread’s context. The master thread is subsequently allowed read-only access to the task’s input blocks, and no access to its output and inout blocks.

The `WAIT_ALL` and `WAIT_ON` macros support the *finish* and *wait on* pragma clauses, respectively. It is not necessary to translate the *start* pragma clause.

## 5. SUPPORT TOOLS

---

```
#pragma css task input(p) \
        output(y)
void a(struct t *p, int y[1])
{
    y[0] = *(p->ptr);
}
```

(a) Arbitrary memory access

```
#pragma css task output(y[n]) \
        input(x[n], n)
void b(int *y, int *x, int n)
{
    for(int k=0; k<n; k++)
        y[k] = x[k] + x[k+1]
}
```

(b) Array too small

```
int x[10];
c(x, 0);

#pragma css task output(x[20])
void c(int *x, int isLong)
{
    int len = isLong ? 20 : 10;
    for(int k=0; k<len; k++)
        x[k] = k;
}
```

(c) Output array too large

```
d(x, y);
// should wait here
// #pragma css wait on (x)
x[0] = 1;

#pragma css task input(x) output(y)
void d(int x[1], int y[1]) { ... }
```

(d) Missing wait

```
#pragma css task input(x,y) \
        output(z)
void e(int x[10],
        int y[10],
        int z[10])
{
    for(int k=0; k<10; k++)
        x[k] = y[k] + z[k];
}
```

(e) Incorrect transfer direction

```
#pragma css task output(x[10])
void f(int *x)
{
    if (x == NULL)
        return; /* do nothing */
    /* ... */
}
```

(f) NULL pointer

Figure 5.1: Example mistakes found by StarssCheck

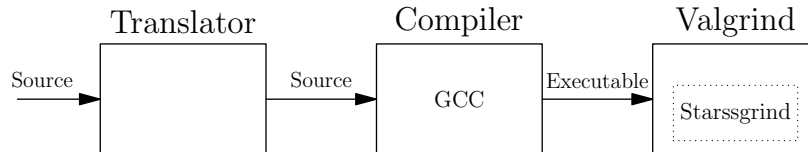


Figure 5.2: Structure of StarssCheck

```

1 __attribute__((__noinline__))
2 void css_wrapped_bmod(float row[32][32],
3                      float col[32][32],
4                      float inner[32][32])
5 {
6     int i, j, k;
7     PUSH_CONTEXT();
8     INPUT_BLOCK(row, sizeof(float[32][32]));
9     INPUT_BLOCK(col, sizeof(float[32][32]));
10    OUTPUT_BLOCK(inner, sizeof(float[32][32]));
11    for (i=0; i<32; i++)
12        for (j=0; j<32; j++)
13            for (k=0; k<32; k++)
14                inner[i][j] -= row[i][k]*col[k][j];
15    POP_CONTEXT();
16 }
17
18 __attribute__((__noinline__))
19 void bmod(float row[32][32],
20          float col[32][32],
21          float inner[32][32])
22 {
23     css_wrapped_bmod(row, col, inner);
24 }

```

Figure 5.3: Translated version of the *bmod* function from Figure 1.6

Table 5.1: StarssCheck client requests

Request	Description
PUSH_CONTEXT(void)	Enter task
INPUT_BLOCK(void *address, size_t len)	Declare input block
OUTPUT_BLOCK(void *address, size_t len)	Declare output block
INOUT_BLOCK(void *address, size_t len)	Declare inout block
POP_CONTEXT(void)	Return from task
WAIT_ON(void *address)	Restore given array
WAIT_ALL(void)	Restore all arrays

## 5. SUPPORT TOOLS

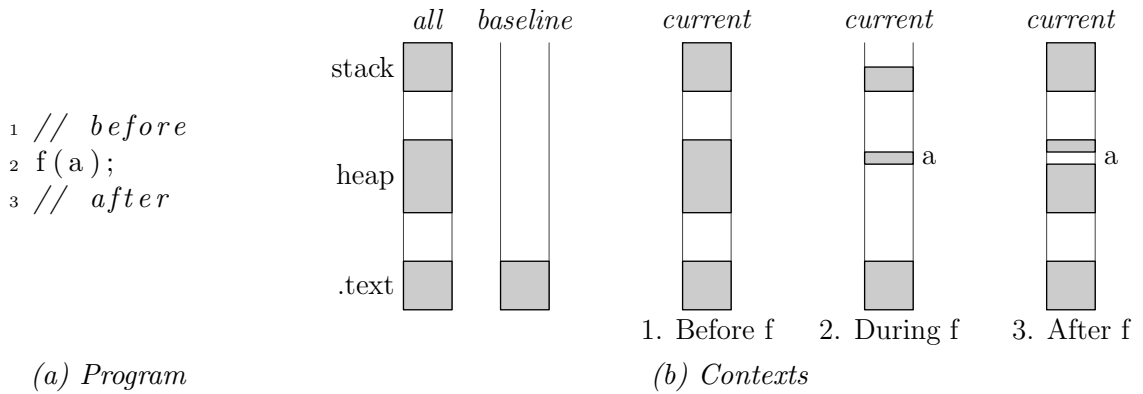


Figure 5.4: Starssgrind Contexts

### Starssgrind contexts

Starssgrind, the Valgrind tool, maintains several *contexts*, which define the accessible regions of memory. The *current context* is currently active: all accesses to memory are checked against it, and bad accesses immediately generate a warning. The *all context* defines the whole memory space accessible by the sequential program. It is the initial context for the main thread, and it defines the memory that the main thread can pass to tasks. The *baseline context* is the starting point for tasks, which contains only the `.text` section. Inside a task, the *parent context* defines the context of the main thread. When an array is passed to a task, it is removed or rendered read only in the parent context. Such regions are moved to the *dead context*, which records the sizes of arrays, so that when the main thread performs a wait, the correct region of memory can be restored.

A context is a disjoint set of *regions*, with each region covering a contiguous part of memory, with read-only or read-write access. The regions are stored in a balanced tree (our implementation uses a scapegoat tree [GR93]). Except within the dead context, adjacent regions with the same access rights get merged.

StarssCheck uses a tree representation rather than shadow bits for three main reasons. The first reason is that we do not expect the compiler to generate accesses outside the supplied arrays. This is different from validity checking in *memcheck*, where copying an array containing uninitialised padding should make the destination padding uninitialised, rather than immediately generating warnings. The second reason is that, for realistic programs, there are few active regions, so it is feasible to use a tree; often even a list is sufficient. The third reason is that extending *memcheck*'s efficient shadow bit representation to handle read-only regions and switching between contexts would be considerable work, with marginal benefit. See Section 5.1.3, however, for possible future work.

Figure 5.4 shows the current context before, during, and after task `f`. Assuming `f` is the first task created, the context before calling `f` is the same as the *all context*. During `f`, only the stack below its arguments, the `.text` section, and the argument `a` are visible. After calling `f`, `a` is not accessible until the main thread waits on `a`.

Since it is expensive to traverse the context tree for every memory access, we use a direct mapped region cache, based on the instruction address. The access is first checked against the region that the instruction previously hit. The region cache is cleared after any client request

```

1 t3:I32 t41:I32 t84:I32 t85:I1 t86:I32 t87:I32 t88:I1
2
3 IMark(0x26A2, 4)
4 PUT(60) = 0x26A2:I32
5 t3 = GET:I32(4)
6 t41 = GET:I32(168)
7 t84 = LDle:I32(0xF028E090:I32)
8 t85 = CmpLT32U(t3, t84)
9 DIRTY t85 ::: cs_helper[rp=3]{0xf0082850}(t3, 0x1:I32, 0x26A2:I32)
10 t86 = LDle:I32(0xF028E094:I32)
11 t87 = Add32(t3, 0x4:I32)
12 t88 = CmpLT32U(t86, t87)
13 DIRTY t88 ::: cs_helper[rp=3]{0xf0082850}(t3, 0x1:I32, 0x26A2:I32)
14 STle(t3) = t41
    
```

Figure 5.5: VEX IR for a single store instruction: `movss %xmm1, (%ecx)`. The address in the cache, for this particular instruction, is `0xF028E090`

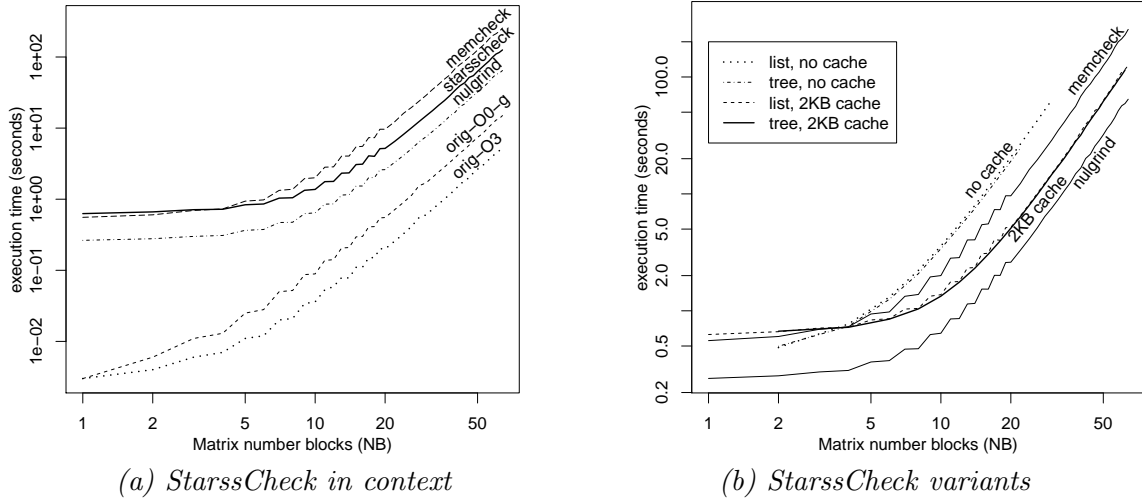


Figure 5.6: Performance results for Sparse LU factorisation

that switches the current context or deletes memory from it. The cache check is inserted directly into the VEX IR, which is the static single-assignment intermediate code used by Valgrind. Figure 5.5 shows the intermediate code generated by a single store instruction, with the non-analysis code in bold.

### 5.1.3 Evaluation

Most importantly, StarssCheck should find all the bugs within its remit, and it should not complain about code that is actually correct. In other words, it should have few false negatives (the former), and few false positives (the latter). StarssCheck finds all the mistakes in Figure 5.1, as confirmed by our test cases, and it only creates the false positives discussed in Section 5.1.3.

## 5. SUPPORT TOOLS

---

### Performance

A tool such as StarssCheck will only be used if it is reasonably fast. Our experiments show that unless tasks are so small that OmpSs itself is slow, the execution time under StarssCheck is similar to memcheck, which should be acceptable.

