RA-LPEL: A Resource-Aware Light-Weight Parallel Execution Layer for Reactive Stream Processing Networks on The SCC Many-core Tiled Architecture

Nilesh Karavadara

A thesis submitted to the University of Hertfordshire
in partial fulfilment of the requirements of the degree of

Doctor of Philosophy (PhD)

April 2016

"Be the change that you wish to see in the world."

Mahatma Gandhi

"Life is like riding a bicycle. To keep your balance you must keep moving."

Albert Einstein

I dedicate my dissertation work to my family.
A special feeling of gratitude to my loving parents, Keshav Karavadara and Vali Karavadara who always supported me throughout my studies and never lost their belief in me.
My sister Usha and brother-in-law Bharat, for always being there when I needed them.
My wife Jaya and, daughters Aria and Jia for making me smile every day and provide with unconditional love and great understanding, and also reminding me the reasons why I am following this dream. Also my nieces Ria, Esha and Neera for filling my life with lots of wonderful moments and also the much needed distraction from the work.

Abstract

In computing the available computing power has continuously fallen short of the demanded computing performance. As a consequence, performance improvement has been the main focus of processor design. However, due to the phenomenon called *"Power Wall"* it has become infeasible to build faster processors by just increasing the processor's clock speed. One of the resulting trends in hardware design is to integrate several simple and power-efficient cores on the same chip. This design shift poses challenges of its own. In the past, with increasing clock frequency the programs became automatically faster as well without modifications. This is no longer true with many-core architectures. To achieve maximum performance the programs have to run concurrently on more than one core, which forces the general computing paradigm to become increasingly parallel to leverage maximum processing power.

In this thesis, we will focus on the ***R**eactive **S**tream **P**rogram* (RSP). In stream processing, the system consists of computing nodes, which are connected via communication streams. These streams simplify the concurrency management on modern many-core architectures due to their implicit synchronisation. RSP is a stream processing system that implements the reactive system. The RSPs work in tandem with their environment and the load imposed by the environment may vary over time. This provides a unique opportunity to increase *performance per watt*. In this thesis the research contribution focuses on the design of the execution layer to run RSPs on tiled many-core architectures, using the Intel's ***S**ingle-chip **C**loud **C**omputer* (SCC) processor as a concrete experimentation platform. Further, we have developed a ***D**ynamic **V**oltage and **F**requency **S**caling* (DVFS) technique for RSP deployed on many-core architectures. In contrast to many other approaches, our DVFS technique does not require the capability of controlling the power settings of individual computing elements, thus making it applicable for modern many-core architectures, with which power can be changed only for power islands. The experimental results confirm that the proposed DVFS technique can effectively improve the energy efficiency, i.e. increase the performance per watt, for RSPs.

Acknowledgements

I always believed what Drake said: *"Sometimes it's the journey that teaches you a lot about your destination"*. I could say that my journey through the PhD process has been difficult, enlightening, and rewarding, and it certainly could not have been reached at this stage without the support of a number of people.

Foremost, I would like to thank my principal supervisor and mentor Priv.-Doz. Dr. techn. Raimund Kirner. Without his excellent guidance, encouragement, and support, this thesis would not have been possible. I feel really fortunate to have worked with him and have learnt a lot of things from him, not just related to studies but also useful things for everyday life. Furthermore, my thanks go to my secondary supervisor Prof. Sven-Bodo Scholz. He caught my interest in compilers and runtime systems in the first place and introduced me to the Compiler Technology and Computer Architecture group (CTCA) group at the University of Hertfordshire. Furthermore, I would like to thank Prof. Alex Shafarenko who hosted me in the CTCA group and provided the well-equipped research environment.

I would like to thank all my colleagues in the CTCA group, for all their technical support as well as many parties and fun times together.

I would particularly thank Carl Joslin, Frank Penczek and Vu Thien Nga Nguyen for their support during the starting period of my PhD journey. I am indebted to Michael Zolda for mentoring me. His insightful comments on my research ideas had an important impact on this thesis. I also enjoyed the time we spent in long technical discussions. He always encouraged me and helped me to preserve my motivation during the hard times of debugging and benchmarking the beast (the SCC). Without his guidance and support, this thesis would not have been possible.

During my years at the Science and Technology Research Institute (STRI) I made many great friends, including Htoo (Phothar), Chaminda, Santanu, Simon, Maxim, Saverio, Vincent, Pavel and many more.

The most awaited time of the day for me was the lunchtime with Phothar, Chaminda and Santanu when we would debate anything and everything. I remember sometimes those conversions would stretch over a few days. It was fun and they provided a much needed distraction during stressful times. Thank you all for the good times we shared together. The support staff at STRI deserve a special mention: Lorraine, Kathy, Michaella, and Avis. They cheerfully dealt with every piece of paperwork that I needed.

I also extend my thanks to the Intel Corporation and Intel Labs, for providing me with the opportunity to work on their research processor, called Single-Chip Cloud Computer (SCC). When they decided to shut down the data-centre that hosted the SCC, they agreed to send the system with the SCC to our University. This would not have been possible without hard-work from Raimund, who convinced them to send the SCC system to us so that I could complete my research.

I would also like to thank my friends Hiren Vadukar, Dr Jay Lakhani, Dr Prakash Vadukar, Vijay Vadukar, Chandresh Kishor, Hardash Bhutiya, Hiren Jungi, Faizal Hala, Jitesh Vithlani, Darshak Vithlani, Jagdish Gogra, Jignesh Joshi, Ketan Mehta, Naresh Kyada, Piyush Goswami, Rasendu Pandya, and Tejas Bapodra, who provided their support across continents.

Also, special thanks to Ben Saunders, Rahul Makwana, Snehal Virchand, and Kapil Gohil. I could not thank Rashid enough for helping me out numerous times, no matter what the time of day. I will never forget driving up to Newcastle upon Tyne simply to have a dinner and come back.

My special thanks to Dr Chris Wellings, Edmund Ward, and my superviser Priv.-Doz. Dr. techn. Raimund Kirner for proofreading this thesis.

Last, but not least, I owe my deepest gratitude to my grandparents, my uncles Masri and Bhoja, my aunties Shanti and Hansa, and my cousins Arun, Amar, Sheena, Selina, and Alysha for their support and motivation all along the way.

Contents

List of Figures

List of Tables

List of Code Listings

List of Algorithms

Abbreviations

Abbreviation	Elongated	Page List
HALLOC	**H**ierarchical **Alloc**ator	5, 7, 53, 66, 68, 74–76, 125, 127, 128, 136–140
SAC	**S**ingle **A**ssignment **C**	18, 27
ADC	**A**nalogue to **D**igital **C**onverter	92, 102
AIR	**A**tomic **I**ncrement Counte**r**	73
ALU	**A**rithmetic **L**ogic **U**nit	35
ANSI	**A**merican **N**ational **S**tandards **I**nstitute	13, 14, 18, 27, 31
API	**A**pplication **P**rogramming **I**nterface	5, 127, 128, 134
AST	**A**bstract **S**yntax **T**ree	22
BMC	**B**oard **M**anagement **C**ontroller	102
CCF	**C**lock **C**rossing **F**IFO	40
CMOS	**C**omplementary **M**etal-**O**xide **S**emiconductor	29
CPU	**C**entral **P**rocessing **U**nit	2, 28, 29, 34, 129–131
CRB	**C**ontrol **R**egister **B**uffer	40, 41, 46–48, 55, 60
CRI	**C**ommon **R**untime **I**nterface	22, 23, 26
CTQ	**C**entral **T**ask **Q**ueue	24, 53, 109
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture	13
DDR	**D**ouble **D**ata **R**ate	31, 35, 42
DES	**D**ata **E**ncryption **S**tandard	97, 98, 107, 110, 111, 122
DF	**D**ata **F**etcher	26, 27, 127

Abbreviation	Elongated	Page List
DFS	Dynamic Frequency Scaling	110, 111, 115, 129, 137
DFT	Discrete Fourier Transform	96
DIMM	Dual In-line Memory Module	39
DMA	Direct Memory Access	32, 34
DPM	Dynamic Power Management	29, 131, 132
DRAM	Dynamic Random Access Memory	34, 35, 39, 42
DSA	Dynamic Storage Allocation	65, 66
DVFS	Dynamic Voltage and Frequency Scaling	vii, 3–7, 9, 29, 30, 38, 39, 77, 79, 82, 84, 85, 87, 92, 93, 95, 110, 111, 115, 122, 123, 125, 129, 131–134, 136, 137, 139–142
ED^{2}P	Energy Delay-squared Product	28
EDP	Energy Delay Product	28
EMA	Exponential Moving Average	89, 90
FFT	Fast Fourier Transform	96, 103, 104, 107, 110, 111, 115, 122
FID	Frequency Identifier	130
FIFO	First In First Out	10, 16, 23, 26, 75, 131
FPGA	Field Programmable Gate Array	34, 39, 42, 102, 127
FPU	Floating Point Unit	34, 35
GCU	Global Clocking Unit	40
GDDR5	Graphics DDR type 5	37
GPGPU	General Purpose Graphics Processing Unit	3
GPU	Graphics Processing Unit	1, 3
HPC	High Performance Computing	3, 28

Abbreviation	Elongated	Page List
I/O	Input/Output	22, 31, 34, 35, 50, 88, 132, 133
IA	Intel Architecture	40
IALU	Integer Arithmetic Logic Unit	34
IM	Input Manager	26, 27, 127
IPN	Interprocessor Network	37
ISA	Instruction Set Architecture	43
KPN	Kahn Process Network	10, 11, 14
LMB	Local Memory Buffer	40
LPEL	Light-weight Parallel Execution Layer	23, 24, 26, 27, 53, 55, 59, 74–76, 103, 104, 137
LUT	Lookup Table	40, 41, 43, 46, 47, 55–57, 59, 60, 65, 74–76, 126, 137
MAR	Memory Access Rate	132
MC	Memory Controller	31, 37, 39, 40, 42, 46–49, 59, 60, 87
MCPC	Management Console PC	39, 47, 102
MESH	Memory Efficient Sharing	128
MIC	Many Integrated Core	37
MIT	Massachusetts Institute of Technology	13
MIU	Mesh Interface Unit	40, 41, 45–47
MMU	Memory Management Unit	47
MPB	Message Passing Buffer	40, 42, 43, 46–48, 55, 60, 63, 65, 74–76, 126, 127, 137, 139
MPBT	Message Passing Buffer Type	43–46
MPI	Message Passing Interface	23, 26, 103, 104, 126, 130–133
MPPA	Massively Parallel Processor Array	2, 110, 139, 142
MSR	Model Specific Register	130
MTAP	Multi-threaded Array Processor	35

Abbreviation	Elongated	Page List
MVF	**M**aximal **V**oltage and **F**requency	111, 115
NoC	**N**etwork **o**n **C**hip	6, 7, 9, 30, 31, 35, 50, 51, 111, 127, 132, 133
ODI	**O**n-**D**ie **I**nterconnect	37
OM	**O**utput **M**anager	26, 27, 127
OS	**O**perating **S**ystem	5, 23, 24, 27, 28, 32–34, 38, 43, 55, 56, 72, 74, 76, 125, 127–129, 132, 134, 136, 138, 139
PCIe	**P**eripheral **C**omponent **I**nterface **E**xpress	37, 39
PCL	**P**ortable **C**oroutine **L**ibrary	55
PE	**P**rocessing **E**lement	21, 22, 26, 27, 31, 33, 35, 131
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace	129
PIO	**P**rogrammed **I**nput/**O**utput	35
PMC	**P**ower **M**easurement **C**ontroller	102
POP-SHM	**P**rivately **O**wned **P**ublic **Sh**ared **M**emory	128
POSIX	**P**ortable **O**perating **S**ystem **I**nterface [for Unix]	23, 71, 76
QoS	**Q**uality **o**f **S**ervice	132
RA-LPEL	**R**esource-**A**ware **L**ight-weight **P**arallel **E**xecution **L**ayer	5–7, 87, 93, 95, 103, 104, 109, 110, 123, 135–137, 139–142
RC	**R**untime **C**omponent	16, 23, 27, 28
RISC	**R**educed **I**nstruction **S**et **C**omputer	34, 35
RMS	**R**esource **M**anagement **S**ystem	77

Abbreviation	Elongated	Page List
RSP	**R**eactive **S**tream **P**rogram	vii, 3–5, 7, 9, 15, 16, 22, 50, 51, 55, 65, 68, 77, 80, 81, 84, 85, 87, 93, 102, 110, 125, 127, 133–137, 139–142
RTS	**R**un**t**ime **S**ystem	6, 9, 13, 16, 18, 22, 23, 26, 27, 53, 55, 74, 125, 132, 142
SCC	**S**ingle-chip **C**loud **C**omputer	vii, 4, 5, 7, 9, 23, 31, 38–44, 46–51, 53, 55, 56, 60, 63, 66, 70, 74–76, 78, 87, 88, 92, 93, 95, 102, 103, 110, 115, 123, 125–128, 132, 133, 136–142
SDF	**S**ynchronous **D**ata **F**low	10, 11, 13, 14
SDK	**S**oftware **D**evelopment **K**it	34
SIF	**S**ystem **I**nter**f**ace	39, 46, 49
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata	35
SISO	**S**ingle **I**nput **S**ingle **O**utput	18, 19, 23
SMC	**S**oftware **M**anaged **C**ache-coherence	128
SMP	**S**ymmetric **M**ulti**p**rocessing	33, 38, 139
SPMD	**S**ingle **P**rogram **M**ultiple **D**ata	130
SRAM	**S**tatic **R**andom **A**ccess **M**emory	34, 35, 40, 42
SSE	**S**um of **S**quare **E**rrors	90
SSH	**S**ecure **Sh**ell	138
T&S	**T**est-and-**S**et	41, 70, 76, 139
TCP/IP	**T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol	47
TD	**T**ag **D**irectory	38
TDMA	**T**ime-**D**ivision **M**ultiple **A**ccess	132
TG	**T**raffic **G**enerator	40
TLB	**T**ranslation **L**ookaside **B**uffer	32, 33

Abbreviation	**Elongated**	**Page List**
UID	**U**nique data **Id**entifier	27
VF	**V**oltage and **F**requency	84, 85
VFI	**V**oltage-**F**requency **I**sland	78, 79
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage	13
VID	**V**oltage **Id**entifier	130
VLIW	**V**ery **L**ong **I**nstruction **W**ord	32
VPU	**V**ector **P**rocessing **U**nit	37
VRC	**V**oltage **R**egulator **C**ontroller	39, 47, 49, 88
WCB	**W**rite **C**ombine **B**uffer	40, 44–46, 60

Chapter 1

Introduction

Gordon Moore's conjecture was: *"The number of transistors placed on an integrated circuit doubles approximately every two years"* [101]. This became known as "Moore's Law"; it proved accurate for several decades and fuelled the growth of the computer industry by providing faster processors. And yet the demand for computational power has always exceeded the supply [98]. As an example, the recently released Oculas Rift virtual-reality headset requires approximately 3.5x the rendering power needed to render a game at the resolution of 1080p and frame rate of 60 frames per second. This is due to the fact that the headset requires even higher resolution, higher refresh rate, and 3D processing capabilities. This means that almost all of the ***G****raphics* ***P****rocessing* ***U****nits* (GPUs) which were capable of rendering a game a few months ago are not powerful enough for the Oculas Rift [20].

The trend of shrinking transistors to increase clock speed and build a faster processor worked well, but it meant that with a faster clock speed, more power was needed per unit area. Although one might think that a smaller feature size means less power consumption, the increased clock frequency and higher number of gates have over-compensated this reduction. With the increase in required power per die area came the problem known as *"Power Wall"*, i.e. processors generated so much heat that dissipating this heat efficiently became a problem [8, 35], so much so that it has become infeasible to build faster processors by increasing the processor's clock speed [119]. Furthermore, in the wake of green computing, the power available to a processor is not increasing in comparison to the amount of data that is to be processed. This has moved the industry's focus from raw performance to *performance per watt*. Another contributing factor is the rise of the mobile devices, for which a longer battery life is usually desired.

Physical barriers in making processors faster by increasing the clock frequency and the need for energy-efficient computing have paved the way for many-core computing to become mainstream.

One approach to the problem of reducing power consumption and increasing processing power is to replace one complex and power-hungry core with several simpler but more power-efficient ones [22, 23, 119].

As the heat and interference caused by increased clock speeds and shrinking transistor size are starting to limit processor designs [22, 23], processors such as the Intel Xeon Phi [30], ARM (Cortex A7 & A15) [45], Tilera TILE64 [13], and Kalray MPPA2-256 [66] reflect a trend towards tiled many-core architectures. This trend of increasing the number of cores integrated in a single die is expected to increase steadily in the foreseeable future [22]. In addition to physical barriers, this move to many-core chips is driven by a need to get more *performance per watt*.

This approach comes at a price. In the past, as the clock frequency increased, programs running on these faster processors became faster without any modification. However, this is not true anymore with the many-core architectures. To achieve maximum performance, the programs have to run concurrently on more than one core [119]. While parallel programming has a long tradition in the field of scientific computing, to leverage maximum processing power made available by many cores on the chip, the general computing paradigm has to become increasingly parallel.

This shift to parallel programming puts a burden on programmers, as the identification and exposition of the concurrency becomes the responsibility of the programmer. Furthermore, programmers have to worry about the decomposition and mapping of a computation to the cores to achieve optimal utilisation.

Another issue on many-core architecture is data consistency. Most current multi-core ***C**entral **P**rocessing **U**nits* (CPUs) are organised around cache-coherent shared address space, in which memory consistency is maintained by the hardware on behalf of the programmer. However, in the Shared Memory Multiprocessor systems the memory consistency is usually handled by the hardware, which incurs some protocol overhead. This protocol overhead can vary, based on the used cache-coherency mechanisms [92, 122]. As we add more cores to the chip, the protocol overhead becomes a bottleneck and all gain from adding more cores is lost. The problem comes from the fact that any technical implementation that exists today is based on the idea of caches. In order to have a consistent view of memory across all cores it is necessary to invalidate cache lines which are modified by other cores. The technologies that exist today are not scalable for thousands of cores [92, 122]. As a solution, one approach is to build a processor with no support of hardware-based cache-coherency; this seems to scale well with hundreds or even thousands of cores.

The programming difficulty and scaling are like two sides of the same coin. Now we have architectures which can scale to hundreds or even thousands of cores, but we are facing increasing difficulties in writing programs that run efficiently on them. Moreover, in order to achieve optimal performance on such an architecture the programmer needs knowledge of the underlying hardware. As an example, ***G**eneral **P**urpose **G**raphics **P**rocessing **U**nit* (GPGPU) provides many computation cores, but in order to get maximum performance the programmer has to know how and when to move the data correctly from the main memory of the computer to the various memories of the GPU hardware. Furthermore, the programmer has to take care of load balancing, i.e. all cores taking part in a computation should have a nearly equivalent workload. Otherwise, some cores will remain idle while others are processing data, increasing the wall clock time that the program takes to complete [119], wasting resources and increasing energy consumption as a result.

Effectively programming many-core architectures is currently the domain of experts. However, one programming paradigm that addresses the complexity involved in achieving maximum processing power from such architecture is *stream processing*.

Under the stream processing paradigm, programs are constructed by computational nodes connected by unidirectional communication channels. Each compute node can be executed as soon as data is available on its input streams. Streams are communication channels to transfer sequences of data among computation nodes. These streams simplify the concurrency-management on modern many-core architectures due to their implicit synchronisation.

In this thesis, we will focus on stream programs that continually respond to external inputs and process virtually infinite sequences of data. We refer to such stream programs as ***Reactive Stream Programs*** **(RSPs)**, to differentiate them from general stream programs, where the programs respond to rather small and finite input data. An example of RSP is real-time video encoding, where the encoder needs to process incoming video frames as they arrive.

The RSPs work in tandem with their environment, where the load imposed by the environment may vary over time. This provides a unique opportunity to increase *p*erformance per watt. Optimisation of power consumption is important, especially if RSPs are used in resource-constrained environments.[1] For example, if the system load imposed by the environment varies over time, the use of dynamic power management techniques like ***D**ynamic **V**oltage and **F**requency **S**caling* (DVFS) can be used to effectively reduce the power consumption of such systems. Expanding the example of real-time video encoding, the computational resources needed vary greatly depending on the captured scenery: detailed sceneries with fast, complex movements are much

[1]Although even for ***H**igh **P**erformance **C**omputing* (HPC) the power consumption is a huge problem.

more computationally intensive than still scenes with few details. The challenge in dynamic power management for RSPs is to save power at the right time, without violating the system's throughput and latency constraints. This is a hard problem, because the future resource requirements can be unpredictable for many important applications.

In this thesis, we present an execution layer that provides light-weight DVFS support for RSPs. We target the ***S**ingle-chip **C**loud **C**omputer* (SCC) many-core processor as a concrete example to evaluate our approach. The SCC is a research processor from Intel. It provides 48 cores and a flexible infrastructure for DVFS. Compared to other commercially available tiled many-core architectures, it supports more independent on-die voltage and frequency domains. Moreover, since the SCC is based on IA-32 cores, a legacy benchmark code can be easily ported.

1.1 Research Questions

Although RSPs are in general well suited for many-core architectures, they work in tandem with their environment, in which the load imposed by the environment may vary over time. Keeping this in mind, this thesis is motivated by the following research question:

> *Is it possible to improve the adaptive resource utilisation and improve the energy efficiency of RSPs on many-core platforms by exploiting knowledge about the states of the system?*

This question is split into the following sub-questions:

1. What is an efficient way to port an existing RSP execution layer to the SCC?
2. What are meaningful performance-indicators to identify the workload situation of an RSP on a many-core processor?
3. Is it possible to have these performance-indicators to be independent of any specific hardware feature of that many-core processor?
4. What are the adequate strategies to optimise the performance per watt of RSPs on many-core platforms?
5. Is it possible to design DVFS strategies that are light-weight and simple, but still adequate to provide substantial reduction in energy consumption of RSPs?

1.2 Contributions

This thesis will make the following contributions, aiming to answer the proposed research question:

1. The development of the first RSP execution layer for the many-core processor SCC that retains the shared memory programming model.[2]

2. We present the ***H**ierarchical **Alloc**ator* (HALLOC), a novel hierarchical memory creation and management scheme for the SCC.

3. We discuss and identify performance metrics to identify the workload situation, which can be used to develop DVFS strategies for RSPs on many-core processors. These metrics do not depend on any specific hardware features of the many-core processor.

4. We present ***R**esource-**A**ware **L**ight-weight **P**arallel **E**xecution **L**ayer* (RA-LPEL), a resource-aware execution layer for RSPs on many-core architectures.

5. We evaluate the effectiveness of our DVFS method for RSPs in terms of improvement of energy efficiency.

Contributions 1 and 2 answer the first research sub-question, i.e. finding an efficient way to port an existing RSP execution layer to the SCC. We have developed the so-called HALLOC memory manager, which provides a way to create the illusion of a shared memory on the SCC. Furthermore, it also hides the cumbersome details of allocation and deallocation from the end-user, offering a simple ***A**pplication **P**rogramming **I**nterface* (API) to work with. The main distinction of HALLOC—that separates it from multi-threaded memory allocators—is that it is not simply a memory manager; instead it is a complete mechanism that creates the shared memory—the instance of an ***O**perating **S**ystem* (OS) running on the SCC is not aware of the existence of this shared memory—at the application level and provides functionality to manage this memory.

Contribution three addresses both the second and third research sub-questions, since they are intertwined. We consider the throughput and available resources as major indicators of performance of the RSPs. Furthermore, the available resources and throughput can be considered to be independent of hardware architecture.

[2]There has been before a successful attempt to port a stream-processing environment to the SCC [135]. This approach was made for a concrete coordination language, called S-Net. Our execution layer is not bound to a particular coordination language, and did not require refactoring the code for the sake of avoiding shared memory communication.

Contributions 4 and 5 answer the fourth sub-question by introducing the RA-LPEL and by experimentally evaluating the different DVFS policies on the actual hardware. We report a substantial reduction in energy consumption.

1.3 Publications

Most of the work in this thesis has been published in the following papers:

- Nilesh Karavadara, Simon Folie, Michael Zolda, Nga Nguyen, Raimund Kirner, *“"A Power-Aware Framework for Executing Streaming Programs on Networks-on-Chip”*, In Proc. International Workshop on Performance, Power and Predictability of Many-Core Embedded Systems (3PMCES), Electronic Chips and Systems Design Initiative (ECSI), Dresden, Germany, March 2014. doi: 10.13140/RG.2.1.1684.4400. Available at http://dx.doi.org/10.13140/RG.2.1.1684.4400

- Nilesh Karavadara, Michael Zolda, Vu Thien Nga Nguyen, and Raimund Kirner, *“A Hierarchical Memory Management for a Load-Balancing Stream Processing Middleware on Tiled Architectures”*, In 18th Workshop on Programming Languages and Foundations of Programming (KPS’15), Technische Universität Wien, Pörtschach, Austria, October 2015. Available at http://www.complang.tuwien.ac.at/kps2015/proceedings/KPS_2015_submission_50.pdf

- Nilesh Karavadara, Michael Zolda, Vu Thien Nga Nguyen, Jens Knoop, and Raimund Kirner, *“Dynamic Power Management for Reactive Stream Processing on the SCC Tiled Architecture”*, EURASIP Journal on Embedded Systems, 2016(1):1–17, June 2016. ISSN 1687-3963. doi: 10.1186/s13639-016-0035-9. Available at http://dx.doi.org/10.1186/s13639-016-0035-9

1.4 Structure of the Thesis

The remainder of this thesis is organised as follows:

Chapter 2 provides the necessary background in the context of this thesis. This includes stream programming with details on models, languages, and properties of stream programs. Furthermore, we describe the execution model for S-Net language. This includes the description of the S-Net language, followed by the detailed description of the compiler, the ***R**un**t**ime **S**ystems* (RTSs), and the execution layers for the S-Net. This chapter also covers some aspects of power and the energy optimisation for computers in general. An overview of the ***N**etwork **o**n **C**hip* (NoC) architectures is also provided with examples. In addition to the S-Net, we also provide a detailed description of the Intel's experimental research processor, the SCC. S-Net serves as a programming language for the RSPs and the SCC is used as a research vehicle for the experimental evaluation of our approach in this work.

Chapter 3 describes HALLOC, a novel hierarchical shared memory creation and allocation mechanism that forms part of the execution layer. It includes design criteria and a detailed implementation explanation of HALLOC. In addition, we will describe the execution layer initialisation procedure on the SCC hardware.

Chapter 4 covers definitions of workload and resource usage situations in general. In addition, it also provides a formal definition of load ranges for RSPs with different arrival rate, and then extends them by taking into account the impact of DVFS.

Chapter 5 discusses RA-LPEL, an execution layer with resource awareness for RSPs. This chapter covers the heuristics used in our experimental evaluation to adjust power consumption. Furthermore, we also describe various DVFS policies. In addition, we introduce guidelines which governs when to employ DVFS based on system state.

Chapter 6 starts by introducing a set of use cases of RSPs. These use cases are used as experimental benchmarks to evaluate efrficiency of our DVFS policies its impact on RSPs. Furthermore, experimental results are reported and analysed, with a focus on performance and energy optimisation.

Chapter 7 discusses the related work of this thesis, focusing on three main areas: another approach that targets the SCC with S-Net, memory-management, and energy optimisation. The related work in memory-management area covers approaches that focus on providing the shared memory-management functionality for NoC, while the related work for power optimisation covers approaches that use DVFS to reduce energy consumption.

Chapter 8 summarises the work. It also discusses future research directions.

Chapter 2

Background

This chapter provides the necessary background for the rest of the thesis. In § 2.1 the programming paradigm on which we are focusing in this work is discussed. Introduction to the stream programming model is covered in § 2.1.1 and in § 2.1.2 we study the properties of the stream programs. An overview of different stream languages is provided in § 2.1.3, followed by an account of ***R**eactive **S**tream **P**rograms* (RSPs) in § 2.1.4. Detailed description of the execution model of RSP with a concrete example of S-Net language is provided in § 2.2. For S-Net we provide an in-depth overview of the language in § 2.2.1, while the compiler, the ***R**untime **S**ystems* (RTSs), and execution layers are covered in § 2.2.2, § 2.2.3 and § 2.2.4 respectively. A brief study of the power- and energy-optimisation is presented in § 2.3. We review some of the examples of the multi-core/many-core ***N**etwork **o**n **C**hip* (NoC) architectures in § 2.4. Finally, § 2.5 describes Intel's ***S**ingle-chip **C**loud **C**omputer* (SCC), an experimental processor with fine-grained ***D**ynamic **V**oltage and **F**requency **S**caling* (DVFS) capabilities. In the context of this thesis we use the S-Net and the SCC as a research vehicle to conduct experiments.

2.1 Stream Programming

The philosophy behind stream/data-flow programming provides a drastically different way of looking at the computation from the von Neumann architecture with control-flow. In control-flow, the focus is on how control moves through the program and the order of the computation. In data-flow, the focus is on how data moves through the program and the availability of the data. With data-flow style scheduling becomes the responsibility of the system, and not of the programmer.