Figure 5.6 shows the execution times for the sparse LU factorisation example from the OmpSs distribution, using the default block size of  $32 \times 32$ . Figure 5.6(a) shows the execution time for square matrices up to  $64 \times 64$  blocks. For matrices of size about  $640 \times 640$  ( $20 \times 20$  blocks) or above, the slowdown of StarssCheck is close to 8 times compared with the original, unoptimised code. For comparison, memcheck is about 16 times slower than the original, and *nulgrind*, which is Valgrind without instrumentation, is about four times slower.

Subfigure (b) compares four variants of StarssCheck, with the region cache enabled and disabled, and using either a tree or a linked list to hold the set of regions. The difference between the tree and linked list is insignificant for this example, but the cache gives a speed-up of about 3.5, if the matrix size is 16 blocks or more. The mean number of regions is approximately 9, requiring an average search depth of 1.5. The average number of regions decreases slightly as the matrix increases, because a greater proportion of time is spent in tasks, which have a below average number of valid regions.

Figure 5.7(a) shows the “nasty” benchmark, which is intended to show worst case performance. This benchmark has extremely fine grain tasks: each task contains a single arithmetic statement. StarssCheck has high overhead because OmpSs itself has high overhead. The execution times are shown in Figure 5.7(c) and (d), and StarssCheck is much slower than memcheck. The region cache provides little benefit, because every access to the *a* array misses. The average search depth in the tree increases logarithmically in the number of tasks, which is the worst case.

The problem with the “nasty” benchmark is that the tasks are too small. Unrolling is a standard technique to increase task granularity. Figure 5.7(b) is a better implementation of the program, with the loops both unrolled by a factor of 1,024. This requires the *a* array to be reordered. Figure 5.7(e) shows the execution time of the modified benchmark, which is comparable to memcheck.

### Limitations

The main limitation of StarssCheck is that, unlike memcheck, it does not track the validity of data. Figure 5.8(a) shows a task with an output block that should be marked *inout*. There is in fact, inside StarssCheck, no difference between `OUTPUT_BLOCK` and `INOUT_BLOCK`, since both allow the task to read and write, and both make the block inaccessible to the main thread until it waits.

Fortunately, this error can be found using memcheck. It is advisable to run under memcheck in any case, because memcheck finds errors that are outside the scope of StarssCheck. StarssCheck could include a translator that invalidates memory corresponding to each output array, using the `VALGRIND_MAKE_MEM_UNDEFINED` memcheck client request. Clearly the functionality of memcheck and StarssCheck could be combined into a single tool.

The current implementation of StarssCheck fully supports CellSs, and a subset of SMPs. SMPs introduces *array region specifiers*, which describe a region in a multi-dimensional array, and *opaque pointers*, which are ignored by the run-time, and allow tasks to exploit



```

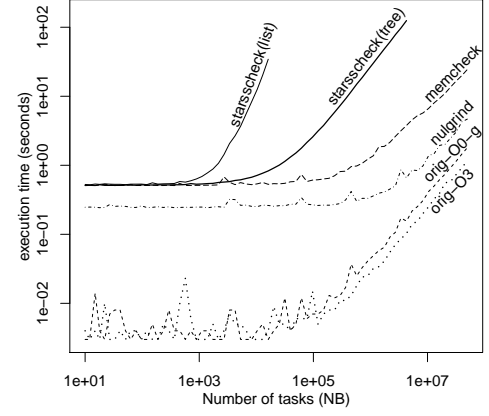
1 #pragma css task inout(p)
2 void f(int *p)
3 {
4     p[0] += 1;
5 }
6
7 int *nasty(void)
8 {
9     int *a = malloc(sizeof(int[NB*2]));
10    int j;
11    memset(a, 0, sizeof(int[NB*2]));
12
13    #pragma css start
14    /* Process even elements */
15    for(j=0; j < NB*2; j += 2)
16        f(&a[j]);
17
18    /* Process odd elements */
19    for(j=1; j < NB*2; j += 2)
20        a[j] += 1;
21    #pragma css finish
22    return a;
23 }
    
```

(a) Source code (nasty)

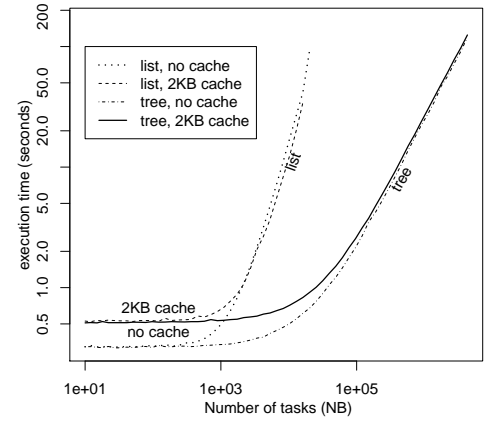
```

1 #pragma css task inout(p)
2 void f(int p[1024])
3 {
4     int k;
5     for(k=0; k<1024; k++)
6         p[k] += 1;
7 }
8
9 int *nasty1k(void)
10 {
11     int *a = malloc(sizeof(int[NB*2048]));
12     int j,k;
13     memset(a, 0, sizeof(int[NB*2048]));
14
15     #pragma css start
16     /* Process even blocks */
17     for(j=0; j < NB*2048; j += 2048)
18         f(&a[j]);
19
20     /* Process odd blocks */
21     for(j=1024; j < NB*2048; j += 2048)
22         for(k=0; k<1024; k++)
23             a[j+k] += 1;
24     #pragma css finish
25     return a;
26 }
    
```

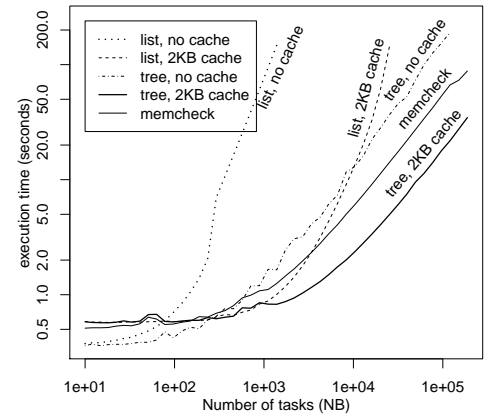
(b) Source code (nasty1k)



(c) StarssCheck in context



(d) StarssCheck variants



(e) After unrolling by 1,024

Figure 5.7: Worst case “nasty” benchmark

## 5. SUPPORT TOOLS

---

<pre>#pragma css task output(x) void b(int x[1]) {     x[0] += 1; }</pre>	<pre>#pragma css task input(x) void f(int x) {     printf("x is %d\n", x); }</pre>
(a) <i>Argument should be marked inout</i>	(c) <i>printf does not respect StarSs</i>

<pre>#pragma css task input(s) output(y) void f(char s[13], int y[1]) {     y[0] = strlen(s); }</pre>
(b) <i>SIMD strlen reads outside array</i>

Figure 5.8: Potential false negatives and false positives

the shared memory hardware. Array region specifiers require a straightforward extension to bounds checking. StarssCheck should skip validity checking for addresses calculated via opaque pointers. This requires tracking, either using static analysis or *shadow bits* similar to memcheck. Tracking of opaque pointers is orthogonal to the rest of the tool.

### Eliminating false positives

Figure 5.8(b) shows an example where StarssCheck can generate spurious warnings. A SIMD implementation of `strlen` may read memory above and below the string itself (but inside the same memory page). Note that memcheck also has difficulty following some C library functions [SN05], and we can use the existing Valgrind machinery to replace functions such as `strlen`. Section 5.3.1 describes how the StreamIt to OmpSs converter introduces `memcpy` operations between the inputs and/or outputs and temporary arrays; these suffer from a similar problem. It does not use Valgrind’s replacement machinery as yet; instead, it uses a simple version of `memcpy` when the code is to be executed under StarssCheck.

Also, certain C library functions, such as `printf`, do not respect the StarSs memory model. The *Suppressions* mechanism of Valgrind can suppress any warnings that arise.

#### 5.1.4 Related work

We are unaware of any other tools that check the input and output definitions of task-based programs. There are, however, several tools that find data races in shared memory. The crucial difference is that StarssCheck verifies information needed by the OmpSs run-time for the program to work at all, whereas the data race detectors discover unordered, and therefore racing, accesses to the same location in memory. Of the motivating examples in Figure 5.1, only subfigure (d) is a data race. The other errors cause the run-time to fail to transfer the correct data in or out of the task.

The Cilk **Nondeterminator** [FL99] finds data races in Cilk programs. A non-deterministic Cilk program is not necessarily wrong, since the language provides locks and it allows communication through shared memory [MIT98]. Nondeterminator checks that programs that are supposed to be deterministic are in fact so. The algorithm assigns an ID to each task

## 5.2 Performance visualisation using Paraver Animator

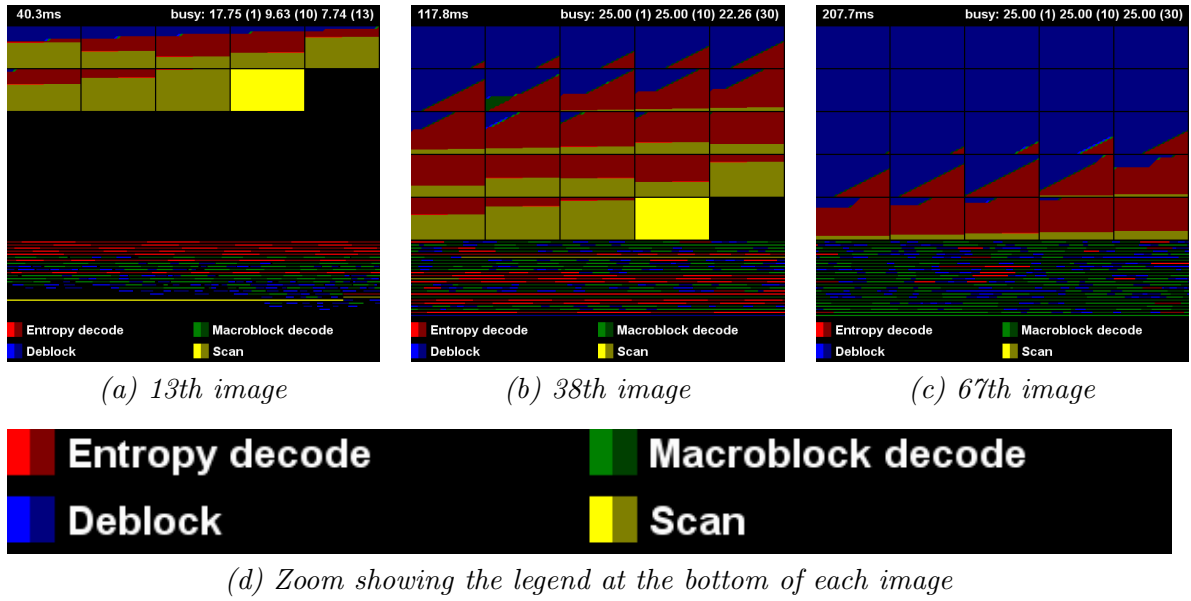


Figure 5.9: Three instants during the execution of the H.264 decoder on 25 processors

at runtime, and maintains, for every location in memory, the IDs of its most recent writer and some previous reader. It checks whether accesses are ordered via Cilk’s shared memory model. Its “SP-bags” algorithm assumes that dependencies between tasks follow a series-parallel DAG, which is true for Cilk but not for OmpSs. Hence it is unlikely to be possible to adapt the algorithm for our purposes.