In data-flow programming a program is modelled as a directed graph of the *operator nodes* representing computations, connected by *arcs* representing data path. In contrast to control-flow paradigm, the activation/execution/firing of the operator node is dependent on the availability of the data on its input arcs. For readers interested in broadening their knowledge of the field of the data-flow programming, we list some surveys as invaluable resources [2, 63, 123, 139].

Similar to the data-flow programming paradigm, a program in stream programming is described by a directed graph in which the operator nodes denote operations and the arcs connecting them denote data dependencies between operations. For convenience, from now on we will refer to operator nodes simply as *nodes*. In the stream programming, the data arcs are represented as *streams*. Now what do we mean by *streams*? According to Lee and Parks there exist different definitions of the term *stream* in the literature [88]. One definition, for example, from Landin [84] and Burge [25], describes streams with recursion and lazy semantics. In this definition a stream is composed of two parts: first is the value of the head of the stream and the second is the procedure, which when evaluated produces the rest of the stream. According to another definition by Franco et al. [39] and by Dennis [34], streams are regarded as channels, or an infinite—or finite with unknown quantity—sequence of elements. In the context of this thesis, we use second definition, with which streams are regarded as channels.

2.1.1 Stream Programming Models

With the introduction of the ***K**ahn **P**rocess **N**etwork* (KPN), Kahn's main intention was to model concurrent systems, but the model has proved to be a convenient way to model signal processing systems as well [64, 65]. In KPN, the operator node is called a *coroutine* or *process*, while stream is identified as *channel*. The processes of a KPN are deterministic and can be modelled as a sequential program, i.e. for the same input they always produce exactly the same output. The channel is an unbounded ***F**irst **I**n **F**irst **O**ut* (FIFO) queue. Since the channels are unbounded and processes are deterministic, the resulting KPN does not depend on the computation or communication delays and exhibits deterministic behaviour. Processes read/write atomic data elements called *tokens* from/to channels. Since channels are unbounded, write operation is non-blocking, i.e. a process can write a token to the output channel at any time. In contrast, a reading operation is blocking, i.e. a process becomes blocked when reading from an empty input channel.

Since KPN channels are unbounded, a buffer overflow or even deadlock is possible. It is difficult to schedule a KPN because of the need to balance relative process rates. In 1987, Lee and Messerschmitt proposed ***S**ynchronous **D**ata **F**low* (SDF), a restricted version of KPN [86, 87]. When a process in the SDF is invoked,[1] it consumes/produces a fixed number of tokens from each of its input/output channels. For an SDF graph, the rate at which a node consumes/produces tokens is known a priori. This makes the number of tokens produced/consumed independent of the input data. In order to avoid deadlock, each channel in SDF has an associated property called *delay*, which corresponds to a sample offset between the input and the output. A channel with some delay d means that the first d number of tokens in that stream is not produced by a node, but is part of the initial state of the program. With the introduction of delay and knowledge of the token rate, SDF does not require unbounded channels as KPNs and it is guaranteed to have a static schedule.

2.1.2 Properties of a Stream Program

According to observations [103, 123] stream programs can be classified by properties of their nodes and streams, such as:

Stream Communication Type Streams can be either *uni-directional* or *bi-directional*. In the case of the former, data travels in only one direction, while in the case of the latter data can travel in either direction. For example, let's look at the two nodes A and B connected by stream S. In the case, when S is uni-directional only one node can write to the stream while another node can only read from it. If stream S is bi-directional, both nodes A and B can read *and* write from/to the same stream.

Node Computation Type Computation performed by a node can be *functional* or *non-functional*. A functional node F with the same input I always produces the same output O; there is no state that can change this. A non-functional node NF with the same input I, may produce a different output, depending on other factors, e.g. the internal state of the node.

Node Computation Behaviour The behaviour of a node can be *constant* or *variable* through the execution. For example, does the processing time for messages stay constant or does it change based on the input value of the message? Another example is *multiplicity*[2], which stays constant regardless of change in the input message or the node's internal state in case of constant behaviour, whereas it would change if the node behaviour was variable.

[1]Hereafter, we will use the term *invoke* to describe the action when a node performs its computation once. This action is also identified as *execution*, *firing* or *activation*.

[2]Multiplicity of a node is the ratio of the number of input messages to the number of output messages per node invocation. A node with multiplicity of n-to-m, consumes n messages from its input stream and produces m messages on its output stream on each invocation.

Inter-node communication The communication that takes place over the stream between two nodes can be *synchronous* or *asynchronous*. If the communication is synchronous then there has to be a mechanism in place that would support synchronisation between the transmission and reception of the messages, i.e. a global clock. In case of asynchronous communication, there is no synchronisation taking place between transmission and reception, and as such a notion of time does not exist for communication purposes between nodes. In this case, reading and writing of the messages are proceeded independently.

Program Structure The structure of the program can be *static* or *dynamic*. For a program with property of static structure, the graph of the program in terms of arrangement and the number of nodes and the relationship between them by means of stream connection would not change. For a program with dynamic structure, the number of streams, the number of nodes, and the connection between them can change dynamically during the execution lifetime of a program.

2.1.3 Stream Programming Languages

As noted in [123] a number of programming languages have integrated the concept of streams. An exhaustive listing of the features of all the streaming languages can be very long. Bearing that in mind, we will give a brief overview of some of the notable languages here.

Each language introduces its own terminology to describe components of the language, some of the examples are:

The graph representing the application contains the vertices representing computation and edges representing connection between them, also denoted as the *network* or an *execution plan*

Vertices representing computation are also denoted as *operator node*, *operator*, *node*, *function*, *kernel*, *filter*, *component*, or *box*

Edges representing connection are also referred to as *channel*, *data path*, *stream*, or *data arc*

The action performed by a node is denoted by terms *invocation*, *execution*, *firing* or *activation*

A single unit of data travelling on a stream can be identified as *envelope*, *message*, or *record*

The functionality of Bobolang [36] e.g. basic operators and data types, is implemented and compiled natively in the underlying systems, e.g. in C++. The streams are uni-directional, while the program structure is dynamic. The data is processed by *operators*. The operators are made up of two parts: The first part contains strictly serial computation called *sub-operators*, i.e. it should not be possible to decompose it any further. The second part specifies how all the sub-operators of an operator are connected. Apart from user-specified sub-operators, there are two implicit special sub-operators that define the input and the output for the operator. Each operator may have multiple inputs or outputs, which in turn allow operations like gather and scatter. The technical documentation presents a detailed description of the language syntax [12].

The application development environment called *Auto-Pipe* [27] is geared towards simplifying the development of complex streaming applications for hybrid architectures. In order to achieve its goal, Auto-Pipe uses a coordination language called *X* [40] to connect kernels written in traditional languages in a data-streaming style, i.e. X can express the nodes and edges of a task-processing graph, as well as the configuration, platform definition, and device bindings. X allows representation of a program in terms of coordination and communication of kernels written in a different computation language. Since Auto-Pipe is targeting hybrid architecture, each kernel may have several platform-specific implementations, e.g. ***A**merican **N**ational **S**tandards **I**nstitute* (ANSI) C, ***C**ompute **U**nified **D**evice **A**rchitecture* (CUDA), and ***V**HSIC **H**ardware **D**escription **L**anguage* (VHDL). All platform-specific implementations of a kernel are required to provide the same interface and streaming data semantics, e.g. input/output ports and data types. This in turn guarantees the correctness of a program, regardless of where each kernel is mapped. The runtime of X is responsible for making sure that the data arrives at the correct kernel, even if it is executed on a different platform. In addition to language X, the Auto-Pipe environment has *X-Com*, a compiler that takes in the application's specification of computation, resources, and a topology of these resources to generate a set of source files that can be compiled for each device in the system. The *X-Dep*, an application deployment tool, is used to deploy the application on the target hardware.

StreamIt [127] from ***M**assachusetts **I**nstitute of **T**echnology* (MIT) is believed to be the most popular streaming language based on the ideas of the SDF in the research community. StreamIt is accompanied by an optimising compiler and a RTS. Since StreamIt employs the SDF model, it requires the kernel to have static input and output rates. That is, the number of items consumed on the input stream and produced on the output stream must be constant, from one invocation to the next. Every StreamIt program is a hierarchical composition of three basic stream structures, called

Pipeline, *SplitJoin*, and *FeedbackLoop*. Although StreamIt is based on SDF, which requires synchronous communication between kernels, it also has a distinct feature called *teleport messaging* the ability to send control messages to other kernels as an asynchronous method call. These control messages are used to change the behaviour of the receiving kernel. In addition, as these control messages do not use the same infrastructure as data items, they can be communicated to the kernels independently of the data-flow graph of an application. To support node synchronisation, StreamIt introduces the conception of *information flow*, in which messages can carry timing information when transferred over streams [128]. This is required as all the kernels in StreamIt execute independently, without any notion of global time. Since StreamIt is based on the SDF model, it only supports a static schedule. This in turn allows the compiler to perform enough analysis and optimisation to produce an efficient implementation.

S-Net is a declarative coordination language that aims to support the transition from sequential code to parallel code, where the concurrency handling is completely managed by S-Net [49]. The kernels are implemented in an independent computational language, e.g. ANSI C. The streams in S-Net are uni-directional. In contrast to StreamIt, the program structure in S-Net can be changed dynamically at runtime and S-Net provides asynchronous communication over streams. Furthermore, there is no restriction on the rate of consumption/production of messages on the streams. Compared to StreamIt, S-Net is more general and closer to KPNs. We will cover S-Net with a more detailed description in Section 2.2.

2.1.4 Reactive Stream Programs

Harel and Pnueli [56] separated the computing systems into two classes: *transformational systems* and *reactive systems*. In transformational systems, a computer program is considered like a black box that accepts inputs, produces outputs by performing transformations on the inputs, and terminates. Transformational systems can be completely described as relations between their inputs and outputs. In contrast, only the relation between inputs and outputs *cannot* describe reactive systems completely. Reactive systems continuously respond to external inputs, i.e. a reactive system maintains an ongoing relationship with its environment by continuously interacting with it.

According to Berry [18] reactive systems can be broken down into two categories based on their relationship with its environment: *reactive systems* and *interactive systems*. The systems that continuously respond to external stimuli produced by their environment at the same speed at which the environment produces it are reactive systems, i.e. the environment decides when to produce the stimuli and the system

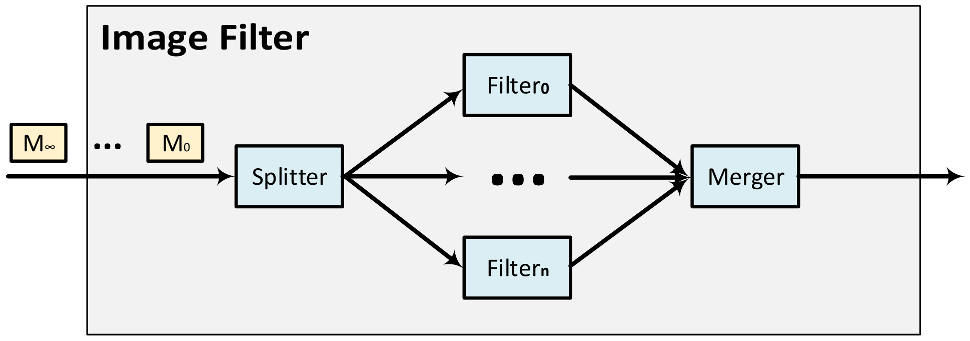


Figure 2.1 Program Structure of the Image Filter Application

responds to it. In contrast, interactive systems respond to external stimuli produced by their environment at their own speed.

The distinction between reactive and interactive systems is based on how they respond to stimuli from their environment, i.e. it is based on a program's semantics, although there is no structural difference between a reactive and interactive system. Taking this into account, we use the same definition of a reactive system as in Harel and Pnueli [56], which covers both interactive and reactive systems and takes no account of the way they respond to their environment.

We use the term ***R**eactive **S**tream **P**rogram* (RSP) as it is by Nguyen [103] to describe programs that fit into the category of a reactive system based on the definition above and designed as a stream program, i.e. RSPs are **stream programs** that continuously **react** to potentially never-ending stimuli from their environment, in contrast to a stream program with a rather small and finite input.

We will use image filter application as a running example of an RSP. Figure 2.1 shows the structure of an image filter application. The application includes a node *Splitter* that reads messages (possibly infinite) containing images from the environment and splits them into messages containing sub-images. The messages with sub-images are sent to different branches where *Filter* nodes perform the actual filtering operation. Messages containing the filtered sub-images are then sent to the *Merger* node, which combines them into complete images and sends them out to the environment.

As we know in RSPs, data arrives from the environment as a virtually infinite sequence of messages. Nodes that receive messages from the environment are called *entry nodes* and the input messages of these entry nodes are called *external input messages*. In our example, node Splitter is an entry node. In similarity to entry nodes, nodes sending messages to the environment are *exit nodes*. The messages produced by exit nodes are called *external output messages*. The node Merger is an exit node, as can be seen from Figure 2.1. The nodes that are between entry and exit nodes are called *intermediate nodes* and, as such, all the messages inside the RSP are referred to as *intermediate messages*, i.e. the messages that are not external input/output messages. In our exemplary image filter application, nodes $Filter_0$ to $Filter_n$ are intermediate nodes.

Conceptually, the execution model of an RSP includes three layers: a compiler, an RTS and an execution layer. The compiler parses the source code of an RSP and generates the object code.

The RTS enforces the RSP's semantics and uses the object code to allocate runtime objects. In terms of enforcing semantics, the RTS makes sure that each ***Runtime Component*** (RC) reads from and writes to appropriate streams. The runtime allocation of the objects includes, but is not limited to, the FIFO buffers that represent streams and RC that represents an instance of a node or an operator.

The execution layer below the RTS, transforms RCs into executable objects called *tasks* and provides an implementation of FIFO buffers to transfer messages between these tasks. The execution layer also provides a scheduler to distribute tasks to physical resources. In simple terms, a task is an iterating process that reads messages from its input streams, performs the associated node's computations, and writes output messages to its output streams. Each iteration of a task is called *RC invocation*. The term *multiplicity* defines the ratio of the number of input messages to the number of output messages per RC invocation. A task with multiplicity of n-to-m, consumes n messages from its input stream and produces m messages in its output stream on each invocation.

2.2 Execution Model of RSPs with S-Net as an Example

As we already know, conceptually the execution model of an RSP includes three layers: a compiler, an RTS, and an execution layer. We use S-Net as a reference language and describe its compiler, RTS, and execution layer. The execution model of S-Net is illustrated in Figure 2.2.

2.2.1 S-Net Language

S-Net [49] is a declarative coordination language. The main design principle of S-Net is to separate computations from concurrency management aspects. S-Net is a pure coordination language, as it does not provide the features to express computations but offers a notation for describing data dependencies in computation. In S-Net the computational logic is encapsulated inside the individual stateless computation components called *boxes*. S-Net relies on auxiliary language to implement these boxes. In principle, any conventional programming language can be used to implement boxes.

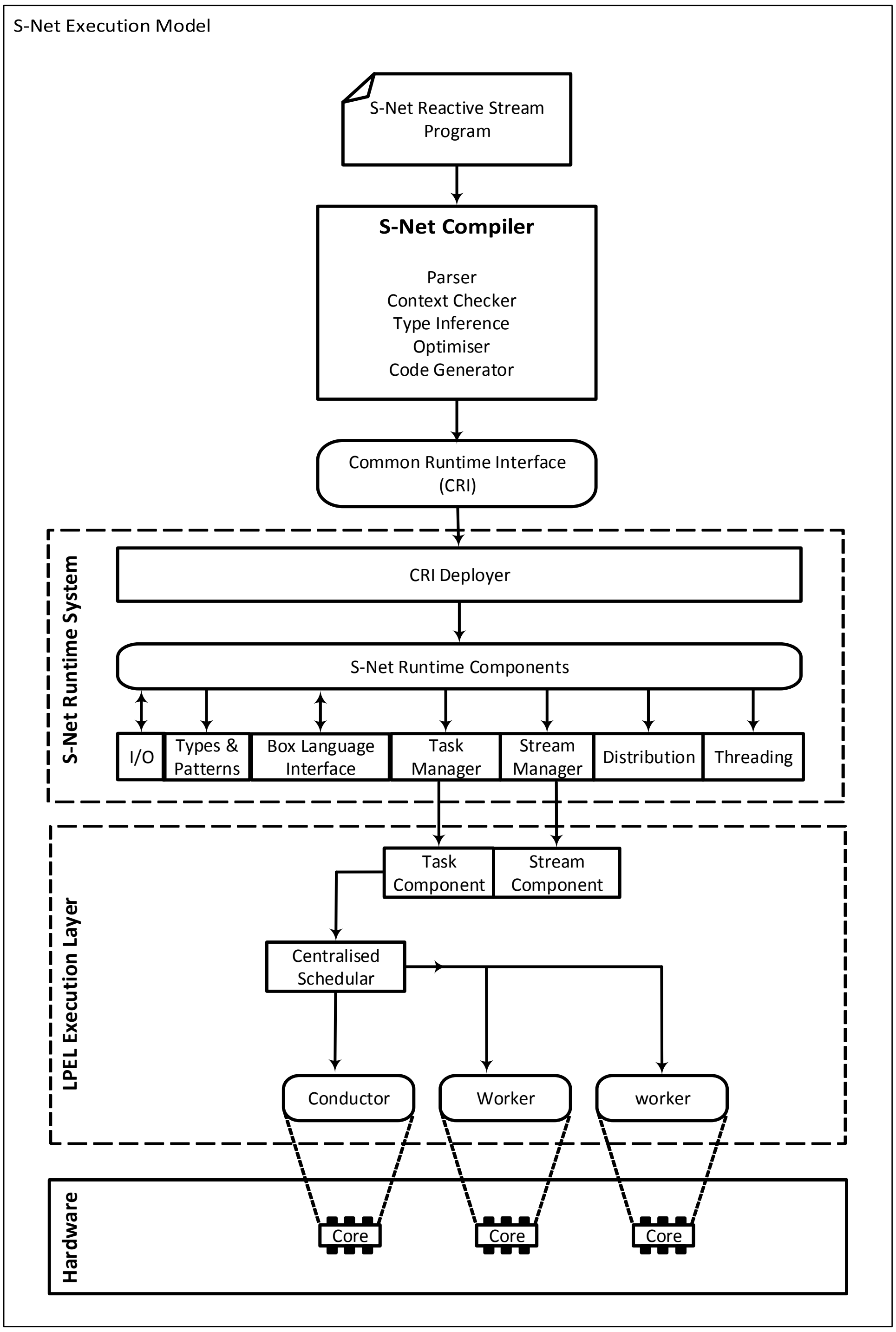


Figure 2.2 S-Net Execution Model Overview [103]

```
net imageFilter ({image}→{filteredImage})
{
  box Splitter((image) → (subImage, <branch>));
  box Filter((subImage) → (subImage));
        box Merger((subImage, <branch>) → (filteredImage));
} connect
        Splitter .. Filter!<branch> .. Merger;
```

Listing 2.1 S-Net Implementation of the Image Filter Application

Currently, interfaces for a subset of ANSI C and ***S**ingle **A**ssignment **C*** (SAC) [48] are provided to be used as a box language. The boxes are connected by streams that facilitate communication between them.

The term *network* is used to describe an application in S-Net with multiple boxes connected via streams. Each box in S-Net is a ***S**ingle **I**nput **S**ingle **O**utput* (SISO) component, i.e. they have a single input stream and a single output stream. These two streams connect a box to the rest of the network. The SISO components are either boxes or previously constructed networks. Large and complex networks can be constructed by combining these SISO components together. Any network, irrespective of its size and complexity, again is a SISO component.

Listing 2.1 shows the S-Net code of our running example of image filter application. In the S-Net, data in the form of the *message* travel through the streams. Each message is comprises a set of label-value pairs. There are two types of labels called *fields* and *tags*. Fields are completely opaque to the S-Net RTS and it gets manipulated only inside boxes, as they are associated with values from the box language domain. In contrast to fields, tags are integer numbers and accessible by both the S-Net RTS and the user-defined boxes. To differentiate the tag and the field, the tags are enclosed within angular brackets. In our running example of an image filter, <*branch*> is a tag while *image*, *subImage* and *filteredImage* are the fields.

The behaviour of the network and box in S-Net is declared by a *type signature*: a mapping from a single input type to one/multiple output type. In our example, node Splitter accepts messages with an image as an input and produces messages that contain the *subImage* as a field and <*branch*> as a tag. Let's look at a more complex example;

```
box foo ((a, <b>) → (c) | (c, d, <e>)
  );
```

The box *foo* accepts messages with field *a* and tag <*b*> and produces messages with either field *c* or with fields *c*, *d* and tag <*e*>.

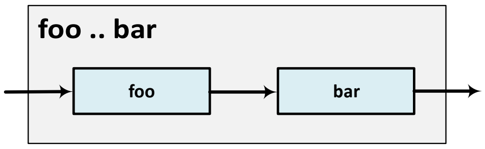


Figure 2.3 Serial Composition

To loosen the restriction of a single input type, S-Net supports an inheritance mechanism, called *flow inheritance*. This allows a box to accept all sub-types of the box's declared input type. Excess fields and tags of a message are bypassed through the box. That means that when a message arrives to a box, only entries with listed labels in the input type are taken by the box to generate output messages. These additional fields and tags are then added to each message emitted by the box in response to the input message if the output message does not contain the field and tags of the same name already. This means box *foo* receives any input records that have at least field *a* and tag <*b*>, but also records with further fields and tags. In our running example, the node Splitter produces messages with type $(subImage, <branch>)$, whereas the next node Filter accepts messages with the type $(subImage)$. In this case, tag <*branch*> is bypassed and added to the output messages that are sent to node Merger.

In addition to the boxes implemented in the box language, there are two built-in types of boxes in the S-Net, the *filter box* and the *synchrocell*. The filter boxes support simple operations on the messages including adding or removing fields, splitting messages, and simple computations on tags. Since boxes/networks in S-Net are SISO, if a box/network requires data from several messages as input, these messages have to be merged first. S-Net provides synchrocell to do just that. A synchrocell is parameterised over the type of message that it is supposed to merge. As soon as it receives messages of all matching types, it releases a single combination of these messages. After producing the merged message, the synchrocells serve as an identity function, forwarding all incoming records.

In order to construct a network composed of boxes that represents a stream program, S-Net provides five network combinators.

Serial composition (denoted as **..**) and parallel composition (denoted as **|**) allow the construction of pipelines and branches respectively. Both combinators are static, in the sense that only one instance for each of their operands is created. As can be seen in Figure 2.3, serial composition connects two components serially. For example, in $foo..bar$ serial composition connects the output of operand foo to the input of operand bar. Parallel composition of $foo|bar$ connects foo and bar in parallel. This can be seen in Figure 2.4. The message is routed to the operand network that best matches its type [49] by parallel compositor.

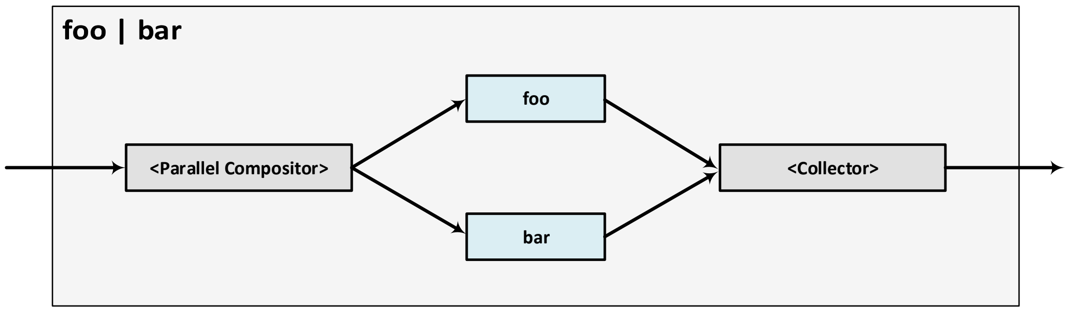


Figure 2.4 Parallel Composition

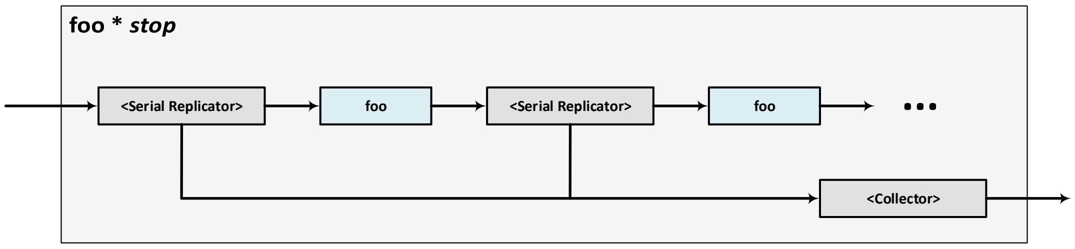


Figure 2.5 Serial Replication

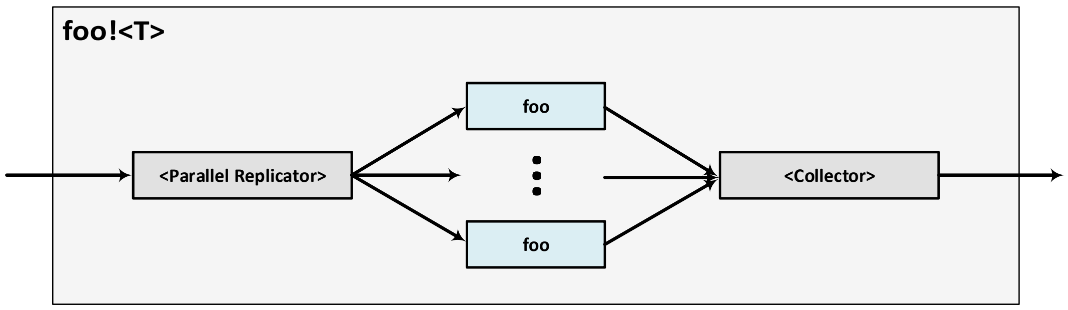


Figure 2.6 Parallel Replication

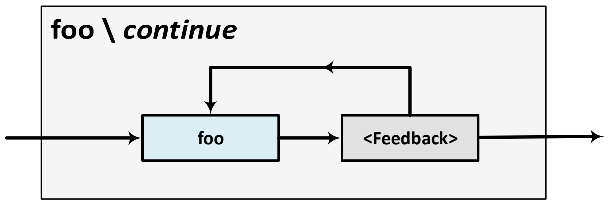


Figure 2.7 Feedback Loop

The next two combinators are dynamic, in the sense that they create replicas of their operands on demand. Serial replication (denoted as $\star$) and parallel replication (denoted as **!**) allows construction of pipelines and branches of dynamic lengths respectively. The serial replicator in $foo \star stop$ will replicate the operand network foo and join then serially until the exit pattern $stop$ is met. Serial replicator is illustrated in Figure 2.5. Parallel replicator in $foo!{<}T{>}$ as depicted in Figure 2.6 will create instances of its operand foo for every unique tag value $<T>$ and combine them in parallel. Each message arriving to parallel replication is processed by only one of these instances. The tag value $<T>$ will determine which instance will process the message and then message is routed accordingly.

The feedback combinator (denoted as $\backslash$) is similar to a serial replication combinator as it takes one operand, but instead of an exit pattern it takes a continue pattern. If the message on the output stream matches the continue pattern, it is sent back as an input message to that operand. Figure 2.7 shows the feedback combinator for $foo\backslash continue$, where a feedback loop around the operand foo is created. All the messages matching the continue pattern $continue$ will be sent back to foo as input messages.

Only serial composition preserves message order, while others do not due to the fact that messages travel through different branches. In order to allow preservation of the message order, S-Net provides deterministic variants of these combinators. They are denoted as $||$ for parallel composition, $\star\star$ for serial replication, and !! for parallel replication.

To support stream programs on a distributed system, the S-Net language is extended with a concept of nodes and placement [50, 51]. A node is an abstract location—representing a ***Processing Element*** (PE) in a distributed system—at which the network or part of it can be placed. These locations are restricted to being a plain integer number, to keep the model as general as possible. As nodes are logical, the mapping between node and physical device is implementation-dependent. In order to cater for placement, S-Net is extended by two placement combinators.

The static placement combinator (denoted as $@NUM$) takes one operand and a location. The operand will be mapped statically to the PE indexed NUM. In $(A..B)@1$

the operand (two boxes A and B connected by serial composition) will be mapped to the PE indexed 1. The dynamic placement combinator (denoted as @<*t*>) supports only the parallel replicator. In $(A..B)!$@<*t*> each unique branch of $(A..B)$ is created, based on the tag <*t*>, and will be mapped to the PE with the same index as the tag <*t*>. In a case in which the placement is not annotated, the operand is mapped to the default PE with index 0. The S-Net language report [52] provides a more formal and in-depth description of S-Net language and supporting features.