Many tools find data races in multi-threaded programs. **Helgrind** [NS07] uses the Eraser algorithm [SBN<sup>+</sup>97] to find data races in POSIX pthreads programs. **ThreadSanitizer** [Ser09] uses a hybrid algorithm based on happens-before and locksets. **CORD** [Prv06] and **ReEnact** [PT03] are hardware techniques to detect data races. **MTRAT** [IBM] is a race detector for Java. Since a StarSs program is ultimately implemented using threads, these tools could find the data race in Figure 5.1(d), but they could not check whether the StarSs pragma annotations are correct.

## 5.2 Performance visualisation using Paraver Animator

This section describes the **prvanim** (Paraver Animator) performance tool, which was developed in the course of this thesis. It is a simple tool, which has, nonetheless, proven quite useful.

It accepts a Paraver trace, an *index file*, and various command-line arguments, and it generates an animated GIF, which shows the progress of the program through time. Each image in the GIF file shows the tasks completed or active at the end of its time interval, together with a simplified Paraver trace and optional legend at the bottom.

Figure 5.9 shows three images from an execution of the H.264 decoder on 25 processors. This is the scan variant of the H.264 decoder, which overlaps decoding of multiple output

## 5. SUPPORT TOOLS

---

frames<sup>1</sup>. This example is for twenty five output frames, arranged in each image in row-major order; each pixel is a macroblock in one of the frames. The colours represent the four tasks that must be completed for each macroblock or frame, each of which is either currently working (bright) or completed (dark)—see subfigure (d). Almost all pixels in the images happen to be in one of four states: scanning for the next frame (whole frame is yellow), unprocessed macroblocks in a ready frame (dark yellow), entropy decoded macroblocks (red), or completed macroblocks (blue).

The index file maps task numbers in the Paraver trace to regions in the image. The tool paints the requested regions bright when the task starts and darker when it stops. The programmer has to somehow create the index file. Although this is extra work, it is an opportunity to write an index file that clearly represents the progress of the application, by laying out the image in whichever way makes the most sense. The `str2oss` tool has an option to generate a `prvanim` index file in addition to the C source code. Each row is a filter, and the iterations of the main loop are arranged from left to right; the width of each task is therefore inversely proportional to that filter’s multiplicity.

The `prvanim` tool is useful for understanding the big picture. Once the problem is understood in general terms, one may need to use Paraver to find the root cause—which may be a communications bottleneck, poor cache hit ratio, or some other problem.

### 5.3 StreamIt to OmpSs conversion

This section describes `str2oss` (StreamIt to OmpSs), a tool to translate a streaming program written in StreamIt [TKA02] into a task based program using OmpSs. As mentioned in Section 1.4.1, the tool translated the StreamIt benchmarks to StarSs, which can be compiled using the OmpSs compiler, so that Chapters 3 and 4 could use the same benchmarks.

The tool converts the kernels in the StreamIt program into functions defining StarSs tasks. Each task processes one or more firings of its kernel’s work function. The tool also generates the main function. The main function allocates memory for the FIFOs, and it contains a loop, each iteration of which calls these functions as needed in a steady-state iteration.

The point of `str2oss` was to save time and enable the research in Chapter 4, rather than being a finished product. Nevertheless, it generates high quality code, so long as it is used with care. The major shortcoming of the tool is that it doesn’t support *peeking* forward in the stream, as indicated by `peek N` on a work function. It also doesn’t support feedback loops, variable rate kernels and teleport messaging. There are also some minor deviations from StreamIt syntax, driven by a pragmatic comparison of the cost of fixing `str2oss` and the cost of editing the benchmarks. Section 5.3.3 discusses these limitations of `str2oss` and suggests how they could be remedied.

The tool doesn’t do kernel fusion, as required by the partitioning optimisation from Section 3.2; and the queue sizing optimisation from Section 3.3 is irrelevant for OmpSs. It supports unrolling, under the user’s control, via a configuration file. It always generates code that works, but for good performance the user must control the high-level transformations described in Subsection 5.3.4.

---

<sup>1</sup>The words “frame” and “image” refer to an H.264 frame and an image in the animated GIF, respectively, although both words could have meant either.

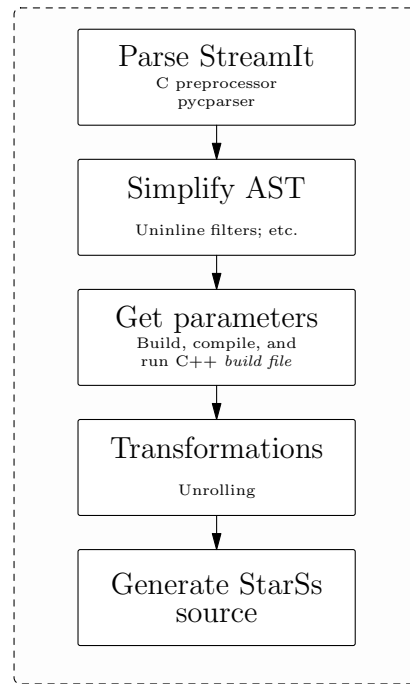


Figure 5.10: Main phases in StreamIt to OmpSs conversion

### 5.3.1 The conversion process

#### Overview and front end

Figure 5.10 shows the steps in converting from StreamIt to StarSs. First, comments are removed using the C preprocessor<sup>1</sup>, and the StreamIt program is parsed, using a parser based on *pycparser* [Ben], to produce an AST (Abstract Syntax Tree).

The second step simplifies the AST. Its most important job is to deal with inlined filters, pipelines and splitjoins. These are written lexically inside their parent, and are most easily dealt with by moving them outside their parent, to the top level, and giving them a name such as `Anonymous_5`. The free variables are collected together to become parameters.

The third step builds the stream graph and finds out, for each filter, the values of its parameters and its static push, pop and peek rates. For the code in Figure 5.14, the parameters for the lowpass filters are `rate`, `cutoff`, `taps` and `decimation`. These values affect the push and pop rates, which are used to calculate the multiplicities, needed for static partitioning and scheduling. Knowing the values of *all* parameters ahead of time allows the main function to simply call the initialisation and work functions, without first having to follow the original hierarchical code to find out their parameters.

The tool constructs the stream graph and finds the parameter values by creating a C++ *build program*. The original StreamIt program is converted to C++, discarding the work and initialisation functions, keeping only the bits that build the stream graph, and calling C++ methods to do so. It prints to `stdout` a representation of the graph as shown in Figure 5.11.

<sup>1</sup>It is convenient for our benchmarks to support the full C preprocessor including `-D...` on the command-line, but the C preprocessor is not part of the StreamIt language.

## 5. SUPPORT TOOLS

Type	Input	Output	Pop	Push	Peek Depth	Name and parameters
1 PIPELINE[	void	void	NA	NA	NA	> SimplifiedFMRadio5
2 FILTER	void	float	0	1	0	>+ FileReader__float "input.in"
3 SPLITJOIN[	float	float	NA	NA	NA	>+ BPFCore Duplicate
4 FILTER	float	float	1	1	1	>++ LowPassFilter 250000000.000000 55.000000 128 0
5 FILTER	float	float	1	1	1	>++ LowPassFilter 250000000.000000 97.998856 128 0
6 SPLITJOIN]						>+ RoundRobin 1 1
7 FILTER	float	float	2	1	2	>+ Subtractor
8 FILTER	float	void	1	0	0	>+ FileWriter__float "output.out"
9 PIPELINE]						
10 Finished						

Figure 5.11: Output from the build program

An alternative would be to analyse the code statically, but the approach described here was easier to implement.

In the output from the build program, the first field is whether the component is a filter, pipeline, or splitjoin. Since it is a linear representation of a hierarchy, each pipeline and splitjoin needs lines to open and close it, represented by the opening and closing square brackets. The second and third fields are the types of the data on the input and output FIFOs. These are usually already known because they are given in the source code, but not always—some inlined stream actors have their types deduced from the context, and this is done by the build program.

If the component is a filter, the next three fields give the number of elements popped, pushed, and peeked for each call to the work function. In StreamIt terminology, the peek count includes the number popped—since the tool doesn’t support peeking forward, the pop and peek counts should be equal. The *depth* field is included only to help debugging, as it gives a visual representation of the depth in the hierarchy. The remaining fields are the function name and the values of its parameters.

After reading in the output from the build program, the next step is to perform the high-level transformations described in Section 5.3.4. Following this, the final phase calculates the filter multiplicities, and generates the StarSs source code as described in the next section.

### Generate StarSs source

Figure 5.12 shows how a StreamIt work function may be converted to a StarSs task. The actual code generated depends on the context in which the filter is being used. The code on the right is for a simple case, and is the unmodified output from `str2oss`. Since the filter’s push and pop rates are known by the caller, in fact at compile time, the items to be popped and pushed are passed, as function arguments `in` and `out`, in arrays of known sizes. The push, pop, and peek<sup>1</sup> operations inside the function are translated to array accesses and pointer arithmetic.

The main program uses arrays to hold the elements in the FIFOs. Figure 5.13 shows parts of the stream graphs for three StreamIt programs, with each subfigure indicating in bold a single array. Subfigure (a) is the simplest case: a FIFO straight from one filter, A, to another filter, B. The main program will allocate a block of memory large enough to contain

<sup>1</sup>The tool does not support peeking outside the input array, but it does support the peek operator, which is just an array subscript, so long as it is only used to read data that will be popped.

<pre> 1 int-&gt;int filter Example 2 { 3     work pop 20 push 10 4     { 5         for(int j=0; j&lt;10; j++) 6         { 7             int x = peek(0) + peek(1); 8             pop(); 9             pop(); 10            push(x); 11        } 12    } 13 }                 </pre> <p>(a) <i>StreamIt filter</i></p>	<pre> 16 #pragma cxx task input(in[20]) output(out[10]) 17 void Example(const int *restrict in, int *restrict out) 18 { 19     for(int j = 0; j &lt; 10; j++) 20     { 21         int x = (in[0] + in[1]); 22         *in++; 23         *in++; 24         *out++ = x; 25     } 26 }                 </pre> <p>(b) <i>Corresponding StarSs function</i></p>
---	--

Figure 5.12: Translation of an example StreamIt function

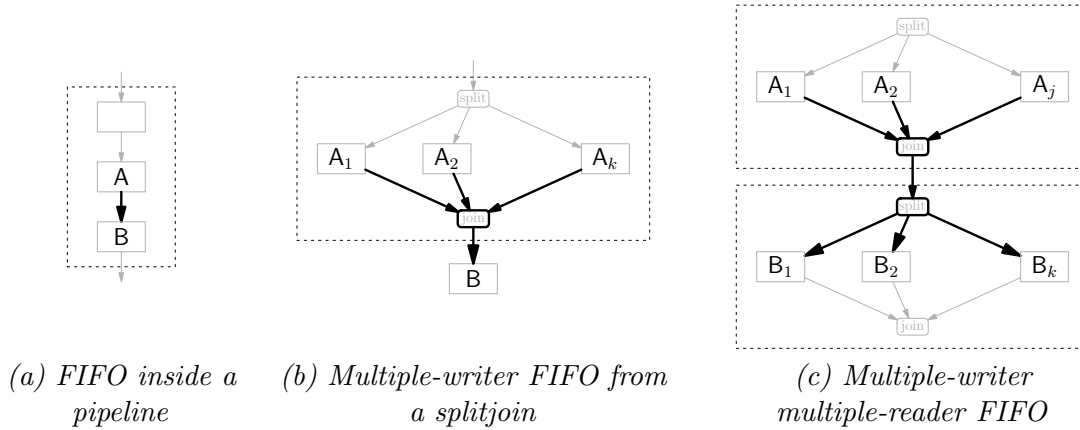


Figure 5.13: Various FIFOs

all the data passed in a steady state iteration, and call functions similar to the function in Figure 5.12(b), one or more times per iteration, to write and read the array. A efficient serial implementation would share one array among several streams, if their data weren't live at the same time, but this is not done by `str2oss`.

There is a complication, which is caused by a limitation of the OmpSs runtime. OmpSs doesn't support dependencies via input and output arrays that partially overlap. A task's input array must coincide exactly with the output array of the task that generated the data, and it must be disjoint with every other output—similarly for other kinds of dependency. This is a reasonable limitation, because the runtime finds the dependency using an associative map from the block's start address to an internal data structure. Any more complicated mechanism would likely require greater runtime overhead.

The number of items pushed by the producer task may be different from the number popped by the consumer task. When they are different, a natural implementation would have them communicating via partially overlapping arrays, which would not work.

The approach taken by `str2oss` is to split the FIFO into segments of length the greatest common divisor of the push and pop rates. Rather than having a single output argument, `out`, to contain all the elements it will push, the producer gets one output argument per segment; similarly for the consumer. The data is manipulated in temporary arrays and gathered from or scattered to the arguments using `memcpy`. This approach sometimes results in an excessive

## 5. SUPPORT TOOLS

---

number of function arguments, often solved with a careful choice of unrolling factor for one or both filters. This is the unrolling transformation described in Section 5.3.4. Although unrolling solves the problem on this stream, it may cause problems elsewhere, particularly if the unrolled filter is connected on the other side to a splitjoin. This behaviour is rather unintelligent, but it is easy to understand, and easy for the high-level phase to model.

Figure 5.13(b) shows a FIFO that is the output of a round-robin join. The `str2oss` tool does not create a task just to do the job of the joiner. In some cases, this would be the right thing to do—see below, but, again, it is a high-level transformation. The elements in the array are ordered, in memory, exactly in the order they will be popped by B. The main program calls kernels  $A_1$  through  $A_k$ , having them write their contributions directly into this array. If a single firing of any of the producers has to write output that is not one contiguous section, then its output argument must be split into several arguments. The function arguments may need to be split further to deal with the limitation of OmpSs that was handled above. Conversely, the *input* to a splitjoin is handled in a similar way: it is an array with multiple readers instead of multiple writers. Figure 5.13(c) is an example where a stream leaves one splitjoin and enters another, so the array has multiple writers and multiple readers.

More complicated examples are handled by inserting an *identity* kernel, which does nothing except pass on data. For instance, if  $A_1$  in Figure 5.13(c) were not a kernel, but a splitjoin, an identity kernel would be inserted after its join. The reason is that it is genuinely harder to support nested splitjoins in the manner described above. In a single splitjoin, described in the previous paragraph, each filter writes a subset of the elements of its output array following a simple pattern: first skip so many elements; then for the rest of the array, alternate between writing a fixed number of elements and skipping a fixed number of elements. A filter inside a nested splitjoin, if not isolated from the outer one, could have to follow a more complicated pattern.

### 5.3.2 Example: simplified FM Radio

This section illustrates the conversion process using the example from Figure 1.10, a simplified version of part of the StreamIt fm benchmark. Since `str2oss` doesn't support peeking forward in streams, the program has to be changed to work without peeking. This can often be done in the way shown in Figure 5.14, by adding state that remembers previous input: a history buffer. This program is not quite the same as the original, because there are 127 new samples, all zeroes, at the beginning of the output. There is also a performance overhead due to shifting samples in the history buffer, but this overhead can be reduced using a more careful unrolled implementation.

Figures 5.15 and 5.16 show the output from `str2oss`. The program as it stands is not very efficient, since the tasks are too small. They would normally be unrolled using the mechanism in Section 5.3.4.

The state on lines 16 and 17 of the StreamIt source code is combined into the structure defined on lines 7 through 11, and initialised by the `init` function on lines 13 through 27. The low pass filter is defined in lines 29 through 41. The subtracter, defined in lines 43 through 53, has its input arrays split in two, since it follows a two-way round robin join.



## 5.3 StreamIt to OmpSs conversion

```
1 /*
2  * Copyright 2001 Massachusetts Institute of Technology
3  *
4  * Permission to use, copy, modify, distribute, and sell this software and its
5  * documentation for any purpose is hereby granted without fee, provided that
6  * the above copyright notice appear in all copies and that both that
7  * copyright notice and this permission notice appear in supporting
8  * documentation, and that the name of M.I.T. not be used in advertising or
9  * publicity pertaining to distribution of the software without specific,
10 * written prior permission. M.I.T. makes no representations about the
11 * suitability of this software for any purpose. It is provided "as is"
12 * without express or implied warranty.
13 */
14
15 float->float filter LowPassFilter(float rate, float cutoff, int taps, int decimation) {
16     float[taps] coeff;
17     float[taps] history;
18     init {
19         int i;
20         float m = taps - 1;
21         float w = 2 * pi * cutoff / rate;
22         for (i = 0; i < taps; i++) {
23             history[i] = 0.0;
24             if (i - m/2 == 0)
25                 coeff[i] = w/pi;
26             else
27                 coeff[i] = sin(w*(i-m/2)) / pi / (i-m/2) *
28                     (0.54 - 0.46 * cos(2*pi*i/m));
29         }
30     }
31     work pop 1+decimation push 1 {
32         float sum = 0;
33         int pop_count = 1+decimation;
34         // Put 1+decimation new samples into the history
35         for (int i = 0; i < taps - pop_count; i++)
36             history[i] = history[i+pop_count];
37         for (int i = 0; i < pop_count; i++)
38             history[taps-pop_count+i] = pop();
39         for (int i = 0; i < taps; i++)
40             sum += history[i] * coeff[i];
41         push(sum);
42     }
43 }
44
45 float->float splitjoin BPFCore(float rate, float low, float high, int taps) {
46     split duplicate;
47     add LowPassFilter(rate, low, taps, 0);
48     add LowPassFilter(rate, high, taps, 0);
49     join roundrobin;
50 }
51
52 float->float filter Subtractor {
53     work pop 2 push 1 {
54         push(peek(1) - peek(0));
55         pop(); pop();
56     }
57 }
58
59 void->void pipeline SimplifiedFMRadio5 {
60     float samplingRate = 250000000; // 250 MHz
61     float cutoffFrequency = 108000000; // 108 MHz
62     float low = 55.0;
63     float high = 97.998856;
64     int taps = 128;
65
66     add FileReader<float>("input.in");
67     add BPFCore(samplingRate, low, high, taps);
68     add Subtractor();
69     add FileWriter<float>("output.out");
70 }
```

Figure 5.14: Non-peek version of the example StreamIt 2.1 program in Figure 1.10

## 5. SUPPORT TOOLS

---

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5 #define pi 3.14159265
6
7 typedef struct
8 {
9     float coeff[128];
10    float history[128];
11 } vars_LowPassFilter_0_t;
12
13 void init_LowPassFilter_0(vars_LowPassFilter_0_t *vars, float rate, float cutoff, int taps, int
    decimation)
14 {
15     int i;
16     float m = (128 - 1);
17     float w = (((2 * pi) * cutoff) / rate);
18     for(i = 0; i < 128; i++)
19     {
20         vars->history[i] = 0.0;
21         if (((i - (m / 2)) == 0))
22             vars->coeff[i] = w / pi;
23         else
24             vars->coeff[i] = ((sin(w * (i - (m / 2))) / pi) / (i - (m / 2))) * (0.54 - (0.46 * cos
                (((2 * pi) * i) / m)));
25     }
26 }
27
28
29 #pragma css task input(in[(1 + decimation)]) inout(out[1]) input(rate, cutoff, taps, decimation)
    inout(vars)
30 void LowPassFilter_0(const float *restrict in, float *restrict out, float rate, float cutoff, int
    taps, int decimation, vars_LowPassFilter_0_t *restrict vars)
31 {
32     float sum = 0;
33     int pop_count = (1 + decimation);
34     for(int i = 0; i < (taps - pop_count); i++)
35         vars->history[i] = vars->history[(i + pop_count)];
36     for(int i = 0; i < pop_count; i++)
37         vars->history[(taps - pop_count) + i] = *in++;
38     for(int i = 0; i < taps; i++)
39         sum += vars->history[i] * vars->coeff[i];
40     *out++ = sum;
41 }
42
43 #pragma css task input(in_0[2/2], in_1[2/2]) output(out[1])
44 void Subtractor__2_1(const float *restrict in_0, const float *restrict in_1, float *restrict out)
45 {
46     float in_array[2];
47     memcpy(in_array + 0*(2/2), in_0, sizeof(float[2/2]));
48     memcpy(in_array + 1*(2/2), in_1, sizeof(float[2/2]));
49     float *in = in_array;
50     *out++ = in[1] - in[0];
51     *in++;
52     *in++;
53 }
54
55 #pragma css task output(out[1]) inout(state) input(filename)
56 void FileReader__float(float *restrict out, FILE *state, const char *filename)
57 {
58     fread(out, sizeof(float), 1, state);
59 }
60
61 #pragma css task input(in[1]) inout(state) input(filename)
62 void FileWriter__float(const float *restrict in, FILE *state, const char *filename)
63 {
64     fwrite(in, sizeof(float), 1, state);
65 }
66
67 int main(int argc, char **argv)
68 {
69     int iter;
70     float out_FileReader__float[1];
71     float out_BPFCore[2];
72     float out_Subtractor[1];
73     FILE *state_FileReader__float = fopen("input.in", "rw");
74     if (!state_FileReader__float)
75     {
76         fprintf(stderr, "Cannot open file 'input.in' for reading\n");
77         exit(1);
78     }
```

Figure 5.15: Translated code exactly as generated by str2oss: part 1