2.2.2 S-Net Compiler

The compilation process of an RSP written in S-Net is composed of various stages, e.g. parsing, type inference, type checking, optimisation, and code generation. In the first stage, the S-Net source code passed as an input to the compiler is transformed to ***A**bstract **S**yntax **T**ree* (AST) representation. Next the compiler carries out the most important task of type inference together with various optimisations and annotations. Next the decision functions—this decision is used by the RTS for routing purpose—are generated, based upon user-defined types, patterns, and from the inferred type information. Finally, the compiler generates C-code in a portable format called the ***C**ommon **R**untime **I**nterface* (CRI) [47]. More in-depth details of the compilation process are covered in the S-Net implementation report [46]. To generate the executable, a C compiler is used to link the object code generated from CRI, the box function object code, and S-Net RTS libraries.

2.2.3 S-Net Runtime Systems and Distribution Layers

Currently for S-Net there are two different RTSs available: S-Net RTS and FRONT [43, 44]. FRONT employs work-stealing mechanism to distribute work to PEs. It also performs transformation on some patterns, e.g. synchrocell followed by serial replicator is turned into a specific box implementation, making it much more efficient than standard S-Net graphs. Since we are only focusing on S-Net RTS remaining discussion will be limited to S-Net RTS as RTS of choice for S-Net.

The interpretation of the CRI format, takes places in the RTS. A component included in S-Net RTS called *CRI deployer* transforms the CRI format to the actually executed representation of the S-Net stream network. The CRI deployer, together with modules for the types and patterns, ***I**nput/**O**utput* (I/O) communication, box language interface, task manager, stream manager and distribution, make up S-Net RTS. The task manager provides functionality such as task-creation and destruction. Similarly, the stream manager facilitates stream creation and destruction functionality.

The CRI deployer is specific to an RTS as it implements the final transformation of the network representation to target architecture. The CRI deployer takes the CRI code and produces representation of the original S-Net program as a graph called *RC graph* that contains RCs connected by streams. Each RC in a RC graph represents an S-Net entity or an operator. Since a box/network in S-Net is SISO, the stream is represented as a single reader and single writer FIFO buffer. The S-Net RC graph is directed and acyclic.

Serial composition is the simple case where operands are connected by pipelined streams, and therefore no extra entity is required as can be seen from Figure 2.3. Figure 2.4 shows parallel composition, in which two entities called *parallel compositor* and *collector* can be observed. The parallel compositor distributes messages from its input stream to operand branches. The collector gathers output messages from its operand branches. Similarly, as can be seen from Figure 2.5 and Figure 2.6, serial and parallel replications also features these operators, but in addition to distributing the messages they also generate a new operand instance dynamically based on need, i.e. depending on exit pattern in serial replication and tag value in parallel replication. The serial/parallel replicator and parallel compositor can be also called *dispatcher* as its main function is to dispatch the incoming message to appropriate RCs.

A distribution module manages distribution of tasks to physical computational resources. Currently S-Net supports multiple distribution strategies. This includes the default distribution layer *nodist*, which uses shared memory to provide communication and is applicable for shared memory multi-processor systems. An alternative is an *mpi* distribution layer which uses ***M**essage **P**assing **I**nterface* (MPI) [38] to facilitate communication and is part of distributed S-Net [51]. Another layer *scc* is similar to mpi distribution layer, but instead of MPI for communication it utilises SCC specific features to provide communication [135, 136].

2.2.4 S-Net Execution Layers

Currently S-Net supports multiple execution layers, e.g. PTHREAD [47] and **L**ight-weight **P**arallel **E**xecution **L**ayer *(LPEL)* [112].

The PTHREAD layer maps each S-Net entity to a dedicated ***P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**]* (POSIX) thread [60]. As the threads in this layer are managed by an ***O**perating **S**ystem* (OS) scheduler, it may suffer from cache- and context-switch related overheads. This is due to the nature of S-Net in which active entities tend to quickly exceed the number of available cores. The read/write operations on streams in this layer are protected by POSIX mutexes.

The LPEL was designed to overcome the problem faced by the PTHREAD layer. The LPEL creates as many threads as the number of the available processing cores. These threads are called *workers* in LPEL. Each S-Net entity is mapped to a lightweight coroutine. This alleviates the scheduling decision from an OS level to the LPEL level. In addition, LPEL also makes it possible to collect monitoring information, such as execution time of box entities or buffer usage. The read/write operations on streams in this layer are protected by atomic processor instructions, which emulate semaphores.

Since our work revolves around LPEL we will keep our discussion limited to this particular execution layer. The execution layer also provides a scheduler to map computation to resources. The LPEL provides two schedulers described in work by Prokesch [112] and Nga [103]. The former provides a decentralised scheduler in which the task to core mapping is done statically, while the latter provides a centralised scheduler and demand-based priority for tasks. We will describe the latter in detail as we also use the same in our work.

2.2.4.1 LPEL - A Stream Execution Layer with Efficient Scheduling

The LPEL is an execution layer designed for S-Net which allows collecting monitoring information and provides control over mapping and scheduling of tasks. In addition, it provides task- and stream-management functionality. LPEL adopts a user-level threading scheme providing the necessary threading and communication mechanisms in user-space. It builds upon the services provided by the OS or virtual hardware, such as kernel-level threading, context switching in user-space, atomic instructions, and timestamping.

Figure 2.8 shows an abstract design of the LPEL. In LPEL, each core is modelled as a *worker*. A special case of a worker is called *conductor*. A task is ready when it has all of its data available on its input stream and its output stream is not full. The LPEL scheduler does not map tasks to worker permanently. Instead, ready tasks are stored in a queue called ***C**entral **T**ask **Q**ueue* (CTQ). It is the responsibility of the conductor to manage this CTQ. The scheduler uses the notion of data demands on streams to derive the task priority. The built-in monitoring framework is used to retrieve stream state information at runtime and to analyse data demands. When a worker is free, it sends a request for a new task to the conductor. On each request for a task from a worker, the conductor retrieves the task with the highest priority and sends it to the worker. While the worker is executing the task, the conductor updates the CTQ, and also updates the task priority if needs be, without interrupting workers.

All the conductor-worker communications are exercised via *mailboxes*. The communication takes place only between conductor and worker; worker-worker commu-

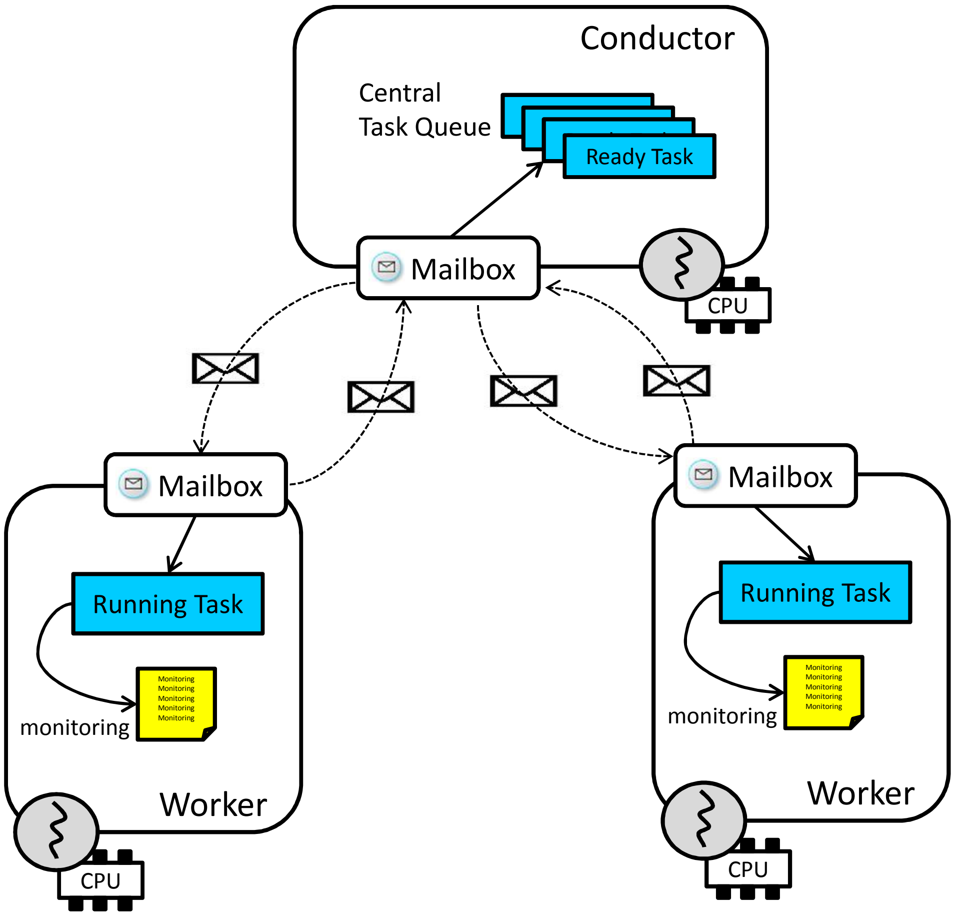


Figure 2.8 LPEL with Centralised Scheduler

nication is not allowed. The mailbox basically consists of a message queue, in which messages are enqueued by conductor/workers and dequeued only by the owning worker/conductor. As the workers access the mailboxes of the conductor concurrently, care must be taken to ensure corruption-free operations of the mailbox—atomic operation or PTHREAD mutexes are used to protect critical region.

The streams in LPEL are uni-directional and implemented as FIFO buffers. The read/write operations are protected by atomic operation or PTHREAD mutexes. Reading from an empty stream will put the task into a blocking state. Since the streams are not bounded, writing is always successful.

2.2.4.2 Distributed S-Net with LPEL

We already know that the S-Net language is extended with a concept of nodes and placement to support the distributed system. Each PE on the distributed system is equipped with its own S-Net RTS and LPEL/PTHREAD execution layer. There is no shared memory to provide communication between workers in LPEL. For this reason the centralised version of scheduler can not be used. Prokesch [112] implements a decentralised scheduler for LPEL that features a local scheduler for each worker. This scheduler is a perfect fit for such a distributed scenario—of course as an alternative to the Pthread based execution layer.

The compiler first takes the S-Net program with placement annotations and generates the PE specific CRI code. Each PE runs an instance of its own S-Net RTS and LPEL layer with a decentralised scheduler. Each PE uses CRI code to create LPEL tasks and streams as usual. Once the task to PE mapping is done at the compile time, it is not changed during the run time. PE-specific LPEL instance with decentralised scheduler controls the scheduling of tasks within the PE.

S-Net RTS provides three components called, ***Input Manager*** (IM), ***Output Manager*** (OM) and the ***Data Fetcher*** (DF). When a stream crosses the PE boundary it is registered with a manager, input stream with IM, and output stream with OM. The task reading/writing to the stream performs operations as normal, while IM/OM transparently moves messages between different PEs.

Let us now look at a simple example: tasks t_1 and t_2 are connected serially and are located on different PEs, PE_1 and PE_2 respectively. When t_1 produces a message on its output stream, OM on PE_1 reads the message and by using MPI sends it to the IM of the corresponding PE, PE_2 in our case. The IM on PE_2 also uses the MPI to receive the message. Once it has the message it writes it to the corresponding stream, the input stream of t_2 in our case.

When a message is to be sent across a PE boundary, only the representation of the field data is sent instead, to avoid unnecessary data transfers. The representation consists of a ***U**nique data **Id**entifier* (UID) and the location of the PE where the actual data is held. This mechanism allows the avoiding the transfer of data until it will be actually needed by a task. At this point, the DF sends a fetch request to the IM of the PE where the data is kept. The DF uses UID and location information to identify the data it needs and the location of the PE. The IM that received the data request informs the DF of the same PE. The DF retrieves the data requested and sends it to the PE that asked for it. The local scheduler of LPEL does not control either IM, OM, or DF, as they are implemented as kernel-level threads and are scheduled by the OS scheduler. One benefit of such a design is that it prevents deadlocks; on the flip side this design may increase OS-level context switches.

2.2.5 S-Net Terminology at a Glance

Here is a quick listing of the S-Net terminology:

Box is a component that encapsulates computational logic. The boxes are stateless components and they are implemented in auxiliary language e.g. ANSI C and SAC [48].

Stream is a communication channel that allows the exchange of information between two boxes/components connected by it.

Task is a runtime instance of box implementation. A task can be described as an iterating process that reads input from its input streams, performs the associated box's computations, and writes output to its output streams.

RC invocation is a term to describe single iteration of a task.

Network describes a stream program or segment of such a program by means of boxes and streams connecting them.

Combinator describes the way the boxes are connected, e.g. in serial pipeline fashion or in a parallel branch manner.

Message is a datum that is processed by a task on an RC invocation. Each message is made up of a set of label-value pairs. There are two types of labels called *fields* and *tags*. Fields are completely opaque to S-Net, as they are associated with values from the box language domain; they get manipulated only inside boxes. Tags are integer numbers and accessible by both S-Net RTS and user-defined boxes.

Multiplicity defines the ratio of the number of input messages to the number of output messages per RC invocation. A task with multiplicity of n-to-m consumes n messages from its input stream and produces m messages in its output stream on each invocation.

2.3 Power and Energy Optimisation

Power management techniques to reduce energy consumption have been extensively studied in prior work. The metrics including but not limited to *energy*, *power*, ***E**nergy **D**elay **P**roduct* (EDP) and ***E**nergy **D**elay-squared **P**roduct* (ED^{2}P) are used in power studies depending on the goals of the work and the type of platform being studied. Energy is often considered the most fundamental of the possible metrics for platforms, ranging from mobile/embedded to data centres and ***H**igh **P**erformance **C**omputing* (HPC) [70]. Energy can be calculated as:

$$Energy = power \times time \tag{2.1}$$

where, energy(E) is measured in joules(J), power(P) is measured in watts(W), and the execution time of an application time(t) is measured in seconds(s).

Power-management techniques have been employed/studied extensively at various levels, such as logic, architecture, and an OS-level [102].

The techniques used at the logic level include but are not limited to: *clock gating* — turning off clock signal to unused circuits; *half-frequency and half-swing clock* — where both edges of a clock signal are used enabling operation at half the frequency in the former, and in the latter the clock swings for only half of the V; *asynchronous logic* — when a clock signal is completely absent and completion signals are used instead.

The techniques used at the architecture level cover various systems: *a memory system* — in which various sub-techniques are used, such as selective activation of cache hierarchy and memory banking, compression of instruction in memory saving instruction fetch energy [15, 89]; *communication system* — e.g. Gray code or bus-invert encoding of address on bus [121] or network-no-chip with topology-aware routing or reconfiguration techniques [108]; *parallel processing* — in which functional units are replicated to allow processing of data in parallel.

The techniques used at the OS level include: *power manager* — as in Linux and Windows OS, the power manager allows diverse options such as timer to put display, hard drive, or other components to sleep mode; in Linux `cpufreq` infrastructure handles ***C**entral **P**rocessing **U**nit* (CPU) frequency scaling [107].

There are other levels at which different techniques are used for power/energy efficiency reasons such as compiler, application, and network. A large amount of research work has been done in this area, and some surveys [11, 16, 28, 99, 100, 129, 130, 134] that cover various topics are recommended to interested readers.

2.3.1 Power Efficiency vs Energy Efficiency

There is distinction between power-efficient and energy-efficient systems. A power-efficient system may not be an energy-efficient system. As an example, a power-efficient system may decrease the clock frequency which in turn saves power, but this affects the execution time of an application running. There comes a time when the execution time of this application increases to a point when overall energy consumed actually increases, rendering system power-efficient but not energy-efficient.

DVFS and ***D****ynamic* ***P****ower* ***M****anagement* (DPM) are two widely used techniques for reducing energy consumption in the processing unit. As name suggests in DVFS, the voltage and/or frequency of a particular component is changed to decrease power consumption at the price of potentially lower performance. The DPM techniques switch the processor/CPU/core to a low-power inactive state as long as possible [11].

In ***C****omplementary* ***M****etal-****O****xide* ***S****emiconductor* (CMOS) technology, the total power consumption can further be decomposed into static and dynamic components [141]:

$$P_{total} = P_{static} + P_{dynamic} \tag{2.2}$$

Where P_{static} represents static power consumption, resulting due to various factors including—but not limited to—sub-threshold leakage, tunnelling current through gate oxide, and leakage through a reverse-biased p-n junction [120, 129]. $P_{dynamic}$ represents dynamic power consumption, resulting due to the activity of logic gates, i.e. charging/discharging of capacitors and temporary current paths between the supply rails. The dynamic power consumption increases with the increase in switching frequency. This is due to the fact that dynamic power consumption occurs only during gate activity. DPM solutions are best suited for decreasing the impact of the static power component, as the static power consumption is always present, even when the circuit is on stand-by. In contrast, the dynamic power consumption is present when there is some activity, and as such, DVFS approaches are more suitable for reducing the dynamic power consumption.

2.3.2 Dynamic Power Consumption

The dynamic power consumption of a core can be calculated as:

$$P_{dynamic} = aCV^2 f \tag{2.3}$$

where a is switching activity factor, i.e. how many transitions occur in a chip, C is physical capacitance, V is supply voltage, and f is clock frequency. Each of these parameters can be used to reduce the dynamic part of power consumption. Power consumption due to the switching-activity factor a can be decreased by using techniques like clock gating, in which clock signal is not sent to idle circuits, e.g. caches. Power consumption due to the physical capacitance C can be reduced by changing low-level design parameters such as transistor sizes and wire lengths. By decreasing the clock frequency f power consumption can be reduced, but as we noted earlier, the energy consumption may or may not improve. Reducing the supply voltage V will also reduce power consumption. This is better than frequency as a small change in voltage will have a big impact on power consumption, although decreasing voltage also requires a decrease in frequency to allow safe operation of the chip.

A change in voltage can change the energy-optimal point of operation for a core. Since core frequency has an approximately linear relationship with supply voltage, and power consumption is approximately quadratic with voltage as can be seen from Equation 2.3, this provides a single knob to adjust power consumption and performance.

Since in the context of this thesis we used only DVFS to reduce energy consumption, we will focus only the dynamic part of the power consumption in general. As we already noted that voltage change has a big impact on power consumption in comparison to other parameters, our centre of interest lies in DVFS techniques. The DVFS techniques can be subdivided further into *offline* and *online*. The distinction is based on when the DVFS decision is made, i.e. statically at compile time (offline) or dynamically at runtime (online). We will visit DVFS again in Chapter 7.

2.4 Many-Core Network on Chip Platforms

The evolution of semiconductor fabrication processes has driven computer architectures from single-core processors to multi/many-core systems. When the number of cores on a chip started to climb the classic solutions like buses could not provide assurance of a reliable connection between them. Luca Bennini and Giovanni De Micheli introduce the NoC concept in 2002 to resolve the interconnection problem with multi/many-core architectures [14]. As name suggests, NoC consists typically of routers, network-adapter interface, and connections to allow a flow of data between

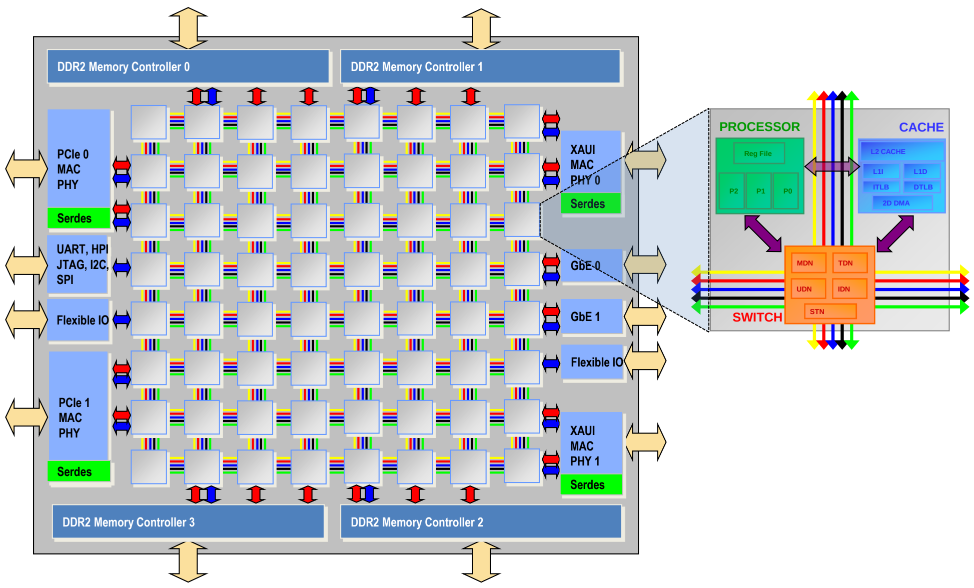


Figure 2.9 TILE64 Architecture Diagram [3]

various components available on/off-chip. Intel's SCC is the architecture upon which this thesis work was developed. However, it is worth mentioning some other example architectures that are similar. We recommend some survey papers [1, 4, 21] for the readers interested in the details of NoCs in general.

2.4.1 The Tilera TILE64 Processor

Processors in the Tile family are based on the Tilera's multi-core architecture. We will take a brief look at the TILE64 processor in particular as an example. The platform supports several programming languages, e.g. full ANSI C, providing a way to make porting of legacy code easy.

Figure 2.9 shows the architecture diagram of the TILE64 processor. The processor features 64 homogeneous PEs arranged in a two-dimensional 8x8 grid. Each PE is referred to as a tile. All the tiles are connected with the I/O, the peripherals and each other via high-speed, on-die, packet-switched, multiple two-dimensional mesh networks (the mesh network is based on Tilera's iMesh interconnect technology) [13, 138]. By employing a dedicated mesh network with different latencies and bandwidths for inter-tile, memory, and I/O communications, the architecture provides a high bandwidth and extremely low latency communication among tiles. There are four on-die ***Memory Controllers*** (MCs) that connect the tiles to on-board ***Double Data Rate*** (DDR) memories.

The tiles on the TILE64 can operate between 600—1000 MHz. Additionally, cores can be grouped into islands to eliminate unnecessary communication and reduce power consumption (unused tiles can be put into sleep mode). As can be seen from Figure 2.9, each tile contains three major components: processor engine, cache engine, and switch engine. The processor engine is a three-way ***V****ery* ***L****ong* ***I****nstruction* ***W****ord* (VLIW) processor architecture with an independent program-counter.

The cache engine contains the tile's ***T****ranslation* ***L****ookaside* ***B****uffers* (TLBs), caches, and cache-sequencers. In addition to the support for both private and shared memory, the TLBs also support pinning blocks of memory in the cache. There are separate 8 KiB L1 instruction and data caches. L1 instruction cache has 8 entries TLB, while data cache has 16 entries TLB. A unified 2-way 64 KiB L2 cache backs the L1 caches. Each tile also contains a 2D ***D****irect* ***M****emory* ***A****ccess* (DMA) engine that supports block copy functions like cache-to-memory, memory-to-cache, and cache-to-cache. There is no L3 cache but each tile's L2 cache can be shared with other tiles, in effect providing a shared L3 cache.

The switch in the switch engine is a full crossbar for non-blocking routing, with credit-based flow control. There are in total five different networks out of that; four are dynamic networks and one is a static network. The dynamic networks are dimensional-ordered wormhole-routed. There is one-cycle latency for each hop through the network for cases when the packets are going straight. If the packet has to make a turn at the switch then latency is increased by one cycle due to the route calculation.

There are five different networks in the iMesh. Each network supports 32-bit unidirectional links, allowing traffic flow in both directions at the same time. The five networks are:

Static Network (STN) is a scalar network with low latency allowing static configuration of the routing decisions. It is mainly used for streaming data from one tile to another via pre-configured routes.

User Dynamic Network (UDN) low latency, user programmable, packet-switched network used for communications between threads running in parallel on multiple tiles.

Memory Dynamic Network (MDN) used for memory transfers such as loads, stores, and cache misses.

Tile Dynamic Network (TDN) supports data transfer between tile caches. TDN works in concert with MDN.

Input/Output Dynamic Network (IDN) is network accessible to OS-level code, not user applications. Used primarily to transfer data between tiles and I/O devices, and I/O devices and memory.

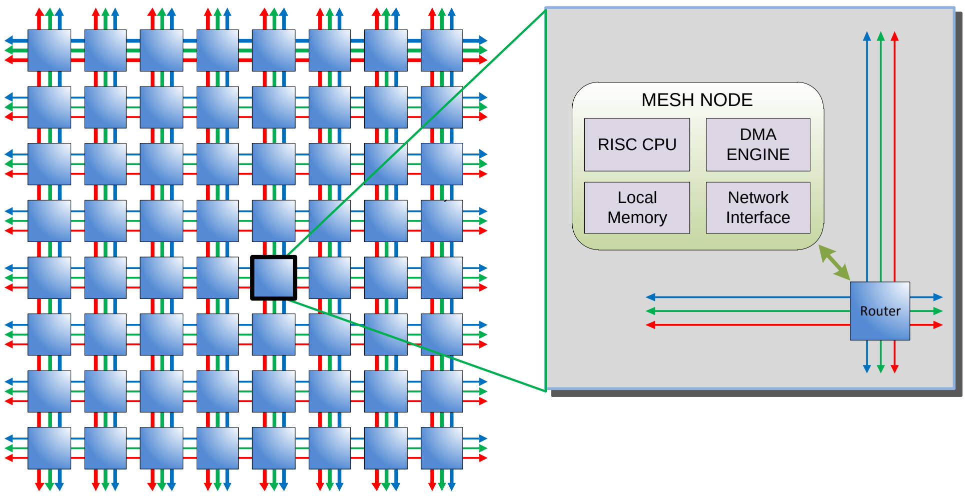


Figure 2.10 Adapteva Epiphany-64 Architecture Diagram [59]

The Tile Processor architecture defines a flat globally shared 64-bit physical address space and a 32-bit virtual address space. In addition to the default hardware backed cache coherent memory the TILE64 also supports other memory modes, i.e. a non-coherent and a non-cacheable memory mode. Different memory attributes and modes are managed and configured by means of page table entries and enforced through TLB entries. TILE64 provides directory-based coherence policy. Every node has directory cache and off-chip directory controller. Tile-to-tile memory request/response transits the TDN. Off-chip memory request/response transit the MDN. The traffic due to the cache coherency was so high that an extra mesh network was added to the later TILEPro processors. A Coherence Dynamic Network (CDN) is used only for passing invalidation messages needed for the cache-coherency protocol.

On TILE64 processor each tile can independently run a full OS, e.g. GNU/Linux. In addition, multiple tiles taken together can run a multi-processor OS like an ***S**ymmetric* ***M**ulti**p**rocessing* (SMP) version of GNU/Linux.

2.4.2 Adapteva Epiphany-64

The Epiphany architecture from Adapteva targets the embedded system domain with the goal of providing low power, multi-core, scalable, parallel, and distributed shared-memory coprocessor [55, 59]. Currently there are two versions of the Epiphany coprocessor available: the Epiphany-16 (16 cores in a 4x4 grid) and the Epiphany-64 (64 cores in a 8x8 grid). The PE in Epiphany architecture is referred to as *eCore*, future versions of the Epiphany are expected to house up to 4096 eCores [106, 132].

Figure 2.10 shows a top-level and node-level diagram of the Epiphany-64 coprocessor. The 64 *mesh nodes/eNode* are organised in a 8x8 2D array, connected to each other by a mesh network. Each eNode consists of a ***R**educed **I**nstruction **S**et **C**omputer* (RISC) CPU (eCore, 800 MHz maximum frequency), 32 KiB of local scratchpad memory, a DMA engine (with two DMA channels supporting both non-blocking and blocking DMA transfers), and a network interface to an *eMesh* router. Each eCore includes a general purpose program sequencer, 64-word—each word is 32 bits—large general purpose register file, ***I**nteger **A**rithmetic **L**ogic **U**nit* (IALU), ***F**loating **P**oint **U**nit* (FPU), a debug unit, and an interrupt controller. There are no caches available on Epiphany architecture. Epiphany-64 is a coprocessor and as such it does not run any OS. The Epiphany architecture uses a flat unprotected memory map. Every mesh node can directly access the complete memory system, without any limitation. Each eCore can address its local ***S**tatic **R**andom **A**ccess **M**emory* (SRAM), other eCores' SRAM, and a shared off-chip ***D**ynamic **R**andom **A**ccess **M**emory* (DRAM). The local memory system is split into four separate sub-banks in order to increase the performance by means of simultaneous memory accesses.

All the on-chip and off-chip communication is handled by an eMesh, an on-chip, high-speed 2D mesh network. Three different 2D mesh networks make-up the interconnect network of the Epiphany64. Different types of traffic flows through their dedicated eMesh. Each Epiphany64 has four independent off-chip *eLinks* that extend the eMesh network and memory architecture. These eLinks allow connection of multiple Epiphany64 on a board and to a ***F**ield **P**rogrammable **G**ate **A**rray* (FPGA). The three different eMesh networks are:

cMesh routes on-chip write traffic. The cMesh network connects an eCore to all four of its neighbours and has a maximum bi-directional throughput of 8 bytes/cycle.

xMesh routes off-chip write transactions, including a case when the traffic destination is an another chip in a multi-chip system configuration. The maximum throughput of the xMesh depends on the available off-chip I/O bandwidth.

rMesh routes both on-chip and off-chip read traffic. The xMesh has a maximum throughput of one read transaction every eight clock cycles.