```

79 FILE *state_FileWriter__float = fopen("output.out", "wb");
80 if (!state_FileWriter__float)
81 {
82     fprintf(stderr, "Cannot open file 'output.out' for writing\n");
83     exit(1);
84 }
85 vars_LowPassFilter_0_t vars_LowPassFilter_0;
86 vars_LowPassFilter_0_t vars_LowPassFilter_1;
87
88 init_LowPassFilter_0(&vars_LowPassFilter_0, 250000000.000000, 55.000000, 128, 0);
89 init_LowPassFilter_0(&vars_LowPassFilter_1, 250000000.000000, 97.998856, 128, 0);
90
91 #pragma css start
92 for (iter=0; iter<10; iter++)
93 {
94     FileReader__float(out_FileReader__float, state_FileReader__float, "input.in");
95     LowPassFilter_0(out_FileReader__float, out_BPFCore, 250000000.000000, 55.000000, 128, 0, &
vars_LowPassFilter_0);
96     LowPassFilter_0(out_FileReader__float, (float *)out_BPFCore + 1, 250000000.000000,
97.998856, 128, 0, &vars_LowPassFilter_1);
97     Subtractor__2_1(out_BPFCore, (float *)out_BPFCore + 1, out_Subtractor);
98     FileWriter__float(out_Subtractor, state_FileWriter__float, "output.out");
99 }
100 #pragma css finish
101 fclose(state_FileReader__float);
102 fclose(state_FileWriter__float);
103 return 0;
104 }

```

Figure 5.16: Translated code exactly as generated by str2oss: part 2

### 5.3.3 Current limitations

The initial implementation of `str2oss` does not support peeking, feedback loops, variable rate kernels, or teleport messaging. To support peeking, the tool must allocate additional buffering, since stream elements are live for a longer time. It must also modify task creation order and add a prologue before the main loop, since peeking forward in the stream modifies the schedule.

There are three ways to implement the additional buffering. The first way is to implement a history buffer inside the task, which requires inout state. This transformation was done manually in Section 5.3.2, and it serialises the filter. The second way is to split the filter in two: the first half updates the history, and is therefore serialised. The appropriate data is passed to the second half of the filter, which does the work, and need not be serialised (unless there is some other inout state). The third way is to allocate and manage the additional buffering in the main function, and pass the appropriate data to the tasks using additional function arguments.

The prologue and main loop can be created using a technique such as phased scheduling [KTA03], which takes account of peeking forward in the streams.

Feedback loops present no special problems. Since feedback loops were not used in the StreamIt benchmarks, there was no requirement to implement support for them.

Variable rate kernels are hard to support without changing the StarSs programming model. In StarSs, the dependency graph must be constructed by the main function, which requires knowing the number of elements pushed or popped by every kernel's work function. When a kernel has variable rates, the main function must therefore synchronise on every one of its tasks. This will likely introduce an unacceptably large overhead and constrain parallelism.

Teleport messaging allows occasional messages to be sent between filters, outside the normal stream flow. Messages are asynchronous method calls from an upstream (sending)

## 5. SUPPORT TOOLS

---

```
1 UNROLL 2000 PolarToRectangular
2 UNROLL 2000 FloatVoid
3 UNROLL 100 .*
4 JOIN 100 FilterBank_0
5 SPLIT 1500 Anonymous_0_0
6 JOIN 2000 Anonymous_0_0
7 SPLIT 100 Anonymous_2_0
8 JOIN 100 Anonymous_2_0
```

Figure 5.17: Example `str2oss` control file for Vocoder

filter to a downstream (receiving) filter, which have guaranteed upper and lower bounds on latency. Latency is measured, not using some notion of global time, but by reference to a pull schedule [CAG06].

Teleport messaging constrains dynamic scheduling [TKS<sup>+</sup>05]. If an upstream filter might send a message with fixed latency; i.e. the upper and lower bounds are equal, then each downstream filter task cannot start executing until the upstream task that may have sent it a message has finished executing. These dependencies must always be added to the dependency graph, even in the common case where there is no message.<sup>1</sup> To add these dependencies, the downstream task must be created after the upstream task, which might not otherwise have been the case, especially if the minimum latency is large.

In addition to the major shortcomings described above, the tool requires some small changes to the StreamIt source code. Some of these relate to differences between StreamIt's Java-derived syntax and that of C. Specifically, `str2oss` requires a semicolon after a `typedef`, and it requires the keyword `struct` before a struct name. Numbers beginning with a zero are treated as octal, whereas the `mpeg2` benchmark for instance expects them to be decimal. These limitations can be addressed by modifying the parser.

Unlike StreamIt, variables are not automatically initialised to zero. There is no support for nesting ordinary C functions inside components; the programmer just needs to move them outside, adding arguments as necessary. The tool generates work functions with arguments with names like `in`, `out`, and `state`. These can be, and sometimes are, the same as local variables, causing problems.

The tool has incomplete support for StreamIt array declarations: the StreamIt declaration `int[N] permutation` is similar to the C declaration `int permutation[N]`. For this reason, the parameters to filters must be scalars. Two benchmarks have arrays passed as parameters: `Serpent` and `fm`. The workaround for `Serpent` is to change the second parameter from an array to a Boolean, which selects between IP and FP in the static data. The workaround for `fm` is even simpler: the work function depends on just one element of the `gains` array. This limitation can be addressed by supporting StreamIt array declarations properly.

---

<sup>1</sup>Since messages are assumed to be sent infrequently, greater parallelism may be exploitable if OmpSs were extended to use speculation.

### 5.3.4 High level transformations

**Unrolling** The `str2oss` tool has limited support for the unrolling transformation, but only under the direct control of the user or external tool. The unrolling transformation is driven using a *control file*, such as that shown in Figure 5.17. Each line in this file is a directive to `str2oss`, which has a directive type, unrolling factor, and a regular expression. The regular expression matches the task function or task function call.

An `UNROLL` directive unrolls a task function. For example, the first line in Figure 5.17 unrolls the `PolarToRectangular` function by a factor of 2000. This is done by inserting a `for` loop around the body of its work function, and multiplying the push and pop rates by this number. The filter unrolling pass happens after parsing the StreamIt code, but before creating and running the build program. Line 3 unrolls every function not matched by the first two lines by a factor of 100.

The `JOIN` and `SPLIT` directives fix the problem illustrated in Figure 5.18. Figure 5.18(a) shows part of the source code for the `des` benchmark, slightly simplified. As can be seen in the diagram of Figure 5.18(b), the outputs of `f` and `ld` are interleaved by a round-robin join. The code generation method described in Section 5.3.1 requires `f` to write the even elements into the array in the right places, and `ld` to similarly write the odd elements. For StarSs, since each argument must be a single block of contiguous memory, this requires the functions to both have, for their outputs, 32 function arguments.<sup>1</sup>

The solution is to transform the source code into that shown in Figure 5.18(c), resulting in the stream graph in Figure 5.18(d). The round-robin join has been changed from (1,1) to (32,32), and a permute filter has been inserted to put the stream back into the right order. A single call to `f` writes one block of contiguous memory, so the function has a single output array, and similarly for `ld`. This transformation is performed by the directive “`JOIN 32 Body_1`”. The `SPLIT` directive works similarly for splits. The regular expression is matched against the specific *instance* of the `splitjoin`, hence the suffix `_1`.

Figure 5.2 gives statistics for conversion of the StreamIt benchmarks. Subfigure (a) shows the statistics for the benchmarks, as used in Chapter 4, with manually chosen high level transformations. The only benchmark with an excessive number of function arguments is `MPEGdecoder`. This benchmark contains a round-robin split with arguments 384, 16, and 3. After the high-level transformations, the producer pushes 40 300 elements at a time into the split, which has arguments 38 400, 1 600, and 300. This FIFO must be split into segments, as described in Section 5.3.1, to avoid partially overlapping arrays. The current implementation requires all segments to be the same length. The segment is the greatest common divisor, which is 100, so the producer has 403 output segments. There is one additional function argument, which is a parameter to the filter. The solution is to fix this one benchmark by hand.

Figure 5.2(b) shows the same statistics when all filters are unrolled by a factor of 100. Some functions have an absurd number of function arguments, up to 45 thousand, for the reasons explained earlier in this section. For comparison, Figure 5.2(c) gives the same statistics for the StreamIt benchmarks with no high-level transformations at all.

<sup>1</sup>This problem is alleviated by region support in SMPs, which is not implemented in the Nanos++ runtime.

## 5. SUPPORT TOOLS

---

**Kernel fusion** Kernel fusion is not supported, but it could be implemented by modifying the graph representation and work functions immediately following unrolling. The transformation should be driven by a convex partition, determined, as above, either by the user or by an external tool, perhaps using the heuristic in Section 3.2. The transformation should contract the kernels to be fused into a single vertex. Any internal streams will disappear from the stream graph, but will become temporary buffering inside the task. The multiplicity of the task should be the greatest common divisor of the multiplicities of its kernels.

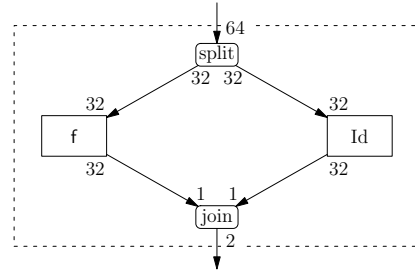
## 5.3 StreamIt to OmpSs conversion

```

1  int->int filter f
2  {
3      work pop 32 push 32
4      {
5          // Body of work not important
6          for(int j=0; j<32; j++)
7              push(pop());
8      }
9  }
10
11 int->int splitjoin Body
12 {
13     split roundrobin(32, 32);
14     add f();
15     add Identity<int>();
16     join roundrobin(1, 1);
17 }
18
19 void->void pipeline RoundRobin_32_32
20 {
21     add FileReader<int>("rr.in");
22     add Body();
23     add FileWriter<int>("rr.out");
24 }

```

(a) Original StreamIt source



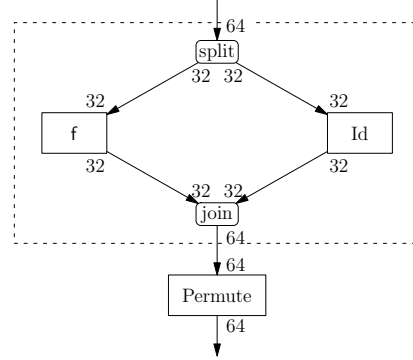
(b) Stream graph, with push and pop rates

```

42 int->int filter f
43 {
44     work pop 32 push 32
45     {
46         // Body of work not important
47         for(int j=0; j<32; j++)
48             push(pop());
49     }
50 }
51
52 int->int pipeline Body
53 {
54     add splitjoin
55     {
56         split roundrobin(32, 32);
57         add f();
58         add Identity<int>();
59         join roundrobin(32, 32);
60     }
61     add Permute(32,1);
62 }
63
64 void->void pipeline RoundRobin_32_32
65 {
66     add FileReader<int>("rr.in");
67     add Body();
68     add FileWriter<int>("rr.out");
69 }

```

(c) Transformed StreamIt source



(d) Stream graph, with push and pop rates

Figure 5.18: An example where str2oss generates a function with too many arguments

## 5. SUPPORT TOOLS

Benchmark	Maximum mult.	Maximum #args	Average #args	#function fcalls	Tasks per iter.	Data bytes
BeamFormer1	2	13	6.27	55	79	68800
BitonicSort	2	5	3.17	58	60	99200
ChannelVocoder	2	18	6.68	57	58	746400
DCT	1	17	5.15	40	40	921600
DES	3	7	3.40	57	69	1177600
FFT	128	5	4.71	17	384	3276800
FilterBank	8	11	6.63	76	426	243200
FMRadio	1	8	4.15	48	48	27200
MPEGdecoder	6	404	26.85	40	180	3450400
Serpent	4	36	3.70	408	432	6976000
tde	480	19	5.72	29	2869	20160000
VocoderTopLevel	20	30	4.66	120	248	342800

(a) As used in Chapter 4, with manually chosen transformations

Benchmark	Maximum mult.	Maximum #args	Average #args	#function fcalls	Tasks per iter.	Data bytes
BeamFormer1	2	1204	134.08	53	77	57600
BitonicSort2	1	1200	411.89	36	36	41600
ChannelVocoder7	50	3200	143.16	55	1721	733200
DCT2	256	3202	2330.75	36	546	512000
DES2	48	6432	1508.60	45	357	870400
FFT5	128	5	4.71	17	384	3276800
FilterBank6	8	802	78.18	67	354	166400
FMRadio5	1	602	91.44	41	41	20000
MPEGdecoder	384	45002	6834.81	31	907	1880400
Serpent	128	13600	1401.04	358	2021	3955200
tde	480	19	5.72	29	2869	20160000
VocoderTopLevel	20	4000	306.70	106	249	209600

(b) Unrolled only, all filters by factor of 100

Benchmark	Maximum mult.	Maximum #args	Average #args	#function fcalls	Tasks per iter.	Data bytes
BeamFormer1	2	16	7.06	53	77	576
BitonicSort2	1	12	4.89	36	36	416
ChannelVocoder7	50	100	24.36	55	1721	7332
DCT2	256	258	44.67	36	546	5120
DES2	48	96	37.67	45	357	8704
FFT5	128	5	4.71	17	384	32768
FilterBank6	8	11	7.25	67	354	1664
FMRadio5	1	8	4.51	41	41	200
MPEGdecoder	384	452	74.06	31	907	18804
Serpent	128	160	15.59	358	2021	39552
tde	480	19	5.72	29	2869	201600
VocoderTopLevel	20	45	5.96	106	249	2096

(c) Not unrolled

Table 5.2: Translation statistics for StreamIt benchmarks



## Chapter 6

# Conclusions

This thesis has developed several new compiler and run-time techniques for stream programming. In conclusion, the main contributions are as follows:

1. The **Abstract Streaming Machine (ASM)**, a flexible machine description and coarse-grain simulator for a statically scheduled stream compiler. The ASM machine model uses a bipartite graph representation to describe the target system to the compiler. The coarse-grain simulator models the program’s dynamic behaviour.

[CRM<sup>+</sup>07] Paul Carpenter, David Rodenas, Xavier Martorell, Alejandro Ramirez, and Eduard Ayguadé. A streaming machine description and programming model. Proc. of the International Symposium on Systems, Architectures, Modeling and Simulation, Samos, Greece, July 16–19, 2007.<sup>1</sup>

[ACO08] ACOTES. IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version. 2008.

[CRA09b] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. The Abstract Streaming Machine: Compile-Time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors. In SAMOS Workshop 2009, pages 12–13. *Best paper award*.

[CRA11] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. The Abstract Streaming Machine: Compile-Time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors. Transactions on HiPEAC, 5(3).

[M<sup>+</sup>11] Harm Munk et al. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. International Journal of Parallel Programming, 39:397–450, 2011<sup>1</sup>.

2. A new **partitioning heuristic** for stream programs, which balances the load across the target, taking account of the processors and communication links. A good partitioning algorithm is crucial if the compiler is to produce efficient code. This algorithm also considers its effect on downstream passes, specifically software pipelining and buffer allocation, using a convexity constraint to control pipeline length. It uses a new formulation of connectivity to model the compiler’s ability to fuse kernels.

[CRA09a] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Mapping Stream Programs onto Heterogeneous Multiprocessor Systems. In CASES ’09: Proceedings of the 2009 Inter-

---

<sup>1</sup>These papers also contain material that is not a contribution of this thesis.

## 6. CONCLUSIONS

---

national Conference on Compilers, Architectures, and Synthesis for Embedded Systems, pages 57–66, 2009.

3. Two static **queue sizing algorithms** for stream programs, which determine the sizes of the buffers used to implement streams. The queue sizing problem is important when memory is distributed, especially when local stores are small. The algorithm adjusts the sizes of the buffers, subject to memory capacities, in order to cover latency and variability in computation costs.

[CRA10b] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Buffer Sizing for Self-timed Stream Programs on Heterogeneous Distributed Memory Multiprocessors. In High Performance Embedded Architectures and Compilers, 5th International Conference, HiPEAC 2010, pages 96–110.

4. Two new low-complexity **adaptive dynamic scheduling algorithms** for stream-like programs. The *apriority* scheduler is specific to one-dimensional stream programs, and it requires information from the compiler. The *gpriority* scheduler does not need such information, and it is more general.

5. **StarssCheck, a debugging tool for StarSs**. StarssCheck runs the program under Valgrind, using a special analysis tool, and generates a warning whenever the program’s behaviour contradicts the StarSs pragma annotations. Many of the errors found by StarssCheck would otherwise be difficult to diagnose, since they would cause race conditions or exceptions deep inside the runtime system.

[CRA10a] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs. Euro-Par 2010–Parallel Processing, pages 2–13, 2010.

---

## Glossary

<b>API</b>	Application Programming Interface.	<b>LIFO</b>	Last In First Out.
<b>ASM</b>	Abstract Streaming Machine.	<b>LS</b>	Local Store.
<b>AST</b>	Abstract Syntax Tree.	<b>Mercurium</b>	Source-to-source compiler used by OmpSs.
<b>BLAS</b>	Basic Linear Algebra Subprograms.	<b>Nanos++</b>	Runtime library for asynchronous task parallelism, which supports multiple programming languages: StarSs, OpenMP, and Chapel.
<b>Cell B.E.</b>	Cell Broadband Engine.	<b>OmpSs</b>	An implementation of StarSs, which also supports the OpenMP standard. The OmpSs compiler accepts both the StarSs syntax and the newer OmpSs syntax. The OmpSs run-time system is built on Nanos++.
<b>CPU</b>	Central Processing Unit.	<b>OS</b>	Operating System.
<b>DAG</b>	Directed Acyclic Graph.	<b>PDG</b>	Partial Dependency Graph.
<b>DCT</b>	Discrete Cosine Transform.	<b>POSIX</b>	Portable Operating System Interface for Unix.
<b>DMA</b>	Direct Memory Access.	<b>PPE</b>	Power Processing Element.
<b>DSL</b>	Domain-specific Language.	<b>SDF</b>	Synchronous Data Flow.
<b>DSP</b>	Digital Signal Processor.	<b>SGMS</b>	Stream Graph Modulo Scheduling.
<b>FFT</b>	Fast Fourier Transform.	<b>SIMD</b>	Single Instruction Multiple Data.
<b>FIFO</b>	First In First Out.	<b>SMP</b>	Symmetric Multiprocessor.
<b>GCC</b>	GNU Compiler Collection.	<b>SPE</b>	Synergistic Processing Element.
<b>GIF</b>	Graphics Interchange Format.	<b>SPM</b>	Stream Programming Model.
<b>GPU</b>	Graphics Processing Unit.	<b>StarSs</b>	Star Superscalar, an extension of C to support task-level parallelism.
<b>HD</b>	High Definition.		
<b>ILP</b>	Integer Linear Programming.		
<b>IO</b>	Input/Output.		
<b>ISA</b>	Instruction Set Architecture.		
<b>KPN</b>	Kahn Process Network.		
<b>LAPACK</b>	Linear Algebra Package.		

---

# Bibliography

- [ACD74] T.L. Adam, K.M. Chandy, and JR Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):690, 1974. [73](#), [74](#)
- [ACO] ACOTES IST-034869. <http://www.hitech-projects.com/euprojects/ACOTES/>. Advanced Compiler Technologies for Embedded Streaming. [8](#)
- [ACO08] ACOTES. IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version, 2008. [6](#), [8](#), [16](#), [19](#), [121](#)
- [AG02] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 2002. [3](#), [24](#)
- [AHSW62] J.P. Anderson, S.A. Hoffman, J. Shifman, and R.J. Williams. D825-a multiple-computer system for command & control. In *Proceedings of the December 4-6, 1962, fall joint computer conference*, pages 86–96. ACM, 1962. [1](#)
- [AK02] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers, 2002. [3](#)
- [Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM New York, NY, USA, 1967. [23](#)
- [App] Apple, Inc. <http://developer.apple.com/library/mac/#featuredarticles/BlocksGCD/-index.html>. Introducing Blocks and Grand Central Dispatch. [5](#)
- [ASRV07] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. HD-VideoBench. A Benchmark for Evaluating High Definition Digital Video Applications. In *IISWC 2007*, pages 120–125, 2007. [89](#)
- [ATN10] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Research Report RR-7240, INRIA, 03 2010. [5](#)
- [ATNW09] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Euro-Par 2009 Parallel Processing*, pages 863–874, 2009. [5](#)
- [Bar08] Barcelona Supercomputing Center. *SMP Superscalar (SMPSs) User’s Manual Version 2.0*, 2008. [10](#)
- [Bar09] Barcelona Supercomputing Center. *Cell Superscalar (CellSs) User’s Manual Version 2.2*, 2009. [10](#)

- 
- [BDG<sup>+</sup>04] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, and J. Labarta. Nanos Mercurium: a Research Compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP*, volume 2004, 2004. 8, 99
- [Ben] Eli Bendersky. <http://code.google.com/p/pycparser/>. pycparser. 109
- [BGH<sup>+</sup>90] J.C. Bier, E.E. Goei, W.H. Ho, P.D. Lapsley, M.P. O'Reilly, G.C. Sih, and E.A. Lee. Gabriel: a design environment for dsp. *Micro, IEEE*, 10(5):28–45, October 1990. 6
- [BH01] T. Basten and J. Hoogerbrugge. Efficient execution of process networks. *Communicating Process Architectures*, 2001. 61, 64, 67
- [BJK<sup>+</sup>95] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995. 5, 74
- [BKSS02] M.D. Beynon, T. Kurc, A. Sussman, and J. Saltz. Optimizing execution of component-based applications using group instances. *Future Generation Computer Systems*, 18(4):435–448, 2002. 6