The Epiphany architecture can be programmed using C, C++, OCL, and has a ***S**oftware **D**evelopment **K**it* (SDK). The program is written in two parts: one for the host CPU and one or more `kernels` for running on the eCore nodes. Generally, the application's initialisation and outer loops are performed on the host CPU, while a numerically-intensive loop is developed as a kernel executed on the eCore.

2.4.3 ClearSpeed CSX700

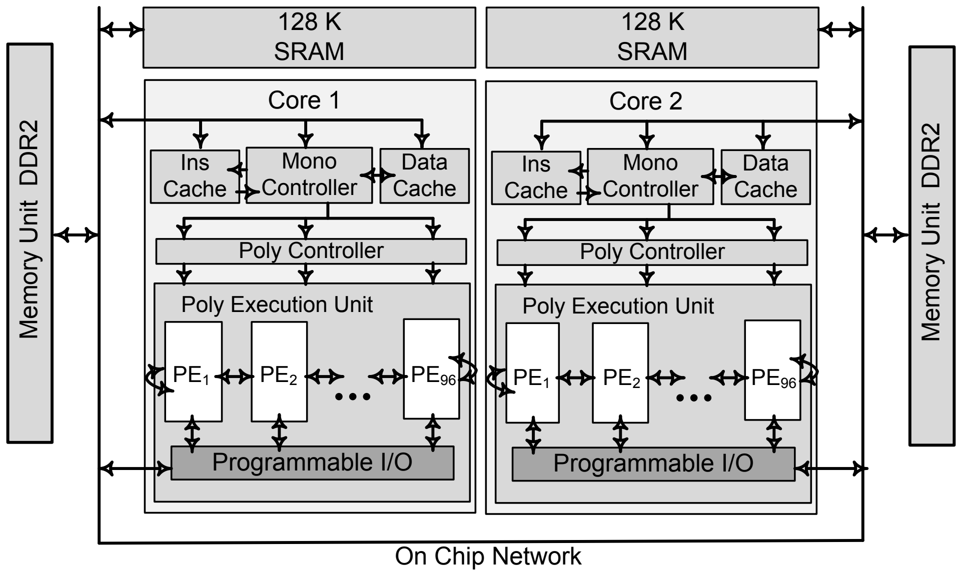
The ClearSpeed CSX700 [126] is a highly parallel ***S**ingle **I**nstruction **M**ultiple **D**ata* (SIMD) architecture and is slightly different to the previous NoCs we have looked at. The ClearSpeed architecture—CSX700 is a coprocessor like Adapteva Epiphany-64—has two cores, each with 96 PEs (192 PEs in total), providing a peak performance of 96 GFOLPS, while consuming power of less than 9 watts.

As illustrated in Figure 2.11a, CSX700 has two similar ***M**ulti-threaded **A**rray **P**rocessor* (MTAP) cores. Each MTAP core has a 128 KiB SRAM that can be accessed by both MTAPs, and a DDR2 memory interface. Each memory interface supports up to 8 GiB of DRAM and can be accessed by both MTAP cores. The CSX700 operates at the clock frequency of 250 MHz. The main components of MTAP are execution units, caches, and I/O [124, 125].

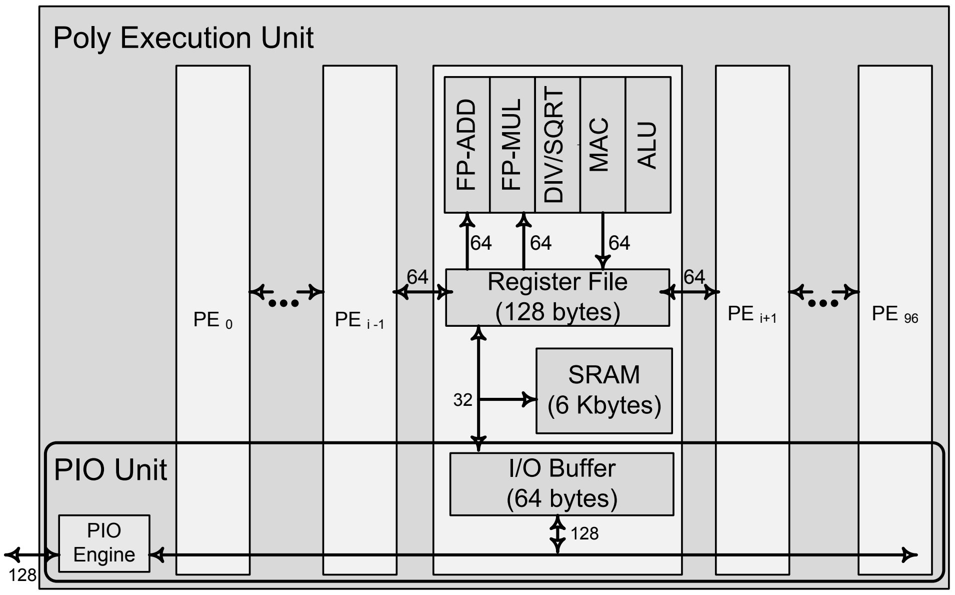
The execution units consist of two main parts. The first is an RISC-like control unit, referred to as a *mono execution unit*, which acts on mono (non-parallel) data and handles program flow control and I/O functions. The second is highly parallel SIMD architecture called *poly execution unit*. The mono and poly execution units support 64-bit addressing, and have basically the same architecture and instruction set. The poly execution unit contains an array of 96 PEs, and acts on poly (parallel) data. A poly controller connects a mono execution unit to the poly execution unit.

The main features of the mono execution are: support for 8 hardware threads, 128 8-bit semaphores (for synchronisation), ***A**rithmetic **L**ogic **U**nit* (ALU), 64-bit FPU, and 8 128-byte register files (one per thread). Also there is 8 KiB instruction and 4 KiB data caches connected to the mono execution unit to speed-up access to code and data.

In a poly execution unit the array of 96 PEs provides both compute power and high bandwidth storage. The PE array operates on a SIMD model, i.e. processing multiple data items in parallel. As we can see from Figure 2.11b, each PE contains a 128-byte register file, 6 KiB of SRAM, single and double precision FPUs, a 64 KiB I/O buffer that connects to ***P**rogrammed **I**nput/**O**utput* (PIO), and high-speed dedicated I/O channels to two adjacent PEs called *swazzle path*. As shown in Figure 2.11b, the register file of each PE is connected to the register files of its left and right neighbours, via swazzle path. This in turn allows single-cycle register-to-register data transfer with neighbouring PEs. The PIO mechanism allows the poly execution unit to perform load/store to the external memory. Communication between both MTAPs, memory units, and other interfaces is facilitated by an on-chip bus network called *ClearConnect Bus*.



(a) Simplified CSX700 Architecture



(b) Simplified Poly Execution Unit Architecture

Figure 2.11 ClearSpeed CSX700 Architecture Diagram [57]

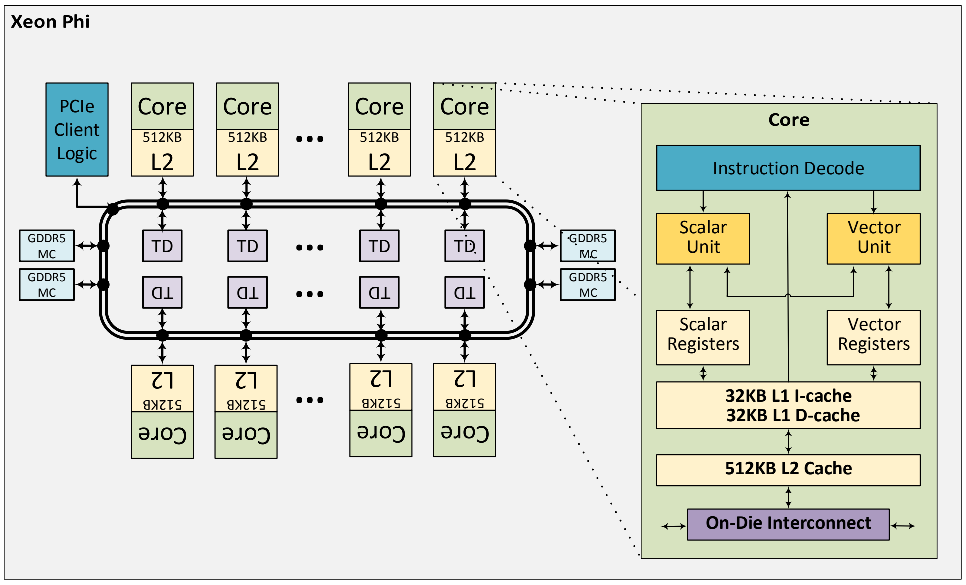


Figure 2.12 Xeon Phi Coprocessor Architecture Diagram

2.4.4 Intel Xeon Phi

All of the coprocessors in the Xeon Phi product family are powered by Intel's ***M**any **I**ntegrated **C**ore* (MIC) Architecture [32, 62].

Intel Xeon Phi has over 50 cores[3], at a maximum of 8 MCs, supporting up-to 16 GiB maximum memory providing 352 GiB/s bandwidth [30, 33] and ***P**eripheral **C**omponent **I**nterface **E**xpress* (PCIe) interface logic; this can be seen from the high-level diagram in Figure 2.12. The cores, PCIe Interface logic, and ***G**raphics **DDR** type **5*** (GDDR5) MCs are connected via a high-performance on-die bi-directional ***I**nterprocessor **N**etwork* (IPN) ring. The basic core clock frequency is 1.238GHz, but during peak workloads and depending on thermal conditions, Intel's Turbo boost Technology can increase the core frequencies to 1.333GHz at maximum. The connection between the host and Xeon Phi is made through the PCIe system interface. The Xeon Phi consumes, on average, 300 watts when operating at the base frequency with all cores active.

The cores on the Xeon Phi are simple-core—without expensive features such as out-of-order execution and branch prediction—based on x86 architecture, supporting four hardware threads per core. As shown in Figure 2.12, each Xeon Phi core has ***V**ector **P**rocessing **U**nit* (VPU), 32 KiB L1 instruction and 32 KiB L1 data cache, 512 KiB unified L2 cache and interface to ***O**n-**D**ie **I**nterconnect* (ODI). The 512-bit wide VPU contains the vector register file with 32 registers per thread context. The

[3]Product with series 31xx has 57 cores, 51xx has 60 cores, and 71xx has 61 cores. Here we will mainly refer to the latest Xeon Phi 7120 with 61 cores.

Xeon Phi offers full cache-coherency across all cores. The coherency is maintained by cross-snooping L2 caches in all cores. This is achieved by having distributed duplicate ***Tag Directory*** (TD) for each core. A TD tag contains the address, state, and an ID for the owner of the cache line. In the case of a cache miss a request is sent from the core that suffered the memory miss to the correct TD via the ring interconnect. Aggregated L2 cache provides about 30.5 MiB of on-chip shared memory. The Xeon Phi has two ring interconnect, one travelling in each direction. The ring interconnect is used by all connected entities, using special controllers called *ring stops* to insert requests and receive responses from the ring.

As Xeon Phi is in effect an x86 SMP on-a-chip architecture running Linux as an OS, a programmer can use familiar programming languages and models, for example C, C++, Fortran, PTHREAD, OpenMP, TBB, and Cilk Plus, etc [37, 113]. There are multiple execution models for Xeon Phi:

Multi-core only application's MAIN() is executed on host processor.

Multi-core Hosted with Many-core Offload application's MAIN() is executed on host processor, and selected routines (highly parallel, computationally intensive) are executed on the coprocessor.

Symmetric execution application's MAIN() runs symmetrically on processor and coprocessor.

Many-core only booted from host processor but application's MAIN() runs on coprocessor only.

2.5 Single-chip Cloud Computer

The *Single-chip* ***C****loud* ***C****omputer* (SCC) [58, 117] is a 48-core experimental processor created by Intel Labs as a "concept vehicle" for many-core software research. The SCC has on-chip message passing buffers and support for dynamic frequency and voltage scaling. Its 48 P54C cores are connected to each other and the main memory via a high-speed, on-die, packet-switched two-dimensional mesh network.

In this section we look at the SCC's hardware architecture in § 2.5.1 and § 2.5.2, some essential registers in § 2.5.3, memory hierarchy in § 2.5.4, followed by an overview of the DVFS feature in § 2.5.5.

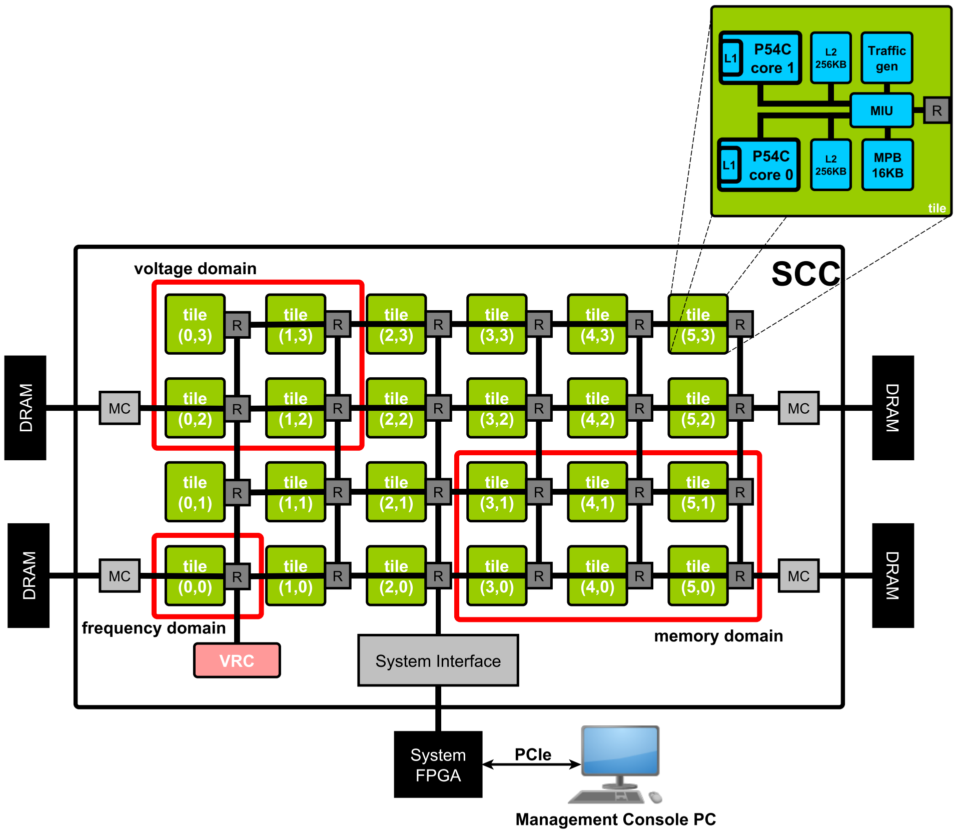


Figure 2.13 Simplified Intel SCC Architecture Diagram [75]

2.5.1 Top-level Architecture

As seen in Figure 2.13 the SCC [58, 117] consists of 24 tiles in a 4x6 grid. Each tile is connected to a high bandwidth, low latency, on-die two-dimensional mesh network via router. This arrangement of tiles/cores resembles a cluster on a single chip. The routers implement fixed X-Y routing, which means the packet first travels along the x-axis and then along the y-axis. There are four on-die MCs. Each MC can address two ***D**ual **I**n-line **M**emory **M**odules* (DIMMs). Each memory module can be 8 GiB at maximum (16 GiB max for each MC); this translates into maximum 64 GiB off-die DRAM in total for the system. For simplicity and clear distinction, we will refer to this DRAM memory as **main memory** from now on. The SCC chip also contains a ***V**oltage **R**egulator **C**ontroller* (VRC) that provides any core or the ***S**ystem **I**nterface* (SIF) the capability to adjust voltage and the frequency of tiles, allowing DVFS. The SIF provides a way for communication between system FPGA (a controller located on the system board) and router on the mesh network. The ***M**anagement **C**onsole **PC*** (MCPC) uses PCIe bus interface to connect to the system FPGA and in turn to the SCC.

2.5.2 Tile-level Architecture

Figure 2.13 also shows a detailed view of an individual tile. Each tile contains two P54C-based ***I**ntel **A**rchitecture* (IA) cores. Each core has associated 32 KiB L1 (data and instruction) and 256 KiB L2 caches. A tile also has 16 KiB (8 KiB for each core) block of SRAM called ***M**essage **P**assing **B**uffer* (MPB). Although, MPB is also called ***L**ocal **M**emory **B**uffer* (LMB) sometimes, we will use MPB through this work to maintain consistency. In addition, to accelerate message transfer between cores, a ***W**rite **C**ombine **B**uffer* (WCB) was added—not shown in Figure 2.13—to the tile. Furthermore, there is a ***T**raffic **G**enerator* (TG) for testing the mesh—not accessible to end user/application—, a ***M**esh **I**nterface **U**nit* (MIU), ***G**lobal **C**locking **U**nit* (GCU), ***C**lock **C**rossing **FIFO*** (CCF)—also not shown in Figure 2.13—and memory ***L**ook**u**p **T**able* (LUT) on the tile as well.

There is only one MIU on the tile, so the cores on the tile have to access MIU in a round-robin manner. The MIU is responsible for handling all the memory and message passing requests. In particular, the MIU packetises the data going from the cores to the mesh and de-packetises data coming from the mesh to the cores. A credit-based protocol is used to make sure of a smooth flow of data on the mesh network; MIU also handles the management of credits. Configuration registers control the operating mode of the SCC. These registers are located in the ***C**ontrol **R**egister **B**uffer* (CRB) that forms part of the MIU. When the cores on the tile need access to resources, e.g. MPB or main memory, an access request is generated. Once the access request is generated, it is then placed in the appropriate queue managed by MIU. There are three different queues:

- main memory queue—initially request is send to router, then to MC, and finally to main memory
- MPB queue—the request is sent to the core's MPB
- Local CRB queue—the request allows access to the local configuration registers, which also reside in the MIU

As we will see in Section 2.5.5, the tiles and mesh network can have different execution frequency. In order to enable asynchronous communication between the two domains, where execution frequency may not be same, the GCU takes an incoming clock and generates the divided local clocks for the router and the cores along with the synchronisation signals. GCU and CCF are responsible to make sure no data is lost while it crosses different clock domains. All the routers on the chip must be running at the same frequency; if not then any data transferred across the clock boundary will result in being lost or damaged.

2.5.3 Configuration Registers and FPGA Registers

Application can control the behaviour of various hardware elements on the tile by means of the configuration registers located in CRB within MIU. For example, these registers control clock divider settings, interrupt handling, and cache configuration. At the start-up these registers are always initialised with default values. An application can memory map these registers, in order to change the mode/behaviour of the hardware elements controlled by these registers at runtime.

Although, here we will look at the some of these registers, we will not dive deeply into their fine detail. Interested readers can find more in-depth information in [75, 79, 80, 83]. Each core on the SCC has LUT entries that map CRBs of all the tiles of the SCC. This allows any core of the SCC to access (read/write) configuration registers of any other core on the SCC, including itself.

Core configuration There are three main parts that form core configuration registers. This includes interrupt registers that are directly connected to the interrupt pin of the cores, control register contains bits to control external events of the processor, and the status register that contains different types of status information from the external interface of the core. Each core on the tile has its own core configuration registers.

L2 cache configuration This register affects the operation of L2 cache, e.g. enabling/disabling the L2 cache or clock gates the cache controller. There are two L2 cache configuration registers on a tile, one for each core.

Sensor Sensor registers aid thermal monitoring functionality. There are two sensor registers per tile. The first enables or disables the monitoring and the second provides the thermal reading value.

Global Clock Configuration There is one global clock configuration register per tile to control the router and tile clocks.

Tile ID This register contains the (x, y) coordinates of the tile and core ID of the reading core, which can be used to determine the physical location of tile/core on the SCC. There is one tile ID register per tile.

Atomic Flag Each tile has two ***Test-and-Set*** (T&S) registers, one for each core.

LUT Each core has its own LUT, which contains 256 entries. These entries are used to access different parts/hardware of the SCC.

In addition to the configuration registers there are additional registers located in FPGA.

Global Timestamp Counter provides a common time base across all the cores. Each core has its own timestamp counter, but these are not synchronised. Furthermore, this counter is based on 125 MHz system clock of the FPGA, and does not get affected by the frequency of the cores on the SCC.

Atomic Increment Counters there are 96 32-bit atomic increment counters. Each counter is pair of registers: the atomic increment counter register and the initialisation counter register. The read/write operation and results for these registers are as follows:

> **Initialisation Counter** returns the current register value on read access. Write access initialises the register with the new value.
>
> **Atomic Increment Counter** an atomic increment of the register value performed on read access. Write access causes an atomic decrement of the register value. The old value gets returned.

Global Interrupt Registers provides a mechanism to send interrupts to any core from any other core on the SCC. The global interrupt register is made-up of 48 interrupt status registers, 48 interrupt mask registers, 48 interrupt reset registers, and 48 interrupt request registers, one for each core.

2.5.4 Memory Architecture

It is important to notice that **the SCC does not provide cache coherency**. If the shared memory is used it becomes the programmer's responsibility to maintain coherency explicitly.

Physically, SCC memory consists of off-chip DDR3 DRAM (main memory) and on-chip SRAM (MPB). The tiles on the SCC are divided into four regions. As can be seen from Figure 2.13 each region, also called *memory domain* contains six tiles. Each domain has a particular MC assigned to it (by default the nearest one).

In the default boot-up configuration, memory controlled by MC is divided evenly into the nearest MiB as 12 parts. Each part serves as a private memory for a core in a memory domain. A part of surplus memory (64 MiB in total, 16 MiB per MC) is allocated as a shared memory among cores. The access to private memory always goes through MC that is assigned to memory domain, while access to the shared memory can go through any MC.

From a programmer's point of view, the SCC provides three different memory types as listed below:

- A private off-chip memory for each core. This memory is cache coherent with an individual core's L1 and L2 caches.
- A shared off-chip memory. This memory may be configured as uncached or cached. If configured as cached it is the programmer's responsibility to manage coherence.
- The MPB, 384 KiB (16 KiB per tile) of physically distributed logically shared on-chip memory.

This division between the shared and private memory is done by means of LUT. Although main memory is divided into shared and private parts at the boot-up time with some default values, where this division occurs can be re-configured by modifying LUT entries. The LUT entries can be changed in two ways: the first is to change them once the cores are booted with an OS; this change will be lost once the cores are re-booted. The second way is to change the default values so that even if the cores are re-booted, changes are preserved.

2.5.4.1 Message Passing Buffer

Each tile has 16 KiB of MPB. By default, 8 KiB are assigned to each core. Since MPB is on-chip, access latency is in par with that of L2 cache. The MPB can be used in multiple ways, e.g. assigning 8 KiB per core or using all 384 KiB as scratchpad memory. Any core on the SCC can access any part of MPB, even though that part is assigned to another core. This provides a way for very low latency communication between cores. Since each 16 KiB segment is physically located on each tile, access latency will differ based on the physical location of the MPB and core. Table 2.1 lists the approximate memory-access latency for reading a cache line from various memory systems. Although the table lists local MPB access with and without bypass, in reality the bypass mechanism cannot be used, due to the hardware design bug [74, 83].

On the SCC, data can be moved at the granularity of the L1 cache line (32 bytes). There is no cache coherency on the SCC, and MPB (in effect, a shred memory) can be accessed (read/write) by any core; this can introduce the problem of inconsistent data (stale data). To address the issue, Intel has provided a special flag called the ***Message Passing Buffer Type*** (MPBT). Any MPB type memory is tagged with MPBT flag by the OS. All cache lines tagged with MPBT will bypass the L2 cache and directly go into L1 cache. The Intel ***Instruction Set Architecture*** (ISA) is also extended with an

Access Location	Approximate latency to read a cache line (cycles)
L2 cache	18 core cycles
Local MPB - with bypass	15 core cycles
Local MPB - without bypass	45 core cycles + 8 mesh cycles
Remote MPB	45 core cycles + 4*n*2 mesh cycles
DDR3 (main memory)	40 core cycles + 4*n*2 mesh cycles + 30 on-die mc (400 MHz) + 16 cycles (400 MHz off-die DDR3 latency)
n=number of hops to the MPB or the memory controller ($0 < n < 10$)	

Table 2.1 Various Memory Access Latencies [77]

instruction called CL1INVMB. Using CL1INVMB will invalidate all the cache lines tagged as MPBT in the core's L1 cache. Access to these invalidated L1 cache lines will force an update of the L1 cache lines with the data from actual memory, as there are no valid cache entries for them in L1 cache. We will look a little more in-depth at caching policies in Section 2.5.4.2.

A WCB was added to facilitate coalesced memory write operations. The WCB combines adjacent writes, up to a single cache line, and writes them to memory in a single write operation.

2.5.4.2 Cache Policy

The P54C core on the SCC can have only one outstanding memory request and will stall on missed reads until data is returned. On missed writes, the core will continue operation until another miss of either type occurs. Multiple outstanding requests at the tile level can be supported by the mesh network and the memory system.

As we already know, the memory on the SCC can be configured as cached, uncached, or MPBT. We can say that caching happens at three levels in the SCC: L1 cache, L2 cache, and WCB. Figure 2.14 details the memory type and how it interacts with caches. The Linux that runs on the SCC exposes three different memory device drivers: DCMDeviceFD (maps memory as cached), NCMDeviceFD (maps memory as uncached), and MPBDeviceFD (maps memory as MPBT type).

Both the L1 and the L2 caches on the SCC are 4-way set associative with a cache line size of 32 bytes, are write-back, and do not allocate on write miss, i.e. are write-around. Since caches are write-around, write misses are treated as a write directly to the main memory.

cached memory as name implies, cached memory is cached at both L1 and L2 cache level. Memory access goes through the L1 and L2 caches, then to the main

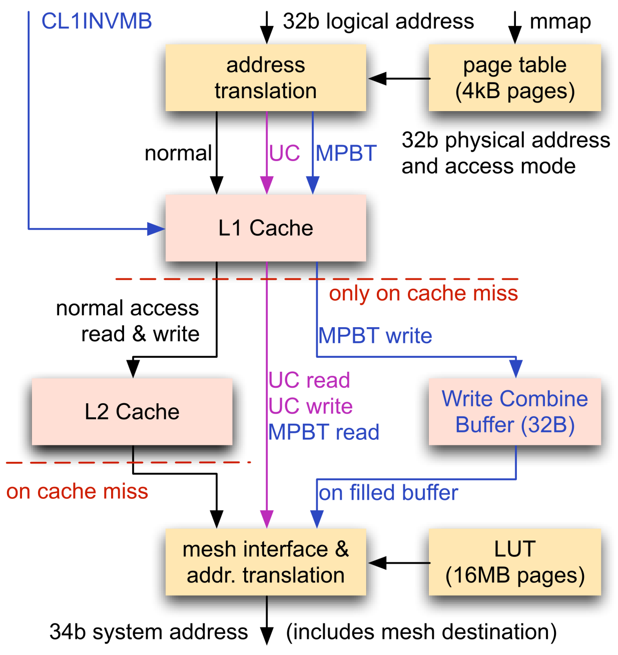


Figure 2.14 Memory Type and Cache Interaction [114]

memory if there are cache misses. Cacheable memory has a granularity of a 32 Byte cache line.

uncached memory if memory is configured as uncached it will bypass both L1 and L2 caches. A read operation to this memory bypasses both caches, and the read value is stored directly in the registers of the core, while write requests are delivered directly to the MIU. The uncacheable memory has a granularity of 1, 2, 4 or 8 bytes.

MPBT memory tagged as MPBT will always bypass the L2 cache and will only be cached in L1 (at a granularity of a L1 cache line). All sequential writes to the same cache line of MPBT type memory will accumulate in WCB; from core's perspective the write operation is complete at that point. The WCB will write data to the main memory when the entire cache line is filled, or write access to a different cache line occurs.

Since there is no cache coherence among the cores, if we want to use cached memory or would like to use MPBT with WCB, special care must be taken. This can be explained more clearly with an example.

Time t_1 Core-A and Core-B reads from the cached memory; the read data is stored in L1/L2 caches.

Time t_2 Core B writes to the same memory address; the data is in cache or WCB.

Time t_3 Core A reads from the same cached memory again; the cached data is returned from L1/L2 (this is stale data).

If memory is mapped as MPBT there are two steps that need to taken: the first is to perform dummy write to different cache line at t_2 to make sure that changed data is written back. The second is to issue CL1INVMB at t_3 before read to invalidate MPBT cache lines in L1 and force retrieval of data from MPB memory.

If memory is mapped as cached, this becomes tricky. Since caches are write-back, at t_2 any changes will still be in caches and we have to flush it to the memory. For L1 we can use WBINVD or INVD instructions to flush or invalidate cache lines. The main problem is that the P54C cores used in the SCC did not have an on-chip L2 cache originally. This is the reason for having the L2 cache external to the core. What this means is that the WBINVD and INVD instructions have no effect on L2. By using control register provided on the SCC, the L2 can be put into reset mode, which initialises all lines as invalid. However, this reset operation cannot be used to invalidate the L2 cache during execution, as it halts the core due to the hardware bug [81]. A manual L2 flushing routine was developed to solve the L2 flushing problem [82]; this routine is a high-latency operation. Again, for more details interested readers are advised to visit [73].

2.5.4.3 Lookup Table

The cores on the SCC are 32-bit and as such they can address up-to 4 GiB of memory. As we know, there are 4 MCs on the SCC and combined they can handle up-to 64 GiB of memory. This is where MIU uses LUT. LUT is a set of configuration registers that converts 32-bit core addresses to 46-bit system address. Each core on the SCC has its own LUT. Each LUT contains 256 entries, one for each 16 MiB segment of the core's 4 GiB physical memory address space. Each entry can point to any memory location, e.g. MPB, CRB, SIF, or main memory.

Table 2.2 shows the entries for a core's LUT for the SCC system with 32 GiB of main memory. The system that we use for our research has 32 GiB of main memory.

As you may recall from Section 2.5.4, there are 4 memory regions, each region containing 12 cores. Each region is served by particular MC. 32 GiB main memory is 8 GiB per MC. This means that each core will have $8192MiB/12 = 682.666MiB$ of private memory. As each entry in LUT points to 16 MiB segments, we will need $682.666MiB/16MiB = 42.666$ slots, by rounding down its 42 slots. In Table 2.2, entries 0–40 and entry 255 make up these 42 slots. LUT entries 128–131 point to

LUT Entry	No of Entries	Map To
0 - 40	41	Private Memory
41 - 127	87	Unused
128 - 131	4	Shared Memory
132 - 191	60	Unused
192 - 215	24	MPB, One entry for each tile
216 - 223	8	Unused
224 - 247	24	CRB, One entry for each tile
248 - 249	2	Unused
250	1	MCPC TCP/IP Interface
251	1	VRC
252 - 254	3	Unused
255	1	Private Memory

Table 2.2 LUT Entries for the SCC with 32 GiB of System Memory [75]

the shared memory ($4 slots * 16 MiB = 64 MiB$), and all the cores maps their LUT to same physical address. Entries 192–215 and 224–247 map MPBs and CRBs of all the tiles of the SCC, e.g. entry 192 maps to MPB of tile (x=0, y=0) and entry 193 maps to MPB of tile (x=1, y=0), and so on. Entry 250 is used for ***T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol* (TCP/IP) communication between the SCC and the MCPC. while entry 251 maps to VRC. There are lots of slots that are empty and do not map to anything in the default configuration. The next section describes briefly how the actual address translation from core address to system address works in the SCC.

2.5.4.4 Address Translation

Figure 2.15 illustrates the translation mechanism employed by the SCC to turn a 32-bit core address in to a 46-bit system address.

As a standard procedure during memory access, the core's ***M**emory **M**anagement **U**nit* (MMU) converts 32-bit core virtual address to 32-bit core local physical address. During a cache miss, MIU receives this 32-bit core local physical address. At this point MIU uses LUT to translate core address to the system address.

The upper 8-bits from the provided 32-bit core local physical address index an entry into the LUT. The LUT returns a 22-bit address, which also contains some routing information, described below.

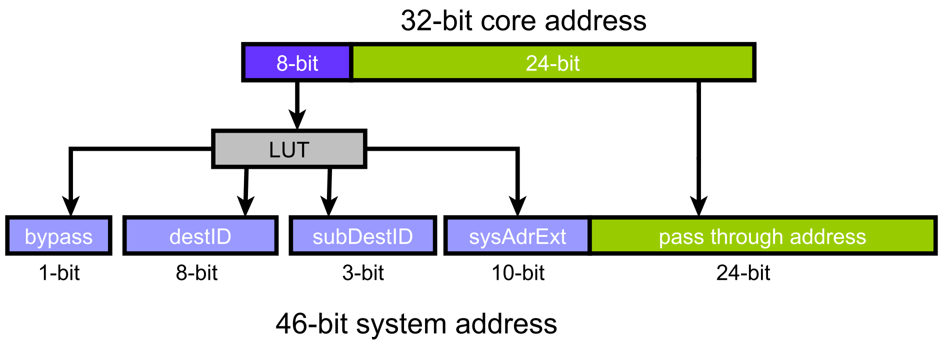


Figure 2.15 Address Translation [75]

bypass bypass is a 1-bit value which is used to specify the tile's local MPB access (although it is not used due to the hardware bug).

destID the 8-bit destination ID (destID) identifies another tile on the SCC for routing via the 2D mesh network.

subDestID the 3-bit sub-destination ID (subdestID) defines the port where the packet leaves the router; which type of memory access will occur is based on which port of the router is defined, e.g. main memory, MPB, or CRB.

sysAdrExt the 10-bit system address extension (sysAdrExt) is prepended to the 24-bit that passed through from the core address, resulting in 34-bit address.

It's only the 34-bit address that is send to the MC, as we know that the maximum supported memory for each MC is 16 GiB. Since $2^{34}/1024^3 = 16$ GiB, the 34-bit address is enough for 16 GiB memory. The 34-bit address is only sufficient for one MC. Still we have all the MPBs and CRBs and various registers. Finally, the 46-bit system address is obtained by combining the 1-bit bypass bit, 8-bit destination ID, 3-bit sub destination ID, and a new 34-bit address (10-bit system address extension prepended to the 24-bit pass through address).

2.5.5 Voltage and Frequency Scaling

Power management on the SCC comprises three components that work with separate clocks and power sources: tiles, mesh network, and MCs. As can be seen from Figure 2.13 on the SCC, there are voltage and frequency domains as well, in addition to the memory domains. These voltage and frequency domains are also known as **islands**. From now on we will refer to them as islands. There are 7 voltage islands and 28 frequency islands on the SCC. Each tile on the SCC is a one frequency island (24

Voltage (Volts)	RCCE Maximum Frequency (MHz)
0.7	460
0.8	598
0.9	644
1.0	748
1.1	875
1.2	1024
1.3	1198

Table 2.3 Voltage and Maximum Frequency Values [77]

total), while MCs—all 4 MCs make one island—VRC, the mesh network, and the SIF make up the rest of the islands [58]. In contrast to frequency islands, whereby each tile is an island, six voltage islands contains 2×2 array of tiles (4 tiles per island) each, while the entire mesh network is regarded as 7^{th} island.

Using the VRC the voltage can be adjusted on a per island granularity. With 6.25 μV steps, voltage can be set between 0.7 and 1.3 volts. The maximum frequency is dependent on the current voltage level and can be varied between 100 to 800 MHz. The frequency of the mesh network can be set at 800 or 1600 MHz, while for MCs it can be set at either 800 or 1066 MHz. Although the MCs and mesh regarded as frequency islands, unlike frequency islands made of tiles their frequency cannot be adjusted dynamically. The frequency of the MCs depends on the frequency of the mesh network. If the mesh network is running at 1600 MHz, MCs can be set at either 800 MHz or 1066 MHz. If mesh network is running at 800 MHz, the frequency of the MCs is set to 800 MHz.

In order to prevent any damage to the SCC chip, scaling the frequency up requires a corresponding change in the voltage. The official SCC documentation [77] and source code of RCCE [78, 97, 131] (a message passing library by Intel) provides a table with the maximum frequency allowed for each voltage level. However, there were stability issues with certain voltage levels and the maximum allowed frequency [94]. As reported in previous studies [42, 53] we also found that some of the voltage frequency pairs resulted in cores becoming unstable or crashing all together (which requires a hard reset of the platform). Through experimental evaluation we have calibrated the voltage and frequency pairs that work for our SCC unit. Figure 2.16 shows our calibrated voltage-frequency levels and the one provided in the RCCE source code. For the frequency range marked as improved, we were able to lower the voltage required, while for the unstable range we had to increase the volts required to make the core

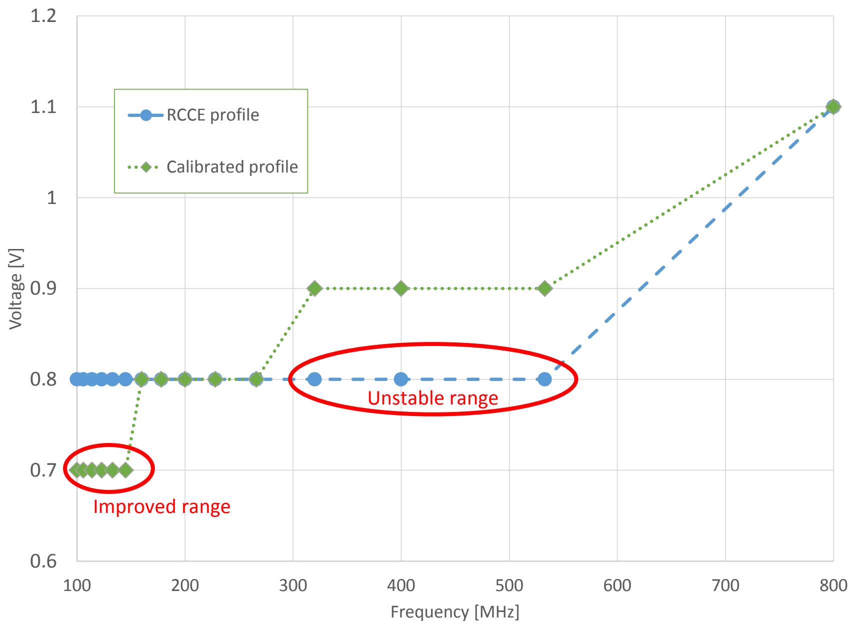


Figure 2.16 Frequency-Voltage Profile of the SCC

stable and prevent crashes. While the RCCE source code provides only two usable voltage levels, our calibrated profile provides four voltage levels.

The voltage and frequency islands enable parts of the SCC to be turned off or dialled down to a lower frequency to minimise power consumption. All adjustments are under the control of the application which could then set any group of tiles to a higher performance level for computationally intensive workloads and to lower the performance level for cores performing memory or I/O operations. Each core can change the voltage and frequency of any other core. This makes it easier to implement different power management schemes, e.g. one core controls the island or one core can control the entire chip. As a concrete example, RCCE [78, 97, 131] implements the first approach. In RCCE each voltage island has defined the "master" core that is responsible for the voltage and frequency scaling of that island. In the second approach, where one dedicated core is responsible for power management of the entire chip is described in [68, 69].

2.6 Chapter Summary

This chapter has laid the foundations for the thesis by providing a background of the topics covered in the context of this thesis. The thesis focuses on introducing energy awareness for RSPs on NoC architectures. An RSP is a reactive system with its internal implementation designed as a stream program. NoC is processor architecture in which multiple cores on a single chip are connected by some sort of network. For stream programming, we briefly described models, properties of stream programs, and some languages. Afterwards, we clearly defined what we meant by RSPs followed by a detailed introduction of the S-Net language. Next was an overview of the power and energy optimisation techniques. For the NoC section we began with examples of NoC architectures. The in-depth description of the SCC, an NoC with 48 cores and a main research vehicle for this thesis, concluded this chapter.

Chapter 3

Design and Implementation of the Execution Layer

We have covered the execution model of S-Net in Section 2.2. In this chapter we will cover the design decisions with a short description of implementation detail of ***L**ight-weight **P**arallel **E**xecution **L**ayer* (LPEL) for the ***S**ingle-chip **C**loud **C**omputer* (SCC). In § 3.1 we justify the need for ***H**ierarchical **Alloc**ator* (HALLOC). This is followed by a detailed description of the shared memory initialisation scheme and a comprehensive explanation of the HALLOC in § 3.2 and § 3.3 respectively. The changes required in order to use LPEL on the SCC is covered in § 3.4. We provide an account of how conductor and workers are initialised on the SCC in § 3.4.1, followed by synchronisation mechanism available in § 3.4.2. We conclude this chapter with a short summary in § 3.5.

Figure 3.1 illustrates the execution model for the SCC. Each core on the SCC has its own instance of S-Net ***R**untime **S**ystem* (RTS) and LPEL. The communication takes place over the shared memory provided by LPEL. Even-though each core has its own RTS and LPEL instance, the scheduling is done by the conductor core, while all the worker cores are responsible for invoking actual tasks to perform some computation.

3.1 Shared Memory Motivation

We described LPEL with ***C**entral **T**ask **Q**ueue* (CTQ) in Chapter 2.2.4.1. The conductor manages centralised task queue which contains tasks that are ready to be invoked. The communication between conductor and workers is provided by means of mailboxes and streams, both implemented on top of the shared memory. The LPEL has some requirements in terms of what services are provided by underlying hardware. For example LPEL needs support for kernel-level threading and context switching in user-

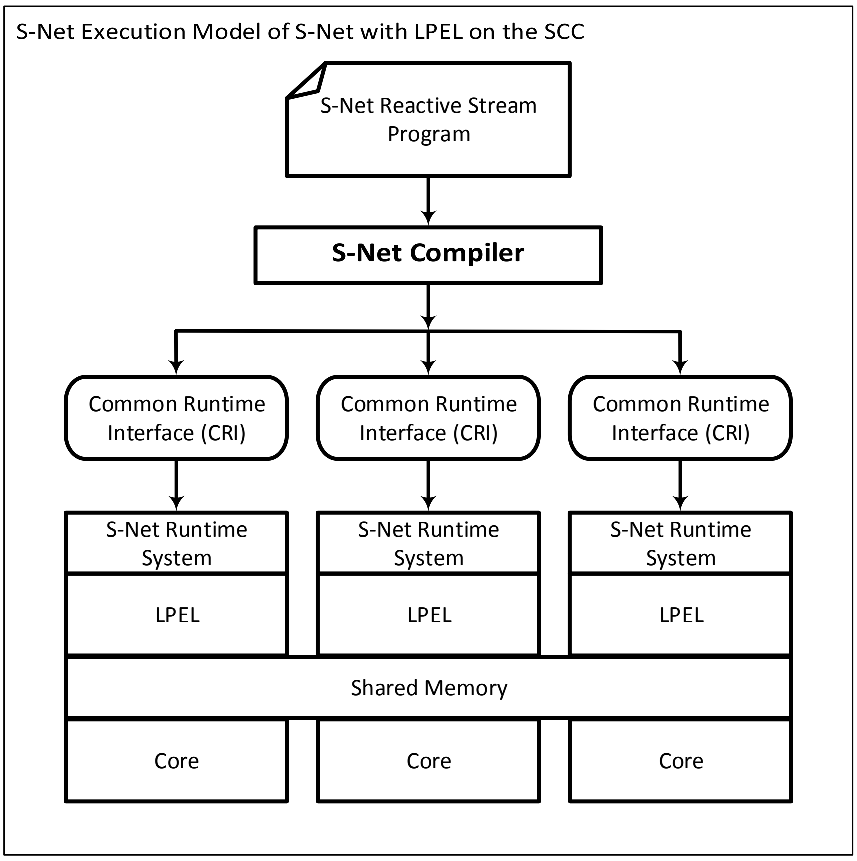

S-Net Execution Model of S-Net with LPEL on the SCC
S-Net Reactive Stream Program
S-Net Compiler
Common Runtime Interface (CRI)
Common Runtime Interface (CRI)
Common Runtime Interface (CRI)
S-Net Runtime System
S-Net Runtime System
S-Net Runtime System
LPEL
LPEL
LPEL
Shared Memory
Core
Core
Core

Figure 3.1 Execution Model of S-Net for the SCC

space. While the former is easily available, the latter poses a problem. For user-space context switch, we need the mechanism to transfer the whole context of the executing task from one core to another. This is a problem as LPEL uses the ***P**ortable **C**oroutine **L**ibrary* (PCL) [91] to provide user-space thread management. PCL provides basic light-weight, low-level mechanism for context-switching from user-space together with management functionality for coroutines. This whole stack of technology (PCL, LPEL and S-Net RTS) makes it more difficult as each task's execution context may add/remove different properties at each layer. Furthermore, our initial investigation confirms that the actual code that implements PCL/LPEL tasks contains reference (pointers) to parts of data held in different memory places. To transfer this whole information, either we have to build the mechanism that packs the whole execution context of a task to be transferred, which may not work due to the memory references or provide illusion of the shared memory in the SCC. Having a shared memory also means that the other functionality such as communication in terms of the mailboxes and streams would also work without much modification.

3.2 Shared Memory Initialisation

We know from § 2.5.4 that each core on the SCC has its own private memory that is accessible only to the core and some shared memory that can be accessed by any core on the scc. The division between shared and private memory is achieved by manipulating each core's ***L**ookup **T**able* (LUT) entries.

The default LUT mapping is listed in Table 2.2. There are 256 LUT entries, of which 0–40 are used by the ***O**perating **S**ystem* (OS) that is running on the core. Entries 192–255 are mapped to ***M**essage **P**assing **B**uffers* (MPBs), ***C**ontrol **R**egister **B**uffers* (CRBs) and so on. We can see that entries 41–127 and 132–191 are unused. Furthermore, entries 128–131 are used for the shared memory. If we combine all of these entries, we can use LUT entries 41–191 to create the shared memory.

To create the shared memory, we improve upon the memory hijacking technique used in the RCCE [76, 131] library. In RCCE, 4 LUT entries (128–131) are mapped to the same physical address-space range on all the cores. As each LUT entry points to a 16 MiB segment of physical memory, this mapping provides 64 MiB of memory which is shared between all the cores.

There are a few problems that we have to overcome in order to have a usable shared memory. Firstly, 64 MiB is not enough to deploy LPEL and some real-world ***R**eactive **S**tream **P**rograms* (RSPs). The problem of not having enough shared memory can be solved by using all the unused entries of the LUT, that is entries 41–191. As each entry points to a 16 MiB segment of memory, by using entries 41–191 we can create approx 2.5 GiB of shared memory.

The next problem to consider is: where will we get the physical memory to which our extended range of LUT entries will point (Don't forget that, by default only 4 of these entries point to the physical memory). As mentioned before, out of 256 entries of the LUT table only the first 41 entries point to the physical memory needed by the individual OS instances. So to obtain enough physical memory to configure as a shared memory, we need to disable 4 of the 48 cores on the SCC. This provides us with enough valid entries from these disabled cores to populate unused/extended entries of the LUT. In total we have 151 unused entries. Now, by disabling 4 cores we obtain $41 * 4 = 164$ entries in total. This is more than enough for our purpose, but as a consequence we lose 4 out of 48 cores on the SCC.

The next problem is that with memory mapped as described in the RCCE, you get a shared memory but the virtual address range is not the same for all the cores. Let's look at a small example: We have some physical memory that we would like to share between two cores. Now core-0 may map this memory at the virtual address starting from 0xb5815000 and core-1 may map at the virtual address starting from 0xb5700000, even though both are pointing at the same segment of physical memory (both cores invoke `mmap` system call to map memory independently). This increases the difficulty of creating complex data structures in the shared memory: a structure allocated in the shared memory might have a pointer pointing to another structure that is also allocated in the shared memory. Since the pointer stores an absolute address, that address is only valid for the core that allocated the memory for the structure, i.e. pointers are not globally valid. For example, core-0 allocated some memory that is pointed to by a pointer p, if we try to access this memory from core-1 the address that pointer p points to will not be a valid address, and as such it may cause data corruption and a segmentation fault in the most extreme case.

Of-course we can use offsets from the beginning of the address-space instead. That is, instead of passing pointer between cores, we can store the starting address of the shared memory for each core and then use this to calculate offset. As an example, core-0 and core-1 can store their start address (virtual address) of the shared memory in to a core local variable called *shm_base*. Now on core-0 instead of storing a pointer p, an offset pointer *offset_ptr* will be stored. This *offset_ptr* can be calculated as $offset\_ptr = p - shm\_base_{core-0}$. Now this *offset_ptr* is sent to core-1 where the pointer p is reconstructed as $p = shm\_base_{core-1} + offset\_ptr$. But this approach of using offsets from the beginning of the address-space would introduce an additional overhead and unnecessary complexity. In addition, it would prevent us from using any third party library if needed.

All of these problems can be avoided if we can somehow map the physical memory to the same virtual address range on all the cores. The system call `mmap()` creates a new mapping in the virtual address space of the calling process. We will look at some of the parameters that we are interested in here; in-depth description of the same can be found in [72]. When `mmap` is called, the address of the new mapping is returned as the result. The actual `mmap()` looks like;

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
off_t offset)
```

The parameters we are interested in are:

addr there are two options for *addr* parameter, if NULL is provided the kernel chooses the address at which to create the mapping. The kernel takes it as a hint where to place the mapping if *addr* is not NULL.

length is the size of mapping in bytes.

flags this flag decides if the memory is shared or not by using MAP_SHARED or -MAP_PRIVATE. In addition, there are some more flags that can be ORed. Out of those flags, we are only interested in a flag called MAP_FIXED; this flag forces the kernel to place the mapping at exactly the address provided by *addr* parameter (instead of taking it as only a hint).

fd the memory device drivers that provides the raw memory/file descriptor (in our case we use NCMDeviceFD which maps memory as uncached).

We now know that there are two steps to create a shared memory: the first step is to change default LUT entries; the second one is to map this newly available memory into the virtual address space of the programme. We will look at those two steps in detail.

3.2.1 LUT Remapping

We will look at how the LUT remapping is done using a concrete example. Since we are not interested in all 256 LUT entries, the following figures show only the relevant entries.

Figure 3.2 shows the default LUT entries for cores core-0, core-1 and core-14. For simplicity, we show the value for each LUT entry as a decimal number. However these numbers are real values, i.e. they were obtained experimentally by reading LUT entries of respective cores. As we can see, entries 0–40 are mapped as private, entries 41–127 and 132-191 are unused, and entries 128–131 are mapped as shared. We

Core 0

LUT #	Values	Maps To
0	6144	Private
1	6145	
2	6146	
:	:	:
38	6182	
39	6183	
40	6184	Private
41		Unused
42		
43		
:	:	:
125		
126		
127		Unused
128	6636	Shared MC0
129	45548	Shared MC1
130	268780	Shared MC2
131	307692	Shared MC3
132		Unused
133		
134		
:	:	:
189		
190		
191		Unused

Core 1

LUT #	Values	Maps To
0	6185	Private
1	6186	
2	6187	
:	:	:
38	6223	
39	6224	
40	6225	Private
41		Unused
42		
43		
:	:	:
125		
126		
127		Unused
128	6636	Shared MC0
129	45548	Shared MC1
130	268780	Shared MC2
131	307692	Shared MC3
132		Unused
133		
134		
:	:	:
189		
190		
191		Unused

●●●

Core 47

LUT #	Values	Maps To
0	307651	Private
1	307652	
2	307653	
:	:	:
38	307689	
39	307690	
40	307691	Private
41		Unused
42		
43		
:	:	:
125		
126		
127		Unused
128	6636	Shared MC0
129	45548	Shared MC1
130	268780	Shared MC2
131	307692	Shared MC3
132		Unused
133		
134		
:	:	:
189		
190		
191		Unused

Figure 3.2 Default LUT Entries for Cores 0, 1 and 47

Core 17 MC-0

LUT #	Values	Maps To
0	6595	Private
1	6596	
2	6597	
3	6598	:
4	6599	
:	:	:
36	6631	
37	6632	
38	6633	
39	6634	
40	6635	Private

Core 18 MC-1

LUT #	Values	Maps To
0	45302	Private
1	45303	
2	45304	
3	45305	:
4	45306	
:	:	:
36	45338	
37	45339	
38	45340	
39	45341	
40	45342	Private

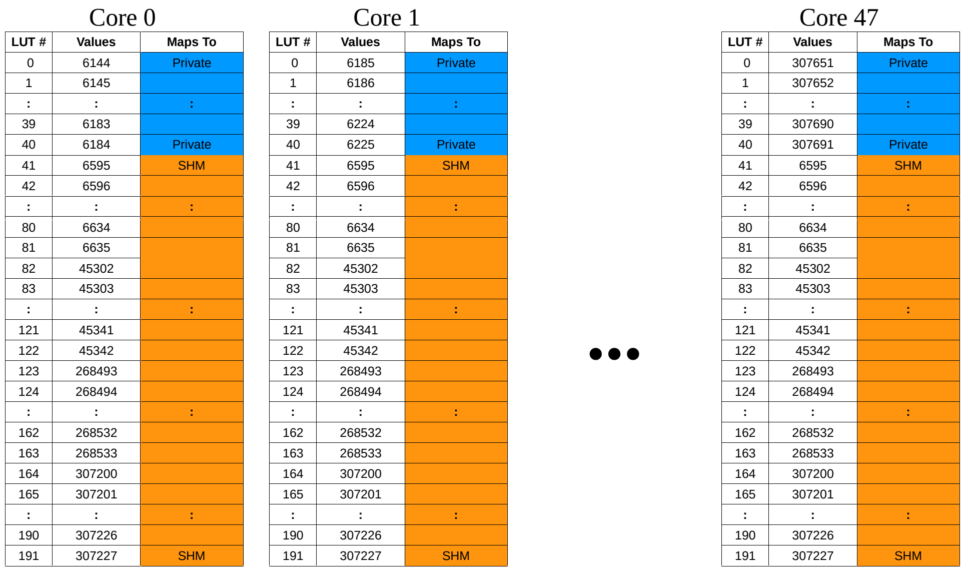
Core 29 MC-2

LUT #	Values	Maps To
0	268493	Private
1	268494	
2	268495	
3	268496	:
4	268497	
:	:	:
36	268529	
37	268530	
38	268531	
39	268532	
40	268533	Private

Core 30 MC-3

LUT #	Values	Maps To
0	307200	Private
1	307201	
2	307202	
3	307203	:
4	307204	
:	:	:
36	307236	
37	307237	
38	307238	
39	307239	
40	307240	Private

Figure 3.3 Private LUT Entries for Donor Cores 17, 18, 29 and 30



Core 0

LUT #	Values	Maps To
0	6144	Private
1	6145	
:	:	:
39	6183	
40	6184	Private
41	6595	SHM
42	6596	
:	:	:
80	6634	
81	6635	
82	45302	
83	45303	
:	:	:
121	45341	
122	45342	
123	268493	
124	268494	
:	:	:
162	268532	
163	268533	
164	307200	
165	307201	
:	:	:
190	307226	
191	307227	SHM

Core 1

LUT #	Values	Maps To
0	6185	Private
1	6186	
:	:	:
39	6224	
40	6225	Private
41	6595	SHM
42	6596	
:	:	:
80	6634	
81	6635	
82	45302	
83	45303	
:	:	:
121	45341	
122	45342	
123	268493	
124	268494	
:	:	:
162	268532	
163	268533	
164	307200	
165	307201	
:	:	:
190	307226	
191	307227	SHM

•••

Core 47

LUT #	Values	Maps To
0	307651	Private
1	307652	
:	:	:
39	307690	
40	307691	Private
41	6595	SHM
42	6596	
:	:	:
80	6634	
81	6635	
82	45302	
83	45303	
:	:	:
121	45341	
122	45342	
123	268493	
124	268494	
:	:	:
162	268532	
163	268533	
164	307200	
165	307201	
:	:	:
190	307226	
191	307227	SHM

Figure 3.4 Remapped LUT Entries for Cores 0, 1 and 47

decide which cores will be the donor cores (these cores will become un-usable) at the boot time—this means we only boot 44 out of 48 cores. This allows us to have at maximum 44 cores to deploy LPEL onto. Since we know which cores will be the donor cores, by using the pre-calculated table we can work out the values that will be written to unused LUT entries by each core.

Figure 3.3 shows the LUT entries for the donor cores that we will use for this example. For this particular example we take a core from each ***Memory Controller*** (MC). So, core-17, core-18, core-29 and core-30 belong to MC-0, MC-1, MC-2 and MC-3 respectively. There is no particular reason behind choosing these cores, any cores can be chosen. However, we have to keep in mind that if we get all the donor cores from the same MC, we may face congestion on the network due to all the cores trying to access data from the same MC. There can be multiple ways to choose donor cores, e.g., a core from each MC, cores from same voltage island, cores that are far away from the MC and so on.

Figure 3.4 shows LUT entries from same cores 0, 1 and 47, but after LUT remapping has been done. As we can see, there is no change in LUT entries 0–40 that points to private part of memory. Entries 41–191 have been changed now and point to the memory region that will be shared by all the cores. Now as we can see all the cores have the same value for the shared part. Entries 41–81 map the values from LUT entries 0–40 of core-17, entries 82–122 map the values from LUT entries 0–40 of core-18, entries 123–163 map the values from LUT entries 0–40 of core-29 and entries 164–191 map the values from LUT entries 0–27 of core-30. From core-30 we only map 28 entries as after that we run out of unused LUT entries (we only map until entry

191). This mapping happens in all the cores that will take part in program execution (of-course maximum 44 cores). For LUT remapping there is no need for any type of synchronisation or communication between participating cores, as we already know values that need to be written to LUT, which is the same for all the cores. Once again just for clarification, we map all entries from a donor core before moving on to the next, this is just for illustration purposes. We can map the number of entry from each donor core in a round robin manner; e.g., we take one entry from each donor core, or let's say three entries from each donor core. Table 3.1 illustrates this with some numbers. In case of mapping a single entry each time, large data may span multiple MCs, or in the case of mapping more entries each time, multiple cores may try to access the same MC to access data in turn.

3.2.2 Memory Mapping

Now we have completed LUT remapping, how do we actually map this memory? it's not very straight forward and there are some caveats we should be aware of. Listing 3.1 and Listing 3.2 show partial pseudocode of a function that initialises the SCC platform and the shared memory respectively.

When the SCC initialisation function is invoked, depending on the role (conductor or worker) of a core, a different part of the initialisation function code is executed. The conductor is responsible for mapping a shared memory and making the resulting virtual address where this mapping occurs available to workers. If the core is a worker, it will wait until the address to map memory becomes available. At this point, it is important to remember that the shared memory does not exists yet. We use MPB to store meta-data and flags that assist at this stage to provide the communication mechanism between cores. Lines 6–11 are executed by the conductor. The conductor first maps MPB memory and sets required meta-data and default values for flags. Next a call to the shared memory initialisation function `SHMInit` (described in Listing 3.2) is made, at this point *v_addr* is NULL (`SHMInit` will set the value to *v_addr* if memory mapping is successful). The resulting virtual address is copied to MPB and instruction CL1INVMB is executed to make sure that it is written to MPB and not stuck in the ***W**rite **C**ombine **B**uffer* (WCB). In next step, flag *flag_conductor_init* is set to indicate the conductor has finished mapping the memory and the virtual address is in MPB and then MPB memory is unmapped. Of-course the MPB memory is mapped again at a later stage in the SCC initialisation function; this avoids any possible conflict betweenthe shared memory mapping and any other mappings, e.g., MPBs, CRBs and atomic registers.

LUT Entry to Map	Donor-Core – LUT Entry Taken	
	One Entry Per Donor Core	Three Entry Per Donor Core
41	C-17 – 0	C-17 – 0
42	C-18 – 0	C-17 – 1
43	C-29 – 0	C-17 – 2
44	C-30 – 0	C-18 – 0
45	C-17 – 1	C-18 – 1
46	C-18 – 1	C-18 – 2
47	C-29 – 1	C-29 – 0
48	C-30 – 1	C-29 – 1
49	C-17 – 2	C-29 – 2
50	C-18 – 2	C-30 – 0
51	C-29 – 2	C-30 – 1
52	C-30 – 2	C-30 – 2
53	C-17 – 3	C-17 – 3
54	C-18 – 3	C-17 – 4
55	C-29 – 3	C-17 – 5
56	C-30 – 3	C-18 – 3
57	C-17 – 4	C-18 – 4
58	C-18 – 4	C-18 – 5
59	C-29 – 4	C-29 – 3
60	C-30 – 4	C-29 – 4

Table 3.1 Example of LUT Mappings with Different Number of Entries in Round-robin Manner