- [BML96] S.S. Battacharyya, P.K. Murthy, and E.A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Pub, 1996. 50
- [Bow69] Sr. Bowdon, E.K. Priority assignment in a network of computers. *Computers, IEEE Transactions on*, C-18(11):1021–1026, November 1969. 74
- [BPBL06] P. Bellens, J.M. Perez, R.M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. ACM New York, NY, USA, 2006. 5, 10
- [Buc93] J.T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, University of California, 1993. 48, 58
- [Buc03] I. Buck. Brook Spec v0. 2, 2003. 6, 67
- [CAG06] CAG MIT. *StreamIt Language Specification, Version 2.1*, 2006. 6, 12, 16, 116
- [CCG<sup>+</sup>00] J. Chaoui, K. Cyr, J.P. Giacalone, S. Gregorio, Y. Masse, Y. Muthusamy, T. Spits, M. Budagavi, and J. Webb. OMAP: Enabling Multimedia Applications in Third Generation (3G) Wireless Terminals. *SWPA001, December*, 2000. 2, 23
- [CEP] CEPBA. <http://www.cepba.upc.edu/paraver/>. Paraver performance visualization and analysis tool. 18, 29
- [CEP01] CEPBA. *Paraver Version 3.0 Parallel Program Visualization and Analysis tool: Tracefile Description*, 2001. 18
- [CGT04] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. *Lecture notes in computer science*, pages 292–303, 2004. 8
- [CHM95] C. Chekuri, W. Hasan, and R. Motwani. Scheduling problems in parallel query optimization. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 255–265. ACM, 1995. 6

- 
- [CLC<sup>+</sup>09] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. Stream Compilation for Real-Time Embedded Multicore Systems. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 210–220. IEEE Computer Society Washington, DC, USA, 2009. [46](#), [68](#)
  - [CRA09a] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 57–66, 2009. [19](#), [38](#), [121](#)
  - [CRA09b] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. The Abstract Streaming Machine: Compile-Time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors. In *SAMOS Workshop*, pages 12–23. Springer, 2009. [19](#), [121](#)
  - [CRA10a] P. Carpenter, A. Ramirez, and E. Ayguade. Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs. *Euro-Par 2010-Parallel Processing*, pages 2–13, 2010. [18](#), [20](#), [122](#)
  - [CRA10b] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Buffer sizing for self-timed stream programs on heterogeneous distributed memory multiprocessors. In *High Performance Embedded Architectures and Compilers, 5th International Conference, HiPEAC 2010*, pages 96–110. Springer, 2010. [9](#), [20](#), [39](#), [122](#)
  - [CRA11] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. The Abstract Streaming Machine: Compile-Time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors. *Transactions on HiPEAC*, 5(3), 2011. [19](#), [121](#)
  - [CRDI05] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation. *IBM developerWorks*, 2005. [10](#), [26](#)
  - [CRM<sup>+</sup>07] Paul Carpenter, David Rodenas, Xavier Martorell, Alejandro Ramirez, and Eduard Ayguadé. A streaming machine description and programming model. *Proc. of the International Symposium on Systems, Architectures, Modeling and Simulation, Samos, Greece, July 16-19, 2007*, 2007. [6](#), [8](#), [16](#), [17](#), [19](#), [121](#)
  - [CSB<sup>+</sup>11] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A Domain-Specific Approach To Heterogeneous Parallelism. In *16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, February 2011. [6](#)
  - [DFA<sup>+</sup>09] A. Duran, R. Ferrer, E. Ayguadé, R.M. Badia, and J. Labarta. A proposal to extend the openmp tasking model with dependent tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009. [10](#)
  - [DG98] A. Dasdan and RK Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(10):889–899, 1998. [59](#)
  - [DG08] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. [3](#)
  - [DYDS<sup>+</sup>10] M. Duranton, S. Yehia, B. De Sutter, K. De Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, et al. The HiPEAC vision. *Network of Excellence of High Performance and Embedded Architecture and Compilation, Tech. Rep*, 2010. [6](#)

- 
- [EJL<sup>+</sup>03] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003. 6, 65
  - [ERB<sup>+</sup>10] Yoav Etsion, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: Using processors as functional units. In *Hot Topics in Parallelism (HotPar)*, Jun 2010. 10
  - [ERL90] Hesham El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.*, 9:138–153, June 1990. 73
  - [ESD02] M. Ekman, P. Stenström, and F. Dahlgren. TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *Proceedings of the 2002 international symposium on Low power electronics and design*, pages 243–246. ACM, 2002. 24
  - [FC07] G. Fursin and A. Cohen. Building a Practical Iterative Interactive Compiler. In *1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART’07)*, 2007. 8
  - [FHK<sup>+</sup>06] K. Fatahalian, D.R. Horn, T.J. Knight, L. Leem, M. Houston, J.Y. Park, M. Erez, M. Ren, A. Aiken, W.J. Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83. ACM, 2006. 5
  - [FL99] M. Feng and CE Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999. 106
  - [FLA10] FLAME Project. <http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage>, 2010. 5
  - [FT87a] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. 76
  - [FT87b] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987. 61
  - [FVPF95] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of the 1995 European conference on Design and Test*, page 503, 1995. 39
  - [GB03] M. Geilen and T. Basten. Requirements on the execution of Kahn process networks. *Lecture Notes in Computer Science*, pages 319–334, 2003. 48, 58
  - [GG93] R. Govindarajan and GR Gao. A novel framework for multi-rate scheduling in DSP applications. In *International Conference on Application-Specific Array Processors*, pages 77–88, 1993. 59
  - [GGR<sup>+</sup>10] Christian Grothoff, Krista Grothoff, Matthew J. Rutherford, Kai Christian Bader, Harald Meier, Craig Ritzdorf, Tilo Eissler, Nathan Evans, and Chris GauthierDickey. DUP: A Distributed Stream Processing Language. In *IFIP International Conference on Network and Parallel Computing*, Zhengzhou, China, 2010. Springer Verlag. 6
  - [GKN06] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*, January 2006. 17



- 
- [GLB00] S. Girona, J. Labarta, and R.M. Badia. Validation of Dimemas communication model for MPI collective operations. *Proc. EuroPVM/MPI*, 2000. 26
  - [GMA<sup>+</sup>02] M.I. Gordon, D. Maze, S. Amarasinghe, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A.A. Lamb, C. Leger, J. Wong, et al. A stream compiler for communication-exposed architectures. *ASPLOS*, pages 291–303, 2002. 67
  - [GMN<sup>+</sup>08] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. Xstream: a signal-oriented data stream management system. *Data Engineering, International Conference on*, pages 1180–1189, 2008. 6
  - [GNU] GNU Radio. <http://www.gnu.org/software/gnuradio/>. GNU Software Radio Project. 7, 35
  - [GR93] I. Galperin and R.L. Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1993. 102
  - [GR05] Jayanth Gummaraju and Mendel Rosenblum. Stream Programming on General-Purpose Processors. In *MICRO 38: Proceedings of the 38th annual ACM/IEEE international symposium on Microarchitecture*, Barcelona, Spain, November 2005. 39
  - [Gra71] RL Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the November 16-18, 1971, fall joint computer conference*, pages 205–217. ACM, 1971. 77
  - [GSS06] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. S-Net: A typed stream processing language. In Zoltan Horváth and Viktória Zsók, editors, *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06), Budapest, Hungary*, Technical Report 2006-S01, pages 81–97. Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary, 2006. 6
  - [GTA06] M.I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ASPLOS*, pages 151–162, 2006. 40, 44, 54, 64, 67, 88
  - [HCAL89] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18:244–257, April 1989. 73
  - [HCK<sup>+</sup>09] A.H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flexstream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 214–223. IEEE, 2009. 65
  - [HCW<sup>+</sup>10] Amir H. Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Macross: macro-simdization of streaming applications. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 285–296, New York, NY, USA, 2010. ACM. 41
  - [HGG<sup>+</sup>99] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the conference on Design, automation and test in Europe*, 1999. 39

- 
- [HJ03] Tarek Hagra and Jan Janecek. A simple scheduling heuristic for heterogeneous computing environments. *Parallel and Distributed Computing, International Symposium on*, 0:104, 2003. 73
  - [HL91] S. Ha and E.A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *Computers, IEEE Transactions on*, 40(11):1225–1238, Nov 1991. 65
  - [HP07] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007. 1, 65
  - [HPFF93] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 1.1. *Scientific Programming*, 2(1–2):1–170, November 1993. 3
  - [HPFF97] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 2.0. January 1997. 3, 5
  - [HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986. 3
  - [IBM] IBM. <http://www.alphaworks.ibm.com/tech/mtrat>. Multi-Thread Run-time Analysis Tool for Java. 107
  - [IBM09] IBM. *Cell Broadband Engine Programming Handbook including PowerXCell 8i Version 1.11*, 2009. 99
  - [IBM11] IBM. *IBM Streams Processing Language Specification*. 2011. 6
  - [iee99] IEEE Standard for Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API)- Amendment D: Additional Real time Extensions [C Language]. *IEEE Std 1003.1d-1999*, 1999. 5
  - [ILO] ILOG. <http://www.ilog.com/products/cplex/>. CPLEX Math Programming Engine. 65
  - [Int10] Intel. A Quick, Easy and Reliable Way to Improve Threaded Performance: Intel Cilk Plus, 2010. <http://software.intel.com/en-us/articles/intel-cilk-plus>. 5
  - [IP95] K. Ito and K.K. Parhi. Determining the minimum iteration period of an algorithm. *The Journal of VLSI Signal Processing*, 11(3):229–244, 1995. 59
  - [JED10] J.C. Jenista, Y.H. Eom, and B. Demsky. OoOJava: an out-of-order approach to parallel programming. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, page 11. USENIX Association, 2010. 5
  - [Kar78] R.M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete mathematics*, 23(3):309–311, 1978. 59
  - [KET06] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 1176–1188, 2006. 10
  - [Khr] Khronos Group. <http://www.opengl.org/>. 6
  - [Khr10] Khronos Group. *The OpenCL Specification Version: 1.1 Document Revision: 36*, 2010. 3, 5

- 
- [Kie99] B. Kienhuis. Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools. *Delft University of Technology, The Netherlands*, 1999. [40](#)