```
Function SCCInit()
  Set v_addr = NULL
  Set flag_conductor_init = False

  If coreID equal Conductor
    Map MPB memory
    Call SHMInit(v_addr)
    Copy value of v_addr to MPB
    Invoke CL1INVMB
    Set flag_conductor_init = True
    Unmap MPB memory
  Else
    Map MPB memory
    While flag_conductor_init equal False Do
      Wait for some time
    End While
    Copy value from MPB to v_addr
    Unmap MPB memory
    Call SHMInit(v_addr)
  End If
```

Listing 3.1 Partial Pseudocode of the SCC Platform Initialisation Function

The lines 13–19 are executed by worker cores. Similar to the conductor part, workers also map and unmap MPB memory at the beginning and at the end for the same reasons. After mapping MPB memory, the worker busy-waits on a *flag_conductor_init*. Once *flag_conductor_init* is set, the worker can retrieve the virtual address (and store in *v_addr*) to map a shared memory from the MPB and calls `SHMInit` with *v_addr* as a parameter.

Similar to the SCC initialisation, a shared memory initialisation also executes slightly different code based on the role (conductor or worker) of a core as can be seen in Listing 3.2. The different branch of code is taken based on value of the *v_addr*. From Listing 3.1 the conductor will always have NULL for *v_addr*, where as for workers it will contain the virtual address. As can be seen in Listing 3.2 we map the memory twice for conductor, this is one of the caveats. Again this has something to do with preventing a mapping conflict in address ranges. First we create an anonymous mapping by specifying *addr* as NULL and without MAP_FIXED flag in `mmap`. This allows the kernel to choose an address for the mapping that does not overlap any other mapping (there are no mapping at this point apart from MPB anyway). Lines 9–13 perform error checking and either exits the program in case of error or unmaps the recently mapped region. In next step, we call `mmap` again but this time we **do not** create anonymous mapping. We specify the address that was acquired in previous `mmap` call as the value for *addr* and also use MAP_FIXED flag to force the kernel to make new mapping at the provided address. Again we check this new mapping was successful as can be seen in lines 18–22, and set *v_addr*. On worker part, `SHMInit` is already called with address to make fixed mapping, and as such `mmap` is called only once and usual error checking is performed (lines 24–29). At the end, variables shmStart and shmEnd are set with values. We use these variables to implement the address validation technique described in Section 3.3.3.

Now some more information about why we perform two `mmap` calls for the conductor.

> If the memory region specified by *addr* and *len* overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded [24].

This can cause problems when we provide a fixed address to `mmap`; conflict with existing mappings can occur resulting in hard to trace bugs (we have experienced this first hand). It becomes hard to trace this kind of memory corruption bug as it may be the case that mapping discards only part of existing mapping. As an example, there are 24 mappings for MPB, one for each tile (MPB is physically distributed on a tile basis). In the worst case we may overwrite part of this mapping, so in the case we think

```
Function SHMInit(v_addr)
  Set local_v_addr1 = NULL
  Set local_v_addr2 = NULL

  IF v_addr equal NULL
    Call mmap() with addr as NULL
    Set local_v_addr1 = return value of mmap()

    If local_v_addr1 equal NULL
      Print error message and exit program
    Else
      Unmap memory mapped at local_v_addr1
    End If

    Call mmap() with addr as local_v_addr1 and MAP_FIXED flag
    Set local_v_addr2 = return value of mmap()

    If local_v_addr2 equal NULL
      Print error message and exit program
    Else
      Set v_addr = local_v_addr2
    End If
  Else
    Call mmap() with addr as v_addr and MAP_FIXED flag
    Set local_v_addr1 = return value of mmap()

    If local_v_addr1 equal NULL
      Print error message and exit program
    End If
  End If

  Set shmStart = v_addr;
  Set shmEnd = base + shared memory size;
```

Listing 3.2 Partial Pseudocode of a Shared Memory Initialisation Function

that we are writing to MPB we may in-fact be writing to uncached shared memory resulting in data corruption or, even worse, nearly untraceable memory access bugs and race conditions. Now in the case of mapping twice, we ask the kernel to provide us with an address first time; this automatically prevents any conflict with existing mappings; and in second stage we use this address to make fixed mapping that we can guarantee to be conflict-free.

3.3 HALLOC: A Hierarchical Memory Allocator

Once the shared memory has been initialised, the next question is how to use this shared memory. Any static objects (e.g. variables) are allocated statically at the link time and will reside in the program's heap space. Since heap space resides in private memory of a process/core and is not accessible to other cores, the focus of our approach is restricted to the ***Dynamic Storage Allocation*** (DSA) only.

Calling standard `malloc` will allocate space in the core's private segment rather than in the shared region of memory. We have written our own `malloc` and `free` functions that are based on K&R `malloc` and `free` [71] to address the issue.

By using the LUT entries to create a shared memory, all cores share the same view on the memory (with memory mapped at the same address range on all the cores). There are multiple ways we can allocate this shared memory to cores. For convenience, from now on we will refer to the shared memory as a *SHM-Arena*

The first approach is to have a global allocator that allocates memory to each core as and when requested. The benefit of this scheme is that it is very simple. As the SHM-Arena is accessible by all the cores, to allocate memory from it each core has to grab the lock, allocate memory and then release the lock. The lock is necessary as we do not want meta-data within the memory allocator to be corrupted due to simultaneous accesses by multiple cores. The drawback of this approach is that locking/unlocking creates unnecessary contention and adversely impacts the performance.

In the second approach, the SHM-Arena is divided into blocks of equal size and then each core can locally manage its block. The problem with this approach is that not all tasks need the same amount of memory. For example in RSPs there is a global input/output task, which requires very small amount of memory. In this case, if we distribute equally-sized blocks of memory to all the cores taking part in execution it will be a waste of resources.

To alleviate this problem, we can fuse the first and second approach to create a hybrid allocator. More concretely, we propose a hierarchical memory management scheme that is composed of two principal components: (a) a "global allocator" component which allows memory management at the SCC level and (b) a "local allocator" component which is associated with each core on the SCC and allows memory management at the core level.

There is only one global allocator that is shared by all the cores. This global allocator allocates blocks of memory from SHM-Arena as and when required by participating cores. In contrast, each core has its own local allocator, responsible for managing (allocation/de-allocation) memory that has been made available by global allocator. When the memory available is depleted and the local allocator is unable to fulfil the request for memory allocation, a core will request another block of memory from the global allocator. In case the global allocator is also exhausted of available memory, the program will simply terminate due to out of memory error (our SHM-Arena is approx 2.5 GiB). Any program that requires more than 2.5 GiB of memory at any given point in time can not be executed with our approach, we regard this case as a limitation of hardware/system (as in practice, for any given platform, there is a limit to the amount of memory that is available).

For clarity of exposition, from now on we refer to the block of memory that is allocated by the global allocator from SHM-Arena as *super-chunk* and when this super-chunk becomes available to the local allocator, we refer to it as *chunk*. Since our DSA is *hierarchical* by nature, we will refer our allocator as HALLOC. The functionality to create shared memory—as described in Section 3.2— is also part of HALLOC.

3.3.1 HALLOC Design Criteria and Concepts

Before we divulge into the details, we present some design criteria of HALLOC;

Explicit (de-)allocation An application has to explicitly call the functions provided by HALLOC to allocate/release memory e.g. `malloc` and `free`.

No reallocation This is true at both levels of hierarchy. At the global allocator level SHM-Arena (single large block of memory) is fixed size, and can not be increased by system level calls e.g. sbrk(). At the local allocator level, once memory is allocated for requested size it can not be increased or decreased e.g. by using `realloc`.

Low fragmentation Immediate coalescing is employed to decrease fragmentation.

Immediate coalescing When a chunk of memory if freed/de-allocated, it is merged straight-away with any available neighbouring free chunks to build up a larger free chunk if possible.

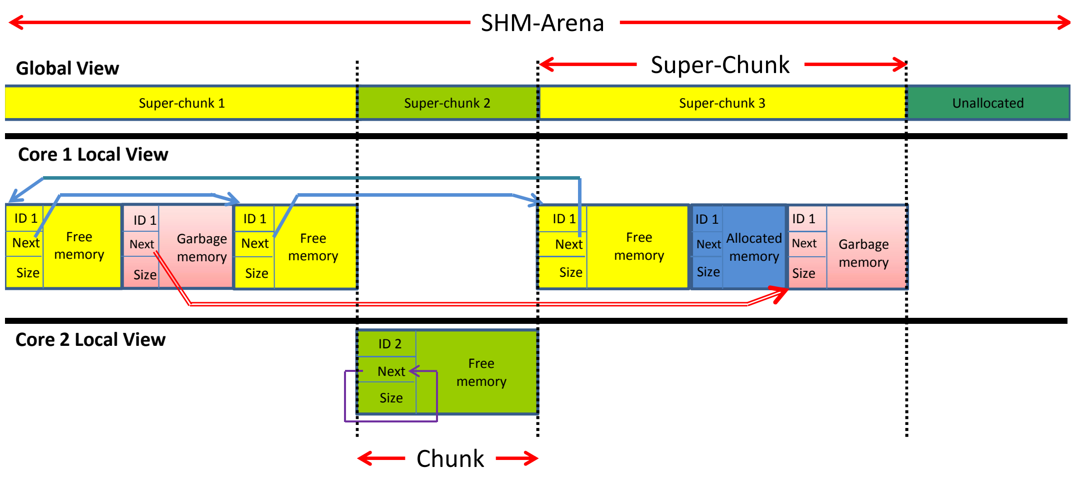


Figure 3.5 Shared Memory Layout

Deferred coalescing In the case where immediate coalescing is not possible, employ deferred coalescing instead. In deferred coalescing, chunk is **not** merged immediately (it takes un-specified time before coalescing happens) with adjacent free blocks.

Ownership The concept of ownership is essential to implement preventative measures for memory blowup problems.

Prevent blowup blowup [17] is a special kind of fragmentation, where required memory is fixed, but the memory consumption still grows with time, ultimately leading to out of memory even though there is some memory that can be made available to use.

First-Fit strategy Ensures that allocations are quick, reduces fragmentation and is easy to implement. To keep implementation simple we use same a first-fit strategy to allocate memory chunk of any size (of-course up to available memory size).

Memory is not zeroed Reduces the overhead of zeroing memory when allocating. In our case, only one application will be running at any given time and as such we can afford not to clear memory while allocating.

super-chunks are not freed To keep implementation simple at this stage, we do not free super-chunks, as we already free chunks it makes sure that cores have enough memory available.

Figure 3.5 depicts the view of shared memory from the perspective of different cores. In addition, Figure 3.5 also depicts SHM-Arena, super-chunk and chunk. SHM-

Arena is one big block of memory which is composed of multiple super-chunks and some un-allocated memory. When a core makes a request to the global allocator it receives a super-chunk. The super-chunks which are managed by a core may not be contiguous. Furthermore, as soon as super-chunk is made available to a core for the allocation it becomes a chunk (i.e. super-chunk is a concept that makes explanation simple). The core keeps its chunks as a circular linked list of free memory. For example, the core-1 manages the super-chunk 1 and 3. If we look at the core 1 view of these super-chunks then each super-chunks is composed of multiple chunks. We have a concept of *ownership* as well. In our example the core-1 owns super-chunk 1 and 3 (and as such all chunks within), where as the core-2 owns super-chunk 2.

The ownership concept is important to prevent the problem of blowup. Simply put, blowup is where memory required is fixed, but the memory consumption still grows with time. This can be illustrated with an example of producer-consumer relationship that is quite a common pattern in programming. Let's consider a program in which a producer repeatedly allocates a same size chunk of memory and gives it to a consumer which frees it. Now if the memory freed by the consumer is un-available to produce, even-though memory required for each allocation never changes consumption of memory will grow with time. Now since we know who the owner of any given chunk is, we can make this chunk available to its owner once it's de-allocated. This is the sort of model for RSP applications we are targeting; in RSP we refer to producer/consumer as source/sink. We refer to de-allocation as *remote free*, if de-allocation of a chunk is performed by a core that is not the owner i.e. core did not allocate the chunk in question. In contrast, de-allocation of chunk is referred to as *local free*, if chunk owner and the core performing de-allocation is the same.

The local allocator in HALLOC manages two lists: first to keep track of free storage, known as *free list*. The second list is the *garbage list* to keep track of garbage storage that needs to be added back to the free list. The free list is only accessed by an owning core, while the garbage list can be accessed by multiple not-owning cores. Even-though we have hierarchy of allocation and de-allocation function, all of the gory details is hidden behind only two functions that are exposed to the end user which are; allocation function *scc_malloc* and de-allocation function *scc_free*.

3.3.2 Memory Allocation

As we already know, the memory allocation in HALLOC happens at two levels. We have a wrapper function called *scc_malloc* which is used by a program when it needs to allocated some storage. Internally scc_malloc calls either the global allocator function called *scc_malloc_global* or the local allocator called *scc_malloc_local*. We will start

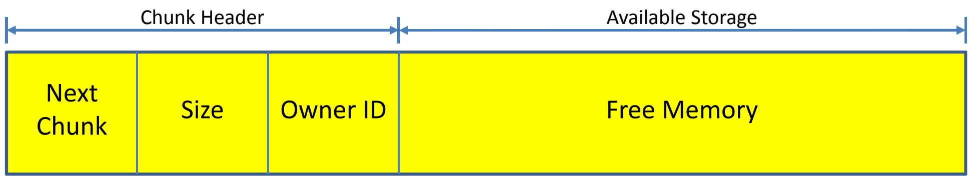


Figure 3.6 A Chunk with Header

with data structure of chunk and then look at the allocators. We do not provide any in-depth details of the local allocator scc_malloc_local as it is a “first fit” algorithm from [71].

Information required in order to manage super-chunk/chunk is kept in the data structure called *chunk header*. The information in chunk header is arranged in three fields: first is a pointer to next chunk in free list or garbage list; second is size of chunk i.e. the memory which is available for storage; and third is owner-id. We use core-id as owner-id for chunks. The chunk header is followed by the memory area that is available for storage to end user. The pointer returned by allocator points at the storage area, not at the chunk header itself. Chunk is depicted in Figure 3.6. Since we know exactly how big the header is, we can always retrieve information needed from the header by employing pointer arithmetic. In order to keep track of free storage available, the local allocator uses the *next chunk* field from the header to create circular linked list called *free list* composed of all the chunks. Once a chunk is allocated, it is removed from the free list and the *next chunk* field is set to NULL.

Algorithm 1 Algorithm *scc_malloc_global* to Allocate *m* Bytes from SHM-Arena

```
Require: $m \leq$ SHM-Arena_size        ▷ size of un-allocated memory in SHM-Arena
1: $tas\_lock()$
2: super-chunk $\leftarrow$ SHM-Arena_ptr
3: super-chunk_next $\leftarrow$ NULL
4: super-chunk_size $\leftarrow m$
5: super-chunk_owner-id $\leftarrow$ core-id
6: SHM-Arena_ptr $\leftarrow$ SHM-Arena_ptr $+(m+header\_size)$
7: SHM-Arena_size $\leftarrow$ SHM-Arena_size $-(m+header\_size)$
8: $tas\_unlock()$
9: Return super-chunk
```

Algorithm 1 describes the global allocator. It is a very simple allocator that keeps track of the size of the SHM-Arena and the starting point of un-allocated memory in SHM-Arena as meta-data. When a request for memory allocation is made by a core, the global allocator performs three steps. First it checks if there is enough memory to allocate. If there is enough memory available, then it continues to the next step—otherwise it returns an error. Next it uses a lock to avoid any corruption of meta-data. Finally it allocates the required size of super-chunk with appropriate header information, and then adjusts the size and starting point of the un-allocated memory in SHM-Arena before releasing the lock. This global allocator can be called by multiple cores simultaneously and as such we will need a lock to guard the critical section. We use ***Test-and-Set*** (T&S) register available on the SCC to implement this lock. Locks implemented using T&S provide atomicity at the SCC level, which is a mandatory property for the global allocator.

Algorithm 2 Algorithm *scc_malloc* to Allocate *n* Bytes from Free List of Chunks

```
 1: mutex_lock()
 2: memptr ← scc_malloc_local(n)                          ▷ K&R First-Fit malloc
 3: mutex_unlock()

 4: if memprt = NULL then
 5:     scc_free_garbage()                           ▷ triggers garbage list clean-up
 6:     mutex_lock()
 7:     memptr ← scc_malloc_local(n)
 8:     mutex_unlock()

 9: if memprt = NULL then
10:     super-chunk ← scc_malloc_global(m)
11:     mutex_lock()
12:     scc_free_local(super-chunk)            ▷ make it available to local allocator
13:     memptr ← scc_malloc_local(n)
14:     mutex_unlock()

15: if memprt = NULL then
16:     Not enough memory available, return Error
17: else
18:     Return memptr
```

Algorithm 2 allocates *n* bytes of memory from the free list. The local allocator scc_malloc_local implements a “first fit” allocation strategy. As interested readers can find a detailed description with sample code in [71], we refrain from listing any code here. Initial implementation of scc_malloc_local is not thread-safe. It is therefore protected by a lock (lines 1–3). Since scc_malloc_local is a core local allocator, mutex lock is sufficient enough. The following situations can occur when scc_malloc is called:

Lines 1–3 allocation request is successful and required memory is allocated by scc_malloc_local, no further action is required, scc_malloc returns pointer to allocated memory.

Lines 4–8 First call to scc_malloc_local returns NULL, we may have some chunks that are already marked for de-allocation waiting in the garbage list. In this case, we trigger garbage list clean-up by invoking scc_free_garbage. Once garbage clean-up is complete, we try again to allocate required memory with scc_malloc_local. In case of success, pointer to allocated memory is returned.

Lines 9–14 Second call to scc_malloc_local also return NULL. In this case we request super-chunk from global allocator scc_malloc_global. Once we have super-chunk before it can be used we have to make it available to scc_malloc_local by adding it to the free list. This is achieved by invoking scc_free_local. we try one more time to allocate required memory with scc_malloc_local. In order to ease the contention on the global allocator, the core always requests m bytes of memory where $m > n$.

Lines 15–18 Since asking for super-chunk is our last attempt to allocate memory. Failure in this case means there is not enough memory available and the program exists with out of memory error. Pointer to allocated memory is returned in case of a success.

As can be seen from Algorithm 2, all the calls to scc_malloc_local and scc_free_local are protected by a mutex from the ***P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**]* (POSIX) thread library to make them thread-safe.

3.3.3 Memory De-allocation

This hierarchical allocation means we will also need a hierarchical de-allocation, but at this point we do not free local chunks back to SHM-Arena or super-chunk. The reasoning behind this decision is that even if we send back chunks to SHM-Arena we may still see fragmentation. This fragmentation is due to the fact that once a super-chunk is added to local allocator it is broken down in to even smaller chunks (as and when new allocation request arrives); this will prevent us performing coalescing at a global level. Now, as we already mentioned earlier, there are two types of deallocation; we refer to them as *local free* and *remote free*.

Each core maintains a garbage list containing chunks marked for de-allocation. In the case of remote free, a chunk is added to the garbage list of owner core.

Again in similarity to allocation functions, we have a few functions that deal with de-allocation of memory allocated with scc_malloc. Here we list all of the de-allocation functions:

scc_free A wrapper function and counterpart to scc_malloc. This wrapper hides all the details of de-allocation from a programmer and calls the appropriate de-allocation function available internally.

scc_free_local A local de-allocator and counterpart to local allocator scc_malloc_local, and as expected it is also described in [71]. Local de-allocator scc_free_local returns the chunk to be freed to the core local free list and provides immediate coalescing functionality. This function is called in case of local free and only by owner core.

scc_free_remote When chunk is de-allocated using remote free, scc_free_remote is called to add the chunk to appropriate garbage list. This function is called by any core that is not owner of a chunk that is being free. In the case of remote free, deferred coalescing takes place to reduce fragmentation.

scc_free_garbage This function is called to de-allocated chunk that are placed in garbage list of a core.

Algorithm 3 Algorithm *scc_free* to De-allocate Memory Pointed by p

```
1:  if not (shmStart ≤ p ≤ shmEnd) then
2:      OS_standard_free(p)                       ▷ p points to private memory
3:      return

4:  if owner-id = core-id then
5:      mutex_lock()
6:      scc_free_local(p)                         ▷ K&R free
7:      mutex_unlock()
8:  else
9:      air_lock()
10:     scc_free_remote(p, owner − id)            ▷ add chunk to garbage list of owner core
11:     air_unlock()
```

We know at which address SHM-Arena starts and ends from Listing 3.2. We can use this information to employ a simple protection mechanism to check if an address is valid. An address is a valid shared memory address if condition $shmStart \leq address \leq shmEnd$ is satisfied (line 1). If the memory pointed by p was allocated in private region using standard `malloc` (`malloc` by OS), then we need to free it using standard `free` (line 2). This may happen plus it's a good protection measure. In case when p was allocated by scc_malloc there are two actions which can be performed. In the first case, owner-id from chunk header and core-id are compared and found to be same (line 4). Then function scc_free_local, which is the standard free function corresponding to scc_malloc_local is called. All the calls to scc_free_local are protected by a mutex lock to ensure thread-safe operation (lines 5–7). Immediate coalescing is also performed by scc_free_local, detailed description and implementation details can be found in [71]. In second case where id mismatch happens, a call to scc_free_remote is performed. Since scc_free_remote can be invoked by multiple cores simultaneously, it is protected by a lock implemented using the ***A**tomic **I**ncrement **C**ounter* (AIR), described in Section 2.5.3. Since implementation of scc_free_remote is very simple, we omit any code/pseudocode. In-short each core maintains an array with each element pointing to garbage list of other cores, where index of an element is equal to the core-id/owner-id. A chunk being remote freed is simply inserted at the beginning of a garbage list.

Algorithm 4 Algorithm *scc_free_garbage* to De-allocated Memory from Garbage List

```
 1: $air\_lock()$
 2: glfirst $\leftarrow$ garbage_list                          ▷ copy the garbage list
 3: garbage_list $\leftarrow$ NULL                        ▷ make the garbage list empty
 4: $air\_unlock()$

 5: $mutex\_lock()$
 6: while $glfirst \neq NULL$ do
 7:     glnext $\leftarrow$ chunk after glfirst in list
 8:     scc_free_local($glfirst$)                                  ▷ K&R free
 9:     glfirst $\leftarrow$ glnext
10: $mutex\_unlock()$
```

Algorithm 4 describes how the core de-allocates the chunk which was added to its garbage list by other cores. This algorithm is executed during the garbage clean-up process that is invoked by scc_malloc when the first attempt to allocate memory fails. there are two stages in scc_free_garbage;

Lines 1–4 We protect this region with a lock implemented with AIR to protect the garbage list from being corrupted due to the concurrent access by other cores. Once lock is acquired we copy garbage list to local list and set garbage list to be NULL (to mark it empty). This is an inexpensive operation as the garbage list is a linked list so that only copying pointer to first chunk is enough. This strategy reduces the time that other cores have to wait in order to access the garbage list while the owner core is in its clean-up routine.

Lines 5–10 Here, since we are accessing free list (accessed only by owner core), mutex is enough to protect from any concurrent access. Then we simply loop through the copied garbage list until the end and add each chunk to the free list by calling scc_free_local.

3.4 LPEL on the SCC

Once we have HALLOC we have to make some changes to get porting of LPEL on the SCC complete. Since LPEL was originally designed for shared memory systems where all the cores are running under the same instance of OS, only one instance of S-Net RTS and LPEL is required, whereas in our case this does not hold anymore. In addition, the previously used synchronisation mechanism does not work either i.e. we can not use PTHREAD mutex to achieve cross-core synchronisation on the SCC as each core runs its own instance of OS. Furthermore, if the user does not specify the desired number of workers on program startup we can not infer available number of cores that will participate (in multi-core/many-core shared memory system with single system image this can be achieved by querying OS). And lastly, we have to make sure that all the calls to standard `malloc` (for memory that is shared between cores) are replaced with scc_malloc.

3.4.1 Conductor/Worker Initialisation on the SCC

When deploying the LPEL on the SCC, it makes sense to create exactly as many workers as there are cores, as the cores of SCC are single-threaded. As there is no shared memory at the beginning, we can not just create conductor/workers on a single core and then distribute them amongst participating cores, which is the way it is done in a normal shared-memory machine. For this purpose, when the execution of a program starts, a configuration file is used to decide which core will be the conductor based on the physical core id.

As mentioned in Section 3.2, to create a truly global shared memory, all the cores have to map part of the program's address-space to the same virtual address range. Meta-data, including a flag necessary to establish communication between cores is located in a predefined location in MPB. This location is predefined and as such all the cores taking part in the computation will know this MPB location.

If a core is a conductor, it starts by initialising the shared memory, tasks, streams and the static parts of streaming network. If core is a worker, it will busy-wait on a flag located in MPB. Once the conductor has mapped the LUT entries and created the shared memory, it places the relevant LUT configuration in the MPB and sets the flag. Once the flag is set, the worker cores configure their LUTs to create mapping of the shared memory to the same virtual address range as the conductor.

The LPEL uses mailboxes to facilitate communication between conductor and workers. Each mailbox is protected by a lock to ensure that no messages are lost and operations are corruption free. Once we create a mailbox associated with conductor and each worker, it does not allow automatic establishment of communication channel between them. There are two ways this can be done: first the conductor and all workers will create their mailbox and then by using MPB communicate the location of their mailbox; the second way is to reserve some memory at the beginning of the shared memory and use it to statically allocate slots where each core will locate its own mailbox by means of core id. Since we have a fixed number of cores on the SCC, we opted to choose the second option. The conductor initialises all the mailboxes in the reserved section of the shared memory, all the workers has a private list of mailbox addresses that is populated with values by calculating offset from beginning of the shared memory.

Once the mailboxes are set-up, the workers request tasks to execute from the conductor and the conductor will fulfil these requests based on demand and task priority. When there are no more messages to be processed, the conductor sends a termination message to all the workers via the mailbox.

3.4.2 Synchronization Primitives

Having access to the atomic instruction to provide synchronisation and protect critical region is one of the requirement of the LPEL, which should be fulfilled by the underlying hardware. LPEL requires synchronisation at different points, some examples are:

- During the initialisation phase, the conductor/workers use a shared flag to synchronise LUT remapping and the shared memory creation.
- The meta-data of the global allocator in HALLOC is accessed/modified by conductor/workers concurrently.
- The meta-data of the local allocator in HALLOC need to be protected against concurrent access by multiple threads within the same core.
- The mailbox is an example of producer/consumer paradigm, where messages are added/removed from the queue concurrently. This queue needs to be protected against concurrent access to ensure messages are not lost and to avoid corruption of the queue.
- Streams are used to transfer messages between tasks. Streams are implemented as ***F**irst **I**n **F**irst **O**ut* (FIFO) buffers, and these buffers need protection to ensure

integrity of data (during reading/writing to the stream) and its associated counters (used to derive data demand for scheduling purposes).

We already use all the hardware registers provided by SCC for synchronisation. The MPB is used to store the shared flag. We use the T&S registers to implement locks that protect the meta-data of the global allocator. We use the atomic counter registers to implement locks to protect the garbage list and the mailboxes.

We still need more synchronisation primitives to protect the streams and for the core local memory allocator. For this purpose we use POSIX (PTHREAD) mutexes. The SCC runs an OS instance on each core, so we create mutexes with the *process shared* attribute set. This allows different workers to access the same mutex, as this access will be seen as it was accessed by different processes. In addition, the function that is called to acquire lock on this mutex i.e. *pthread_mutex_lock* is replaced by calls to the non-blocking version of the same function *pthread_mutex_trylock*.

3.5 Chapter Summary

This chapter provided a detailed description of the design and implementation of LPEL for the SCC. We have covered memory management mechanisms including, LUT remapping, memory mapping and shared memory allocation/deallocation. We also covered the concept and the design criteria of HALLOC, a novel hierarchical memory allocator. We provided an account of how conductor and workers are initialised on the SCC and how we use synchronisation primitives provided by the SCC hardware.

Chapter 4

Taxonomy of Resource-aware Execution

This chapter describes the resource model and different resource load scenarios. We study the influence of load balance on power consumption in § 4.2. We provide formal definitions of load ranges for ***R**eactive **S**tream **P**rograms* (RSPs) with different arrival rates in § 4.5. Furthermore, in § 4.6 these definitions are then extended with inclusion of ***D**ynamic **V**oltage and **F**requency **S**caling* (DVFS), and as such we take into account the impact DVFS has on available resource and system state.

4.1 Resource Model and Resource Management System

The *resource* is a reusable entity that is employed in order to function effectively. The model is concerned with two categories of resources; the first one is *available resources*, these are the actual resources provided by the system. The second one is *required resources*, these are the resources that are requested by the tasks in order to function. The resource pool can contain different types of resources, e.g., number of cores on a many-core machine, frequency at which these cores can operate, or available network bandwidth.

In the simplest term, a ***R**esource **M**anagement **S**ystem* (RMS) is defined as a service that manages a pool of resources that is available to optimise the system performance. The term system performance can indicate various metrics such as power consumption, energy consumption, throughput, responsiveness and fault-tolerance.

Let Ψ be a type of resource then $\Psi_{capacity}$ and Ψ_{demand} are availability of the resource and requested resource respectively. A pool with n types of resource is:

$$\Psi_{pool} = \{\Psi_1, \Psi_2, ..., \Psi_n\} \tag{4.1}$$

4.2 Influence of Load Balancing

The term *load* can be described as, the relationship between the amount of available (capacity) processing capacity of a resource and the amount required (demand) by the work that needs to be carried out on this resource. Considering this, *load balancing* can be defined as a strategy to distribute the load across the available resources almost uniformly such that optimal resource utilisation can be achieved.

There are two main approaches to achieve load balancing: static and dynamic. The static load balancing approach performs better in terms of complexity but it requires prior knowledge about resource capacity and workload. The dynamic load balancing approach is more complex but it takes into account the state of resources at the decision time and does not require prior knowledge of the workload.

We already know from our power equation Equation 2.3 that the dynamic power consumption of a core is approximately proportional to the product of its operating frequency by the square of its operating voltage:

$$P_{dynamic} \propto f \cdot V^2. \tag{4.2}$$

This means that a small change in voltage has a big impact on power consumption, as power consumed by a processor is directly proportional to V^2. A reduction in operating voltage generally also requires a proportional reduction in frequency [102, 109]. A quadratic reduction in power consumption becomes possible to obtain by varying the voltage along with the frequency.

On the one hand, for a chosen voltage there is a maximum frequency at which a circuit can run without problem; running faster than this maximum frequency will result in bad data. On the other hand, for a chosen voltage, setting frequency that is much below the maximum frequency allowed for this voltage level will result in waste of power.

Generally a more common approach for the many-core processors is to have islands that can have different frequency and voltage. These islands are also known as ***V**oltage-**F**requency **I**slands* (VFIs), where multiple computational units are arranged into groups to form a cluster/island. As an example, the ***S**ingle-chip **C**loud **C**omputer* (SCC) allows to set the frequency of each tile (of 2 cores), and the voltage of each island (of 8 cores). But as we previously noted, for a given voltage there is a maximum frequency to ensure problem-free operation. This mean if we set voltage accordion to the need of a core running with highest frequency in the island, we may be wasting power as 7 other cores may not require this high voltage level for safe operation.

This behaviour supports the argument that load balance is beneficial if we want to apply DVFS to a system with VFIs. As with a load balanced system, almost all the cores in an island will have uniform amount of work to do and as such we can reduce the frequency for all the cores in the island by the same amount, effectively opening a way to reduce voltage as well, with the assurance of safe operation of all the cores.

In contrast to load balancing, the core consolidation works by aggregating load on to a minimum number of cores. Maximum voltage and frequency are than set for these cores, but this may mean that cores on the island that do not have any work will waste power.

Maximal dynamic power savings are achieved on a perfectly load-balanced system, by reducing the speed of all islands by the same amount and avoiding different speeds on different cores. This can easily be seen for a two-core system:

We already know that for a chosen voltage there is a maximum frequency at which a circuit can run. As such, the dynamic power consumption of a processing unit is approximately proportional to the cube of its operating voltage, equivalent to the cube of its operating speed or frequency, resulting in $P_{dymanic} \propto speed^3 \propto f^3$. Now the power for a two-core system is:

$$P_{dynamic} = P_{dynamic_{core_1}} + P_{dynamic_{core_2}} \propto speed^3_{core_1} + speed^3_{core_2}. \tag{4.3}$$

Now let's assume we want to run the system at a momentary speed of:

$$s = speed_{core_1} + speed_{core_2} \tag{4.4}$$

which is determined by the current workload, hence:

$$Power \propto speed^3_{core_1} + (s - speed_{core_1})^3 = s^3 - 3s^2 speed_{core_1} + 3s\, speed^2_{core_1}, \tag{4.5}$$

which is minimal for $speed_{core_1} = s/2$. In other words, $speed_{core_1} = speed_{core_2} (= s/2)$ yields the minimal power consumption for any momentary system speed s. It is not hard to generalise this result for n cores.

This reasoning justifies chip-level DVFS for a dynamically load-balanced system, greatly simplifying the act of choosing the right voltages/frequencies in the presence of core-level load fluctuations, as any required load balancing is then automatically performed by the system.

4.3 Resource Underload

The term *underload* describes the condition where, capacity of the resource vastly exceeding demand for the resource, resulting in unnecessary waste. In other words, if the resource usage of the system can be reduced substantially without having any adverse effect on the performance, the system is said to be in an underload situation. The underload condition can occur due to many reasons, such as lack of available input to be processed by the system, or internal delays due to data dependencies resulting in most of the resource being idle. More formally the underload for resource Ψ_i is:

$$\Psi_{i_{capacity}} \gg \Psi_{i_{demand}} \tag{4.6}$$

4.4 Resource Overload

The term *overload* describes the condition when demand begins to exceed capacity and reaches a point where the resource is fully saturated such that it can hardly (or maybe can not) perform its specified function any more. To put it into simple words, overload is defined as the point when the demand for resource exceeds the capacity of that resource resulting in dramatically reduced functionality. The overload can be caused by a number of reasons, such as over estimation of resource capacity, under estimation of workload, or allocation of resource without considering load and capacity. More formally the overload for resource Ψ_i:

$$\Psi_{i_{capacity}} \ll \Psi_{i_{demand}} \tag{4.7}$$

4.5 Load Ranges for RSP with Different Arrival Rate

The RSP can be modelled as nodes connected by streams. This means they are similar to communication networks in the sense that they route messages from one source node to a destination node, with a slight difference being that in RSPs, some computation will be performed on the messages by the nodes. Despite the difference, performance of RSPs can be described by using same matrices used for communication networks, such as throughput and latency. We define throughput and latency as:

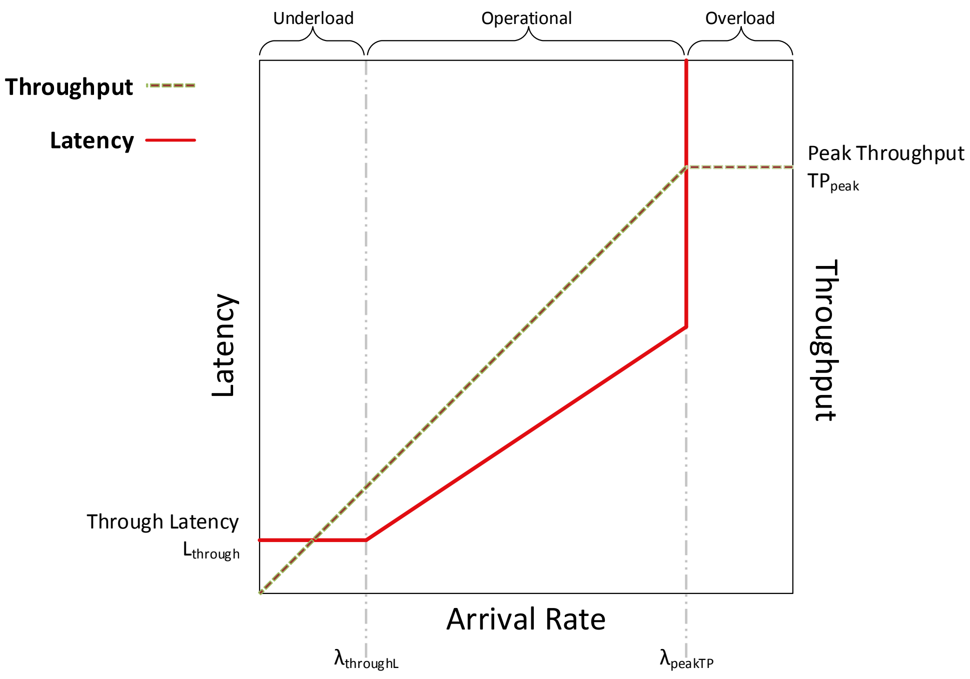


Figure 4.1 Load Ranges for RSP with Different Arrival Rate in Terms of Throughput and Latency [103]

Throughput is the rate of completely processing external input messages. Throughput is measured in messages per time unit.

Latency from a general point of view can be described as a time delay between the cause and the effect. In the case of RSPs, it would be the time difference between, the arrival of a message at the program, to when it is processed. Although we do not differentiate, the latency includes two parts; the amount of time a message has to wait in the input queue; and the amount of time it takes a node to process it and generate output. In order to derive a more stable view of the system, we take into account the *average latency* calculated as the arithmetic mean of latencies of all external input messages.

Let λ be the *arrival rate* at which external input messages arrive and $\overline{M_{cp}}$ be the average number of external input messages currently being processed by the RSP. Figure 4.1 shows three different states of the system that can be described as:

Underload Range When the first external input message arrives, it is the only message in the system and it is processed exclusively i.e. there is no resource sharing during message processing. At this point, the latency for every message processed will be at its smallest value, and so does the the average latency termed *through latency* ($L_{through}$). Since, all messages are being processed exclusively, the throughput will be equal to the arrival rate, $TP = \lambda$. The highest value of λ at which latency is still at its smallest value $L_{through}$ and $\overline{M_{cp}} < 1$ is $\lambda_{throughL}$.

The range $[0, \lambda_{throughL})$ is defined as the underload range. The observations that can be made in this range are:

- resource usage is $\Psi_{capacity} \gg \Psi_{demand}$
- for all value of λ within the rage $(0, \lambda_{throughL})$ the latency is almost constant i.e. $L_{through}$

Operational Range When $\lambda > \lambda_{throughL}$ and $\overline{M_{cp}} > 1$ the processing of messages in the system requires resource sharing or even wait in the input queues. If we have bounded $\overline{M_{cp}}$ then latency L is also bounded.

At the highest value of λ at which $\overline{M_{cp}}$ is still bounded, the system reaches its highest throughput termed *peak throughput* (TP_{peak}). The value of λ at TP_{peak} will be λ_{peakTP}.

The range $[\lambda_{throughL}, \lambda_{peakTP}]$ is defined as the operational range. The observations which can be made in this range are:

- λ_{peakTP} is equivalent to TP_{peak}
- resource usage is $\Psi_{capacity} \geq \Psi_{demand}$
- for all value of λ within the rage $[\lambda_{throughL}, \lambda_{peakTP}]$ the latency increases with increasing value of λ

Overload Range TP_{peak} is the maximum arrival rate the system can cope with. If the arrival rate exceeds this value i.e. $\lambda > TP_{peak}$, the system will get saturated and as such $\overline{M_{cp}}$ will become infinite, in turn making latency infinite as well.

The range $[\lambda_{peakTP}, \infty)$ is defined as the overload range. The observations that can be made in this range are:

- resource usage is $\Psi_{capacity} \ll \Psi_{demand}$
- for all values of λ that are higher than λ_{peakTP} the latency becomes ∞

4.6 Load Ranges for RSP with Different Arrival Rate and DVFS

Figure 4.2(B) and (C) extend the previous definition of the different load ranges with DVFS. Let's assume the architecture provides three discrete voltage and frequency

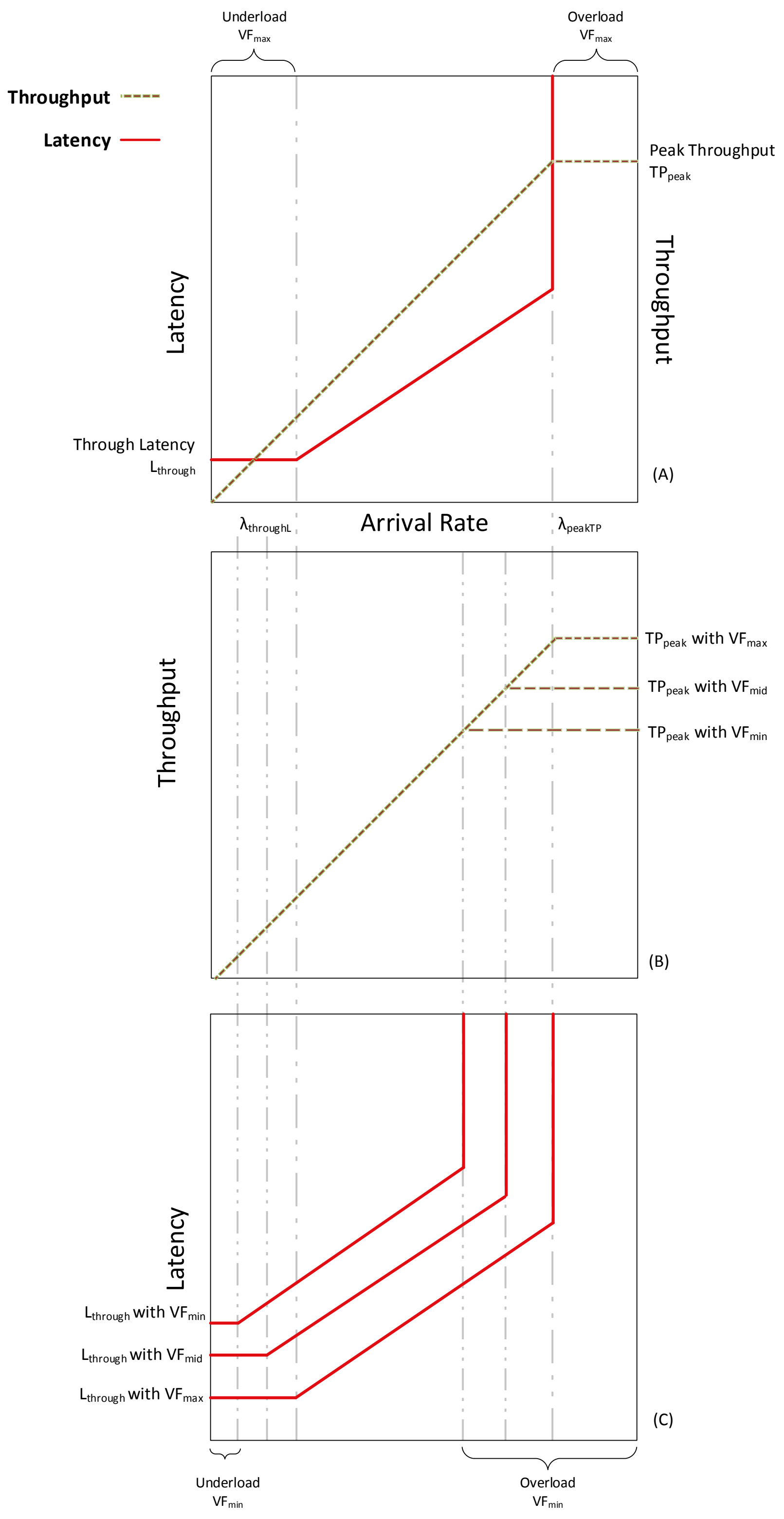


Figure 4.2 Load Ranges for RSP with Different Arrival Rate and Effect of the DVFS

settings: VF_{max} and VF_{min} are the upper and lower bounds of the available voltage and corresponding frequency value respectively, whereas VF_{mid} is some value in the middle.

Figure 4.2(B) shows the impact of DVFS on the throughput, where as Figure 4.2(C) shows the impact of different the ***V**oltage and **F**requency* (VF) settings on the latency.

Now the values of TP_{peak} and $L_{through}$ depend on which VF settings they are in. As in our example, we have three different values for VF, and as such we can observe three different values for throughput and latency:

VF_{max} For setting VF_{max} things do not change, they stay as they were in Section 4.5. Where $[0, \lambda_{throughL})$ would be underload, $[\lambda_{throughL}, \lambda_{peakTP}]$ is operational, $[\lambda_{peakTP}, \infty)$ is defined as the overload range.

VF_{min} The real difference we can observe is with VF_{min}. As Figure 4.2 (C) shows the $L_t hrough$ is increased as the clock cycle of the core slows down. Also the operation range is shorter than in the other two VF settings. What happens here is that by using VF_{min}, we actually narrow the underload range, while at the same time we expand the overload range. What this means is that we can move to VF_{mid} if we detect that we cannot keep up with λ. Similarly we can also come to VF_{mid} from VF_{max} if we can see that there are lots of resources waiting for work. Now we can extend this to as many VF settings as there are available, effectively creating as many overload, underload points which we can use to tune the DVFS.

As an example, for VF_{min}; the underload range, the operational range and the overload range can be:
$\left[0, \lambda_{throughL}^{VF_{min}}\right)$, $\left[\lambda_{throughL}^{VF_{min}}, \lambda_{peakTP}^{VF_{min}}\right]$ and $\left[\lambda_{peakTP}^{VF_{min}}, \infty\right)$.

Now the interesting thing is if we detect overload in this, we would switch to VF_{mid} effectively increasing clock cycles of the cores so that they run faster and in turn can cope with higher λ. This will bring the load into the operational range of VF_{mid}, thus enabling power increase only when required.

4.7 Chapter Summary

In this chapter we described the resource overload and underload in general and defined two matrices, throughput and latency. We use these matrices as indicators of performance and system state of RSPs. Next we formally described different load ranges specifically for RSPs, also taking into account dynamic arrival rate of messages.

Then we extended it with the effect of DVFS and described how knowledge of the state of the resources can be used with different VF settings to derive DVFS policies that can be used at runtime to drive DVFS decisions. This in turn allowed us to make the execution of RSPs more energy efficient, thus achieving our goal.

Chapter 5

RA-LPEL: Resource-Aware Execution Layer

This chapter describes the ***R**esource-**A**ware **L**ight-weight **P**arallel **E**xecution **L**ayer* (RA-LPEL), for the ***R**eactive **S**tream **P**rograms* (RSPs). We start with a recapitulation of power management functionality provided by the ***S**ingle-chip **C**loud **C**omputer* (SCC) hardware in § 5.1. The account on development of heuristics that we use to adjust power consumption of RSPs at runtime is given in § 5.2. This includes the ways that we detect overload and underload situations in RSPs. In addition, we define the ***D**ynamic **V**oltage and **F**requency **S**caling* (DVFS) policies as well. We discuss chip level DVFS and island level DVFS in § 5.3. We conclude this chapter with a short summary in § 5.4.

5.1 Recapitulation of Power Optimisation with the SCC

Power management on the SCC consists of three components which work with separate clocks and power sources: tiles, mesh network and memory controllers. The maximum frequency is dependent on current voltage level and can be varied between 100 to 800 MHz for cores. The frequency of the mesh and memory controllers can only be set at the SCC boot time, i.e. it cannot be changed by application at runtime like frequency of cores. The frequency of the mesh network can be set at 800 or 1600 MHz, while for ***M**emory **C**ontrollers* (MCs) it can be set at either 800 or 1066 MHz.

The cores on the SCC are arranged in voltage and frequency domains also known as islands. The frequency domains on SCC are fine grained i.e. each tile (made up of two cores) on the SCC is a frequency domain. The voltage domains are course grained, 2×2 array of tiles (4 tiles or 8 cores per island) make-up a voltage island. The entire mesh network is regarded as an island, although voltage change for this island is not allowed. Theoretically using the ***V**oltage **R**egulator **C**ontroller* (VRC), the voltage can be adjusted on a per island granularity. With 6.25 μV steps, voltage can be set between 0.7 and 1.3 volts, again support for very fine grained voltage change is not available, and as such we can only change voltage in 0.1 V steps. In order to prevent any damage to the SCC chip, scaling the frequency up requires a corresponding change in the voltage.

The voltage and frequency islands enable parts of the SCC to be turned off or dialled down to a lower frequency to minimize power consumption. All adjustments are under the control of the application which could then set any group of tiles to a higher performance level for computationally intensive workloads and lower the performance level for cores performing memory or ***I**nput/**O**utput* (I/O) operations. Each core can change the voltage and frequency of any other core. This makes it easier to implement different power management schemes; e.g. one core controls the island or one core can control an entire chip.

5.2 Heuristics to Adjust Power Consumption

In reactive stream processing, the system operates in direct response to inputs from its environment. If the load imposed by the environment varies, dynamic power management techniques can be used effectively to reduce the power consumption of such systems.

Our strategy is to reduce the frequency and voltage when there are more than enough resources and the system can easily cope with the input rate, and to increase them when the system becomes overloaded.

As mentioned in Section 2.1, stream programs consume messages from a dedicated entry stream and produce output messages to a dedicated exit stream. At runtime, we would ideally want a situation where:

$$ir \approx k \cdot or,$$

where ir is the rate at which messages enter the network, or is the rate at which messages leave the network, and k is the multiplicity of the network (cf. Section 2.1). More precisely, we want to avoid the case of $ir \gg k \cdot or$, which means that the system is internally accumulating messages, a behaviour that will eventually lead to memory exhaustion, whereas the case of $ir \ll k \cdot or$ is just impossible.

For simplicity, we will (w.l.o.g.) assume a multiplicity of $k = 1$, i.e. the network produces exactly one output message for each input message. In that case we can drop the factor k and obtain:

$$ir \approx or. \tag{5.1}$$

To achieve the near balance of Equation 5.1, we have to consider a dynamic input rate from the environment. The system must be equipped with sufficient computational resources to process messages fast enough under the maximal input rate ir_{max}, otherwise the system may become *overloaded*. In this case the output rate will stay close to some maximal achievable output rate or_{max}, i.e.

$$or \approx or_{max} \ll ir, \tag{5.2}$$

and messages will either start to accumulate inside system, or they will be dropped. On the other hand, if the input rate falls below or_{max}, the system becomes *underloaded*, i.e. :

$$ir \approx or \ll or_{max}, \tag{5.3}$$

and unused system resources may cause unnecessary power drain.

One way to deal with this situation is to use a platform where we can dynamically adjust the performance in response to demand. If we use voltage and frequency scaling to adjust the performance, or_{max} becomes dependent on the current voltage V and frequency f.

According to Equation 5.2 we can detect overload by checking, if the average output rate falls below the average input rate. The overload at the time t is given by:

$$ol(t) = \frac{ir(t) - or(t)}{ir(t)}. \tag{5.4}$$

5.2.1 Detecting Overload

In many streaming applications, the input rate depend on a stateful stochastic process. Thus it makes sense to consider the overload history to predict the potential future overload. We wanted to keep the overhead of our strategy low and therefore decided to use a simple ***E****xponential* ***M****oving* ***A****verage* (EMA) predictor, which is extremely fast to calculate and has minimal memory overhead. At a given time t, we calculate the predicted future overload $ol_{pred}(t+1)$ as:

$$ol_{pred}(t+1) = \begin{cases} 0 & \text{if } t = 0 \\ \alpha_{ol} \cdot ol(t) + (1-\alpha_{ol}) \cdot ol_{pred}(t) & \text{if } t \geq 1, \end{cases} \tag{5.5}$$

The smoothing factor $0 < \alpha_{ol} < 1$ is application-specific. A high value makes the prediction depend mostly on the recent history, whereas a low value makes the prediction more dependent on the long-term history. A suitable smoothing factor can be found by minimising an error measure, like the ***Sum of Square Errors*** (SSE), on representative sample data. This can either be achieved by graphically checking the fit for different parameters, or by applying a least squares approach [96].

5.2.2 Detecting Underload

We detect underload situations by observing used resources. More precisely, we examine the slack, e.g. the number of workers that are waiting for work to be assigned. Like in the overload case, we use an EMA to make a prediction about the future underload. At a given time t, we calculate the predicted future number of unused workers $ww_{pred}(t+1)$ as:

$$ww_{pred}(t+1) = \begin{cases} 0 & \text{if } t = 0 \\ \alpha_{ww} \cdot ww(t) + (1-\alpha_{ww}) \cdot ww_{pred}(t) & \text{if } t \geq 1, \end{cases} \tag{5.6}$$

Again, the smoothing factor $0 < \alpha_{ww} < 1$ is application-dependent. A high value makes the prediction depend mostly on the recent history, whereas a low value makes the prediction more dependent on the long-term history. A suitable smoothing factor can, again, be found by minimising an error measure on representative sample data.

5.2.3 DVFS Policy

Our policy for adjusting the frequency at runtime can be summarised by the following set of rules:

1. Increase the frequency of all the islands by one step, when the number of waiting workers is predicted to fall short of a given lower threshold ($ww_{pred}(t+1) \leq ww_{th}$) and the overload is predicted to exceed a given upper threshold ($ol_{pred}(t+1) > ol_{th}$).

Underload Condition	Overload Condition	Action
$ww_{pred}(t+1) \leq ww_{th}$	$ol_{pred}(t+1) \leq ol_{th}$	none
$ww_{pred}(t+1) \leq ww_{th}$	$ol_{pred}(t+1) > ol_{th}$	increase frequency
$ww_{pred}(t+1) > ww_{th}$	$ol_{pred}(t+1) \leq ol_{th}$	decrease frequency
$ww_{pred}(t+1) > ww_{th}$	$ol_{pred}(t+1) > ol_{th}$	decrease frequency

Table 5.1 Policy for Changing the Frequency

2. Decrease the frequency of all the islands by one step, when the number of waiting workers is predicted to exceed the given lower threshold ($ww_{pred}(t+1) > ww_{th}$).

3. Otherwise do not change the frequency.

Table 5.1 summarised all possible situations and the action implied by the above rules. Of course, the frequency is only varied within the limits given in Figure 2.16.

If some messages are half-way processed in the system, the effect of the change is not seen immediately in its full extent. We therefore limit the rate of frequency adjustments. The maximal allowable rate of frequency adjustments depends on the application-specific maximal processing latency l_{max} and on the smoothing factor α_{ol}.

5.2.4 Frequency Adjustment

As pointed out in [7], the operating speed of a processing unit on a multi-core platform is approximately proportional to its operating clock frequency:

$$speed \propto f, \tag{5.7}$$

whereas its dynamic power consumption is approximately proportional to the product of its operating frequency by the square of its operating voltage:

$$P_{dynamic} \propto f \cdot V^2. \tag{5.8}$$

Since a lower operating voltage increases the circuit delay, the operating voltage always imposes an upper bound on the operating frequency, and it follows that the dynamic power consumption of a processing unit is approximately proportional to the cube of its operating voltage:

$$P_{dynamic} \propto V^3, \tag{5.9}$$

or, equivalent, to the cube of its operating speed or frequency:

$$P_{dynamic} \propto speed^3 \propto f^3. \tag{5.10}$$

We use the full range of available frequency settings, that is from 100 MHz to 800 MHz. We change the frequency by adjusting the operating frequency divider $f_{div_{cur}}$:

$$\begin{aligned} f_{div_{inc}} &= f_{div_{cur}} - 1 \\ f_{div_{dec}} &= f_{div_{cur}} + 1 \end{aligned} \tag{5.11}$$

As pointed out above, there is a minimal required voltage level for each frequency, which we also have to adjust in order to avoid chip failures (cf. Figure 2.16).

5.3 Islands or Chip Level DVFS

The cores on the SCC are arranged in islands. This arrangement allows fine grained control over voltage (at granularity of 8 cores) and frequency (at granularity of 2 cores) at the granularity of island level. We do not utilise this feature of the SCC, instead we perform DVFS at chip level. There are multiple reasons behind this decision.

In order to calculate power consumption, we need both, supply voltage(V) and current(A). While SCC is equipped with many ***A**nalogue to **D**igital **C**onverters* (ADCs) that can measure the supply voltages for individual islands, it only provides current at chip level, which means power can only be measured for all the cores and the mesh together as a whole.

In addition, Our main aim has been to develop a light-weight power management approach. If we would had opted to use island level DVFS, it would require a very complex DVFS policy which might not be light-weight. As we know, while frequency can be set for two cores, the voltage can only be set at island level, i.e. for 8 cores. Now in case of an island level DVFS, the scheduler has to keep an account of which core has how much load. This could be a problem, for example, if one of the core in the frequency island (two cores on a tile form one frequency island), has more work than other, we have to keep the frequency level such that it would not have negative impact on the core with high workload. This gets more difficult, as on the SCC we can only set voltage for a minimum of 8 cores. We already know that, for a chosen frequency there is a minimum voltage required to ensure the circuit can run without any problem, setting the voltage higher means waste of power. Now we have to make sure that workload on the cores on an island is nearly identical so that we can have voltage/frequency that caters for the needs of all the cores on the island.

What this means is that if we want to utilise the island level DVFS feature of the SCC we have to modify the scheduler in order to keep track of which task is assigned to which core/island, in addition to overall load fluctuation due to the nature of the RSPs. Furthermore, with the centralised scheduler, it's highly likely that by the time we can get all the information required to calculate workload on the cores in an island and required changes in voltage/frequency value, the task assigned to core may change and the load associate with it as well, rendering the previous calculation useless.

One interesting observation here is that, if the load is nearly identical for all of the cores on an island we can set voltage/frequency which is fit for all the cores. Now if all the cores on the chip have nearly identical workload it would make perfect sense to have system level DVFS. The centralised scheduler that we use provides exactly that, a near balance of the workload amongst all the workers.

As we already argued in Section 4.2 with a small example, that with a load balanced system almost all the cores will have uniform amount of work to do and as such we can reduce the frequency for all the cores by the same amount. This effectively opens a way that allows light-weight DVFS mechanism such as ours, that reduces energy consumption of RSPs without affecting performance.

5.4 Chapter Summary

In this chapter we described a resource-aware execution layer called RA-LPEL for the RSPs. We also covered the heuristics used in experimental evaluation to adjust power consumption at run time. Our heuristics were based on system state such as overload and underload. We used the correlation of the input rate and the output rate to detect overload situations in the system. In addition, we used monitoring information about idling cores to detect underload situations. Based on these two metrics, we derived our DVFS policies that govern the decision of whether to change voltage and frequency of the SCC or not.

Chapter 6

Experimental Evaluation

We ran two sets of experiments. First set is to evaluate the efficiency of the ***R**esource-**A**ware **L**ight-weight **P**arallel **E**xecution **L**ayer* (RA-LPEL) on the ***S**ingle-chip **C**loud **C**omputer* (SCC) in terms of throughput, latency and scalability. The second set is to evaluate the efficiency in terms of impact of ***D**ynamic **V**oltage and **F**requency **S**caling* (DVFS) and resulting reduction in power consumption.

We describe our choices of use cases for the experiment in § 6.1. Then, we describe the experimental setup in § 6.2. The results for throughput, latency and scalability are discussed in § 6.3. Afterwards, in § 6.4 we examine the impact of various energy policies of RA-LPEL on power consumption. Finally, in § 6.5 we summarise the topics discussed in this chapter.

6.1 Use Cases

We describe our use cases that we use experimentally to evaluate RA-LPEL on the SCC. These use cases are implemented as stream programs using the language S-Net. Here we provide the functional description and concrete implementation of the use cases in the S-Net.

Each use case contains main structure which performs the computation. Normally this main structure is comprised of multiple sub-tasks. To increase the level of concurrency, S-Net provides parallel replication to create multiple instances of the main structure. In all the described use cases, parallel replication is shown as **!!<node>**, where **!!** is the parallel replication operator of S-Net and tag **<node>** decides number of instances created. Note that each use case is set-up with an appropriate concurrency level. This ensures that we avoid the case where workers are idle because there is no available task to execute, while at the same time, the number of tasks is not too large to cause overhead.

6.1.1 Fast Fourier Transform (FFT)