  - [KL70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970. [50](#)
  - [KL88] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *Software, IEEE*, 5(1):23–32, January 1988. [73](#)
  - [KM<sup>+</sup>72] D.J. Kuck, Y. Muraoka, et al. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, pages 1293–1310, 1972. [3](#)
  - [KM08] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multi-core platforms. *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 114–124, 2008. [42](#), [46](#), [56](#), [57](#), [64](#), [65](#), [67](#)
  - [Koh75] W.H. Kohler. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Transactions on Computers*, 100(24):1235–1238, 1975. [73](#)
  - [KTA03] Michal Karczmarek, William Thies, and Saman Amarasinghe. Phased scheduling of stream programs. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 103–112, New York, NY, USA, 2003. ACM. [115](#)
  - [KTJR05] R. Kumar, DM Tullsen, NP Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005. [23](#)
  - [LA00] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM. [8](#), [41](#)
  - [Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974. [3](#)
  - [LBS] W.I. Lundgren, K.B. Barnes, and J.W. Steed. Gedae: Auto Coding to a Virtual Machine. [6](#), [40](#), [67](#)
  - [LCM<sup>+</sup>05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200, 2005. [98](#)
  - [LDWL06] S. Liao, Z. Du, G. Wu, and G.Y. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–207. IEEE Computer Society Washington, DC, USA, 2006. [67](#)
  - [Lee86] E.A. Lee. *A coupled hardware and software architecture for programmable digital signal processors (synchronous data flow)*. PhD thesis, University of California, Berkeley, 1986. [59](#)
  - [Lee06] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. [2](#)

- 
- [LK78] JK Lenstra and A.H.G.R. Kan. Complexity of Scheduling under Precedence Constraints. *Complexity*, 26(1), 1978. [72](#)
  - [LM87] E.A. Lee and DG Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. [39](#), [59](#), [77](#)
  - [LMT<sup>+</sup>04] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. *13th International Conference on Parallel Architecture and Compilation Techniques*, pages 267–277, 2004. [39](#)
  - [LSB09] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *ACM SIGPLAN Notices*, 44(10):227–242, 2009. [5](#)
  - [M<sup>+</sup>11] Harm Munk et al. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. *International Journal of Parallel Programming*, 39:397–450, 2011. 10.1007/s10766-010-0132-7. [ix](#), [8](#), [16](#), [19](#), [23](#), [121](#)
  - [MAJ<sup>+</sup>09] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez, and A. Ramirez. Parallel scalability of video decoders. *Journal of Signal Processing Systems*, 57(2):173–194, 2009. [89](#)
  - [MAS<sup>+</sup>02] M. Maheswaran, S. Ali, HJ Siegal, D. Hensgen, and R.F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 30–44. IEEE, 2002. [73](#)
  - [Mat02] P.R. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002. [6](#)
  - [Mat04] P. Mattson. *PCA Machine Model, 1.0*, 2004. [40](#)
  - [MB06] Joseph Muscat and David Buhagiar. Connective Spaces. *Mem. Fac. Sci. Eng. Shimane Univ. Series B: Mathematical Science*, 39:1–13, 2006. [44](#)
  - [MIT98] MIT LCS. *Cilk 5.4.6 Reference Manual*, 1998. [106](#)
  - [Moo65] G.E. Moore. Cramming more components onto integrated circuits(Cramming more components onto integrated circuit for improved reliability and cost). *Electronics*, 38:114–117, 1965. [1](#)
  - [MRC<sup>+</sup>07] J. Meng, S. Rohinton, S. Che, J. Huang, J.W. Sheaffer, and K. Skadron. Programming with Relaxed Streams. Technical Report CS-2007-17, University of Virginia, 2007. [6](#)
  - [MTHV04] P. Mattson, W. Thies, L. Hammond, and M. Vahey. Streaming virtual machine specification 1.0. Technical report, Technical report, 2004. <http://www.morphware.org>, 2004. [40](#)
  - [Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. [8](#)
  - [Mur71] Y. Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1971. AAI7121189. [3](#)
  - [Nana] Nanos Group. <http://nanos.ac.upc.edu/content/presenting-nanos>. [20](#), [71](#)

- 
- [Nanb] Nanos project. <http://nanos.ac.upc.edu/content/mintaka-instrumentation-library>. Mintaka Instrumentation Library. 18
- [NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007. 97, 98, 107
- [NVI08] NVIDIA Corporation. <http://developer.nvidia.com/cuda/>, 2008. NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 2.0. 3
- [OH96] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In Luc Boug, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par’96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 573–577. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0024750. 73
- [OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005. 1
- [OIS<sup>+</sup>06] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 45(1):85–102, 2006. 5
- [Ope09] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, May 2009. 5
- [Org08] OpenMP Organization. OpenMP Application Program Interface, v. 3.0, May 2008. 3
- [Par95] T.M. Parks. *Bounded scheduling of process networks*. PhD thesis, University of California, 1995. 48, 58
- [PBL08] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE International Conference on Cluster Computing*, pages 142–151, 2008. 5, 10
- [Pol60] M. Pollack. The maximum capacity through a network. *Operations Research*, pages 733–736, 1960. 61
- [Prv06] M. Prvulovic. Cord: cost-effective (and nearly overhead-free) order-recording and data race detection. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 232–243, Feb. 2006. 107
- [PT03] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Annual International Symposium on Computer Architecture*, volume 30, pages 110–121, 2003. 107
- [RDF98] N. Ramsey, J.W. Davidson, and M.F. Fernandez. Design principles for machine-description languages. *ACM Transactions on Programming Languages and Systems*, 1998. 39
- [Rei] J. Reinders. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. 2007. 5
- [RSL02] M.C. Rinard, D.J. Scales, and M.S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 2002. 5

- 
- [RVVA04] R. Rangan, N. Vachharajani, M. Vachharajani, and D.I. August. Decoupled software pipelining with the synchronization array. In *Parallel Architecture and Compilation Techniques, PACT 2004. Proceedings. 13th International Conference on*, pages 177–188, 2004. [67](#)
  - [SB97] Y. Smaragdakis and D. Batory. DiSTiL: A transformation library for data structures. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL)*, 1997, page 20. USENIX Association, 1997. [6](#)
  - [SBN<sup>+</sup>97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997. [107](#)
  - [Ser09] Serebryany, Konstantin and Iskhodzhanov, Timur. ThreadSanitizer—data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009. [107](#)
  - [SFB<sup>+</sup>09] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics (TOG)*, 28(1):1–11, 2009. [6](#)
  - [SGB06] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd annual conference on Design automation*, pages 899–904, 2006. [67](#)
  - [SL93] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4:175–187, 1993. [73](#)
  - [SN05] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 2. USENIX Association, 2005. [98](#), [106](#)
  - [SP09] Raül Sirvent Pardell. *GRID Superscalar: a Programming Model for the Grid*. PhD thesis, Technical University of Catalonia (UPC), 2009. [5](#), [10](#)
  - [THW99] H. Topcuoglu, S. Hariri, and M.Y. Wu. Task Scheduling Algorithms for Heterogeneous Processors. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, page 3. IEEE Computer Society, 1999. [73](#)
  - [TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *International Conference on Compiler Construction*, 4, 2002. [6](#), [12](#), [39](#), [45](#), [108](#)
  - [TKA<sup>+</sup>10] George Tzenakis, Konstantinos Kapelonis, Michail Alvanos, Konstantinos Koukos, Dimitrios Nikolopoulos, and Angelos Bilas. Tagged procedure calls: Efficient runtime support for task-based parallelism on the cell processor. In Yale Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 307–321. Springer Berlin / Heidelberg, 2010. [5](#)
  - [TKS<sup>+</sup>05] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *Principles and Practice of Parallel Programming*, pages 224–235, 2005. [116](#)

- 
- [TOP] TOP500. <http://www.top500.org/>. 2
- [Ull75] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975. 72
- [Uni09] University of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures*, 2009. 5
- [VDKV00] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):36, 2000. 6
- [vdWdKH<sup>+</sup>04] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzter, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217, 2004. 28
- [VWY07] V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 585–589. ACM New York, 2007. 61
- [WG90] M.Y. Wu and D.D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990. 73
- [WTS<sup>+</sup>97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, et al. Baring It All to Software: Raw Machines. *Computer*, pages 86–93, 1997. 67
- [WY02] Namyoon Woo and Heon Young Yeom. k-depth look-ahead task scheduling in network of heterogeneous processors. In *Revised Papers from the International Conference on Information Networking, Wireless Communications Technologies and Network Applications-Part II, ICOIN '02*, pages 736–745, London, UK, UK, 2002. Springer-Verlag. 73
- [YG94] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1994. 73

---



# Index

acolib, [8](#), [28](#)  
ACOTES, [8](#), [16](#), [23](#)  
ASM, [8](#), [23](#), [48](#), [58](#)  
  
Basten and Hoogerbrugge, [67](#)  
bmod, [10](#), [99](#)  
Brook, [67](#)  
  
connected, [44](#)  
convex, [42](#), [118](#)  
CPLEX, [65](#)  
  
dynamic, [8](#)  
  
fires, [13](#), [44](#), [59](#), [61](#)  
  
Gedae, [40](#), [67](#)  
  
kernel, [6](#)  
KPN, [65](#)  
  
master thread, [5](#)  
multiplicity, [31](#), [35](#), [76](#), [109](#)  
  
Nanos++, [20](#)  
  
OMP Superscalar, [108](#)  
OmpSs, *see* OMP Superscalar  
ORAS, [40](#)  
  
Paraver, [18](#)  
partitioning, [7](#)  
PDG, [70](#)  
prvanim, [18](#), [107](#)  
Ptolemy II, [65](#)  
pycparser, [109](#)  
  
R-Stream, [67](#)  
  
SDF, [39](#), [65](#)  
SDF tool, [67](#)  
SGMS, [42](#), [65](#)  
SPIR compiler, [68](#)  
SPM, [16](#), [44](#)  
Star Superscalar, [9](#)  
  
StarssCheck, [97](#)  
static, [8](#)  
str2oss, [18](#), [108](#)  
stream, [6](#), [12](#)  
StreamIt, [12](#), [108](#)  
StreamRoller, [65](#), [67](#)  
SVM, [39](#)  
  
task, [5](#)  
taskgroup, [17](#)  
tolower, [16](#)  
TTL, [28](#)  
  
unrolling, [7](#)  
  
Valgrind, [98](#)  
  
worker, [5](#)