```
net fft
{
  box initialiser((X, <node>) → (X, Y, W, <step>));
  box round((X, Y, W, <step>) → (X, Y, W, <step>));
} connect
       ( initaliser .. round .. round )!!<node>;
```

Listing 6.1 S-Net Implementation of the FFT Algorithm

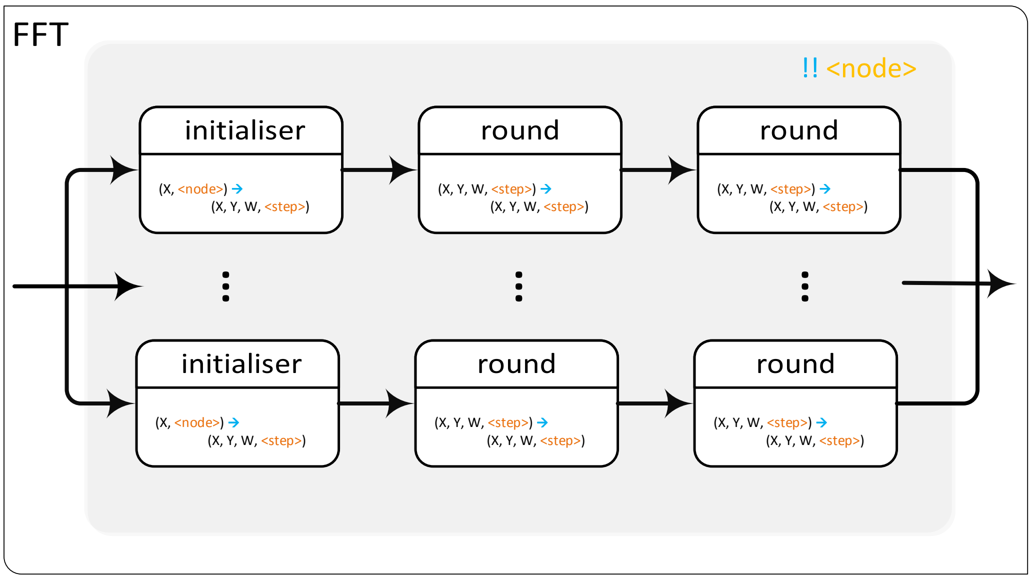

FFT
!! <node>
initialiser
(X, <node>) → (X, Y, W, <step>)
round
(X, Y, W, <step>) → (X, Y, W, <step>)
round
(X, Y, W, <step>) → (X, Y, W, <step>)
initialiser
(X, <node>) → (X, Y, W, <step>)
round
(X, Y, W, <step>) → (X, Y, W, <step>)
round
(X, Y, W, <step>) → (X, Y, W, <step>)

Figure 6.1 A Schematic Representation of the FFT as an S-Net Network

Fast Fourier Transform (FFT) [31] is an algorithm to compute the ***Discrete Fourier Transform*** (DFT) of a sequence. Some example applications of FFT include discrete sine or cosine transforms (used for MP3/MPEG/JPEG encoding), solving partial differential equations, and digital signal processing in general.

Listing 6.1 shows S-Net implementation of FFT application, also denoted as **FFT**. S-Net network of FFT is depicted in Figure 6.1. The application contains a main structure which performs the FFT algorithm. The main structure includes the `initialiser` box which allocates memory to store temporary values during the transformation (Y), generates the series of sine and cosine waves (W) and decides the step size (<step>) for each following FFT round. The `round` box applies step stages, each of which calculates the N/2 frequency spectra from N frequency spectra from the previous stage. The `round` box is applied in pipeline manner until all the stages have been calculated.

To increase the level of concurrency S-Net parallel replication mechanism is employed to create multiple instances of the main structure. The number of instances is defined by tag <`node`>.

6.1.2 Data Encryption Standard (DES)

```
net des
{
  box initP((plainText, key, <node>) → (text, key, <round>));
  box roundP((text, key, <round>) → (text, key, <round>));
  box finalP((text, key, <round>) → (cipherText));
} connect
       ( initP .. roundP .. roundP .. roundP .. roundP ..
         roundP .. roundP .. roundP .. roundP .. roundP ..
         roundP .. roundP .. roundP .. roundP .. roundP ..
         roundP .. roundP .. finalP
       )!!<node>;
```

Listing 6.2 S-Net Implementation of the DES Algorithm

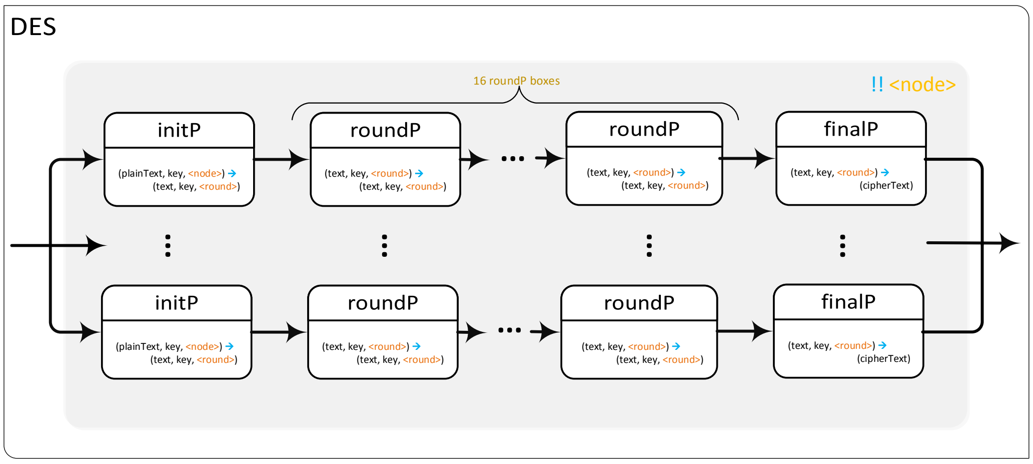

DES
16 roundP boxes
!! <node>
initP
(plainText, key, <node>) → (text, key, <round>)
roundP
(text, key, <round>) → (text, key, <round>)
roundP
(text, key, <round>) → (text, key, <round>)
finalP
(text, key, <round>) → (cipherText)
initP
(plainText, key, <node>) → (text, key, <round>)
roundP
(text, key, <round>) → (text, key, <round>)
roundP
(text, key, <round>) → (text, key, <round>)
finalP
(text, key, <round>) → (cipherText)

Figure 6.2 A Schematic Representation of the DES as an S-Net Network

***D**ata **E**ncryption **S**tandard* (DES) use case is a block cipher application and denoted as **DES** [105]. Block cipher means it takes in plaintext blocks of a given size and returns encrypted/ciphertext blocks of the same size. DES operates on a sequence of messages, each message contains a number of plaintext blocks and their corresponding keys. Listing 6.1 shows S-Net implementation of this use case, while Figure 6.2 is a graphical representation of the S-Net network of DES. The main structure which performs the DES encryption is made up of three S-Net boxes. The box `initP` applies an initial permutation to the given plaintext and then the permuted block is divided into two blocks of equal size. The box `roundP` applies the cypher function to those two blocks once. As shown in Listing 6.2, to apply 16 rounds of ciphering there are 16 instances of the box `roundP` connected in a pipeline manner. The box `finalP` applies the final permutations after joining up the two blocks into one.

6.1.3 Colour Histogram Calculation

In the field of computer vision and photography, a colour histogram is a representation of the distribution of colours in an image. Image histograms have wide use in thresholding, mostly used in computer vision. Edge detection, image segmentation, and co-occurrence matrices are some of the applications where threshold values obtained from histograms can be used.

This use case takes in digital images as its input, and RGB colour histograms are produced as the output. The main structure of this application contains two components: a `splitter` box that separates RGB channels of the a given image, and a `calcHist` box that calculates the colour histogram of the given image. The images produced by the `splitter` box are assigned with different tag values (<colour>) so that they are passed to different instances of the `calcHist` box. The S-Net implementation of this use case is shown in Listing 6.3, where Figure 6.3 is a graphical model of the network. The execution time of each box is dynamic, as it depends on the size of input image.

```
net histogram
{
  box splitter((image, <node>) → (chnlImage, <colour>));
  box calcHist((chnlImage, <colour>) → (histImage, <colour>));
} connect
        ( splitter .. (calcHist)!!<colour> )!!<node>;
```

Listing 6.3 S-Net Implementation of the Histogram Application

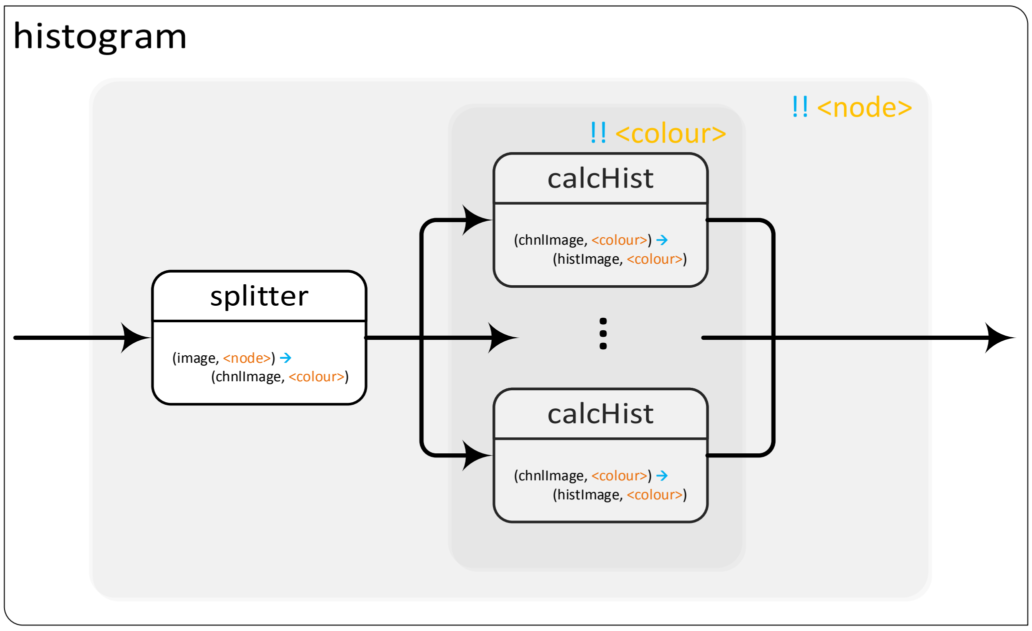


Figure 6.3 A Schematic Representation of the Histogram Application as an S-Net Network

6.1.4 Image Filter

```
net filter
{
  box splitter((image, <size>, <node>) → (subImage, <tag>));
  box filters((subImage, <tag>) → (subImage, <tag>));
  box merger((subImage, <tag>) → (filtImage));
} connect
       ( splitter .. (filters)!!<tag> .. merger )!!<node>
```

Listing 6.4 S-Net Implementation of the Image Filter Application

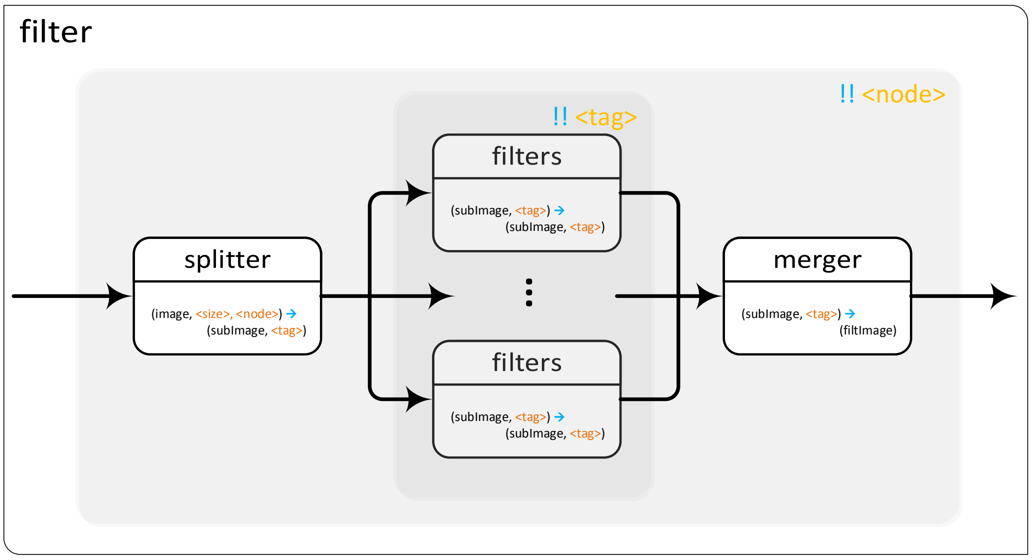


Figure 6.4 A Schematic Representation of the Image Filter Application as an S-Net Network

Image filtering is used to reduce noise and/or extract useful image structures in most applications in computer vision and computer graphics field.

As shown in S-Net implementation of image filter in Listing 6.4 the main structure of this application contains a `splitter` box, `filters` box and `merger` box. The `splitter` box takes an image as an input and splits it into multiple sub-images. The size of input image and requested sub-image size (<size>) has direct influence on how many sub-images are generated. Each sub-image is assigned a different tag value (<tag>). This tag value is used to decide which instance of `filters` box will receive the sub-image in question. The box `filters` apply a series of filters on the sub-image. The box `merger` takes in filtered sub-images and assembles them into a single filtered image. The execution time of each box depends on the given input image size and number of `filters` box instances created. The S-Net network of image filter is visualised in Figure 6.4.

6.1.5 Face Detection

```
net faceDetection
{
  box classifier1((image, <node>) → (image, <res>));
  box classifier2((image, <res>) → (image, <res>));
  box classifier3((image, <res>) → (image, <res>));
} connect
  ( classifier1 .. classifier2 .. classifier3 )!!<node>;
```

Listing 6.5 S-Net Implementation of the Face Detection Application

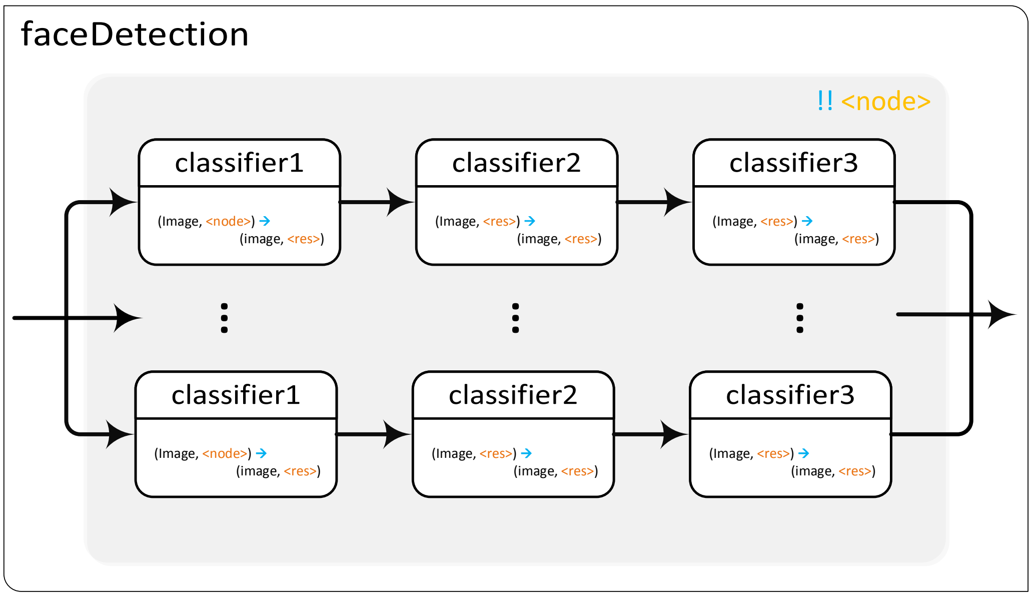


Figure 6.5 A Schematic Representation of the Face Detection Application as an S-Net Network

Face detection (more general case is feature detection) is used widely in computer vision field. Examples of the applications include but are not limited to biometric identification and authentication, surveillance marketing and photography.

The face detection application applies a sequence of classifiers to detect different features to be found in an image, like eyes, mouth, nose, etc. The source code and diagrammatic representation of the S-Net network of face detection is shown in Listing 6.5 and Figure 6.5 respectively. The main structure of face detection contains different classifiers arranged in pipeline manner. Each classifier operates on the same image but looks for different features. For simplicity of the implementation we used a simulation of the actual feature detectors and decided randomly whether the feature predictors were matched, i.e. if a feature is detected or not. Based on the value of tag <`node`>, multiple instances of the main structure are created to increase the level of concurrency.

6.2 Experimental Set Up

In our experiments we used a default sccKit 1.4.2 configuration, with memory and mesh running at 800 MHz. For performance experiments in Section 6.3 cores are running at fixed frequency of 533 MHz, while for power experiments in Section 6.4 cores run at various frequencies during execution of a benchmark. We used the SCCLinux device *NCMDeviceFD* for the shared memory mapping.

We used 4 out of the 48 cores of the SCC as donors for the shared memory, and 4 further cores to model an external source/producer and sink/consumer for ***R**eactive **S**tream **P**rograms* (RSPs). This left us with at most 40 cores as workers that process messages.

6.2.1 Data Collection and Post-Processing

As mentioned in Section 2.5 the SCC is connected to ***F**ield **P**rogrammable **G**ate **A**rray* (FPGA) called the ***B**oard **M**anagement **C**ontroller* (BMC). While SCC is equipped with many ***A**nalogue to **D**igital **C**onverter* (ADC) sensors which can measure the supply voltages for individual islands, however it does not provide current at this level. In order to calculate power, we need both supply voltage and the current. Both of these values are available at the chip level on the 3.3 V rails, which means power can only be measured for all cores and the mesh together as a whole.

The ***P**ower **M**easurement **C**ontroller* (PMC) situated in the FPGA/BMC periodically collects the data from the measurement ADCs and stores them in the power measurement registers. These registers can be memory mapped and read by the cores or the ***M**anagement **C**onsole **PC*** (MCPC). Furthermore, FPGA also provides a global timestamp counter that can be used across cores to have a reliable/consistent time source. Since the global timestamp counter is located on FPGA it does not get affected by frequency change on the SCC cores.

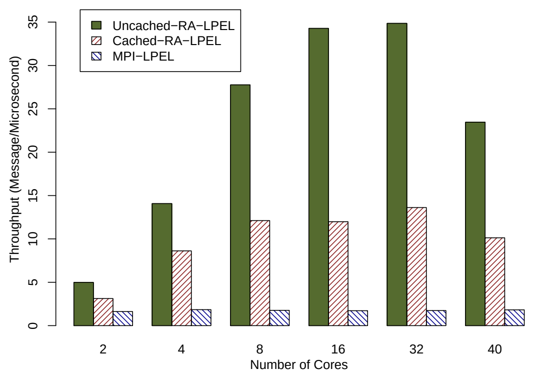
As we mentioned before, we only need to observer input rate, output rate and waiting workers. In addition, we also keep track of voltage and current of the SCC chip. We achieve this by memory mapping registers mentioned above. All of this information is collected and used by the power manager (a part of conductor) at run-time. The power manager also writes all the information in to a log file for post processing. Post processing is only required to generate graphs and analyse the effects of different energy policies. Power consumption is calculated by multiplying the voltage and current consumption at any given moment (i.e. when each measurement is taken). Average power consumption of a benchmark execution is the arithmetic mean of all the power readings.

6.3 Performance Experiments

In this section we evaluate the efficiency of RA-LPEL with dynamic load balancing on the SCC and compare it to distributed S-Net with decentralised version of ***L**ight-weight **P**arallel **E**xecution **L**ayer* (LPEL) [112] with manual load balancing as an execution layer. In the latter, a global mapper is used to allocate tasks to cores and a local scheduler for each core. The core-local scheduling policy is round-robin, whereas the global mapper uses either a round-robin policy or a static mapping. The ***M**essage **P**assing **I**nterface* (MPI) is used to facilitate communication between cores. We also evaluate the scalability of RA-LPEL for varying numbers of cores.

The SCC does not provide cache coherency and offers no direct control over cache flushing, so we have to ensure consistency when using the cache. We use two variants of RA-LPEL: In Cached-RA-LPEL only the task stack consisting of non-shared data is cached, whereas in Uncached-RA-LPEL we do not use caching. For LPEL with manual load balancing MPI is used and memory is not shared, so we can make full use of caching. In this approach, which we denote MPI-LPEL, each benchmark is mapped to achieve the best load balance, i.e. each instance of the pipeline is mapped on a separate core. The first core is special: Besides processing messages, it is also responsible for receiving input messages from the environment, distributing messages to the other cores and collecting them, and sending them out to the environment. The MPI communications occur only between the first and all other cores. To ensure the message order, MPI must be used in blocking mode.

Figure 6.6 shows the maximum throughput and minimum latency of the FFT benchmark. Uncached-RA-LPEL outperforms Cached-RA-LPEL by a factor of at least 1.5 for both, throughput and latency, even though caching is disabled in Uncached-RA-LPEL. Since the SCC is configured as a shared memory platform, the caches need to be flushed to ensure data integrity among cores. This causes a significant overhead that caching cannot compensate.



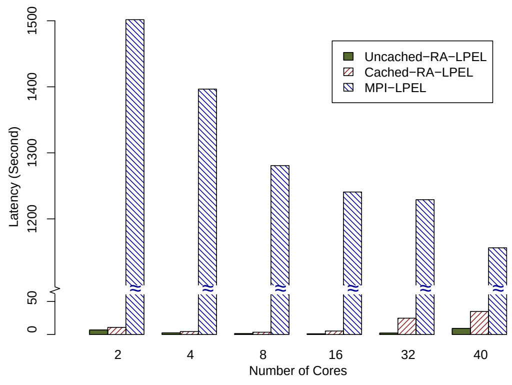


Figure 6.6 Performance of FFT on Uncached-RA-LPEL, Cached-RA-LPEL and MPI-LPEL

MPI-LPEL has the lowest throughput, because the communication performance of MPI is inferior to direct memory access. The maximum communication bandwidth between 2 cores is around 2.78 MiB/s for MPI. Transferring 64 kB between 2 cores takes more than 22 ms via MPI but only 15 ms via direct memory access. With 2 cores the throughput achieved by MPI-LPEL is less than for Cached-RA-LPEL and Uncached-RA-LPEL, and for more cores the MPI bandwidth is shared. MPI-LPEL requires one core to communicate with all other cores, sending input messages and receiving output messages. Due to similar load on the cores, this communication is likely to coincide. MPI introduces a (de)serialising and (un)packing overhead and operates in blocking mode and this forces each core to wait while sending messages via the MPI interface. As a result the throughput for MPI-LPEL can be seven times smaller than for Cached-RA-LPEL and 20 times smaller than for Uncached-RA-LPEL, as shown in Figure 6.6.

MPI-LPEL has a higher latency than Uncached-RA-LPEL and Cached-RA-LPEL. Besides the before mentioned reasons, the RA-LPEL scheduler affords control over the consumption rate of input messages to optimise latency [104]. MPI-LPEL lacks this feature and allows the program to consume input messages even when it is overloaded and unable to process them. The latency for MPI-LPEL can be 370 and 900 times higher than for Cached-RA-LPEL and Uncached-RA-LPEL, respectively.

Figure 6.7 shows how Uncached-RA-LPEL scales for the FFT benchmark. From 2 to 16 cores the throughput scales roughly linearly, but more cores imply more memory accesses. Memory is managed by 4 memory controllers and extensive access can cause contention. Therefore throughput does not scale well between 32 and 40 cores. Although FFT operates on a sizeable amount of data (64 kB), the computation time is relatively small. On average each task takes 65 ms to process a message, so each core must access a large amount of data frequently.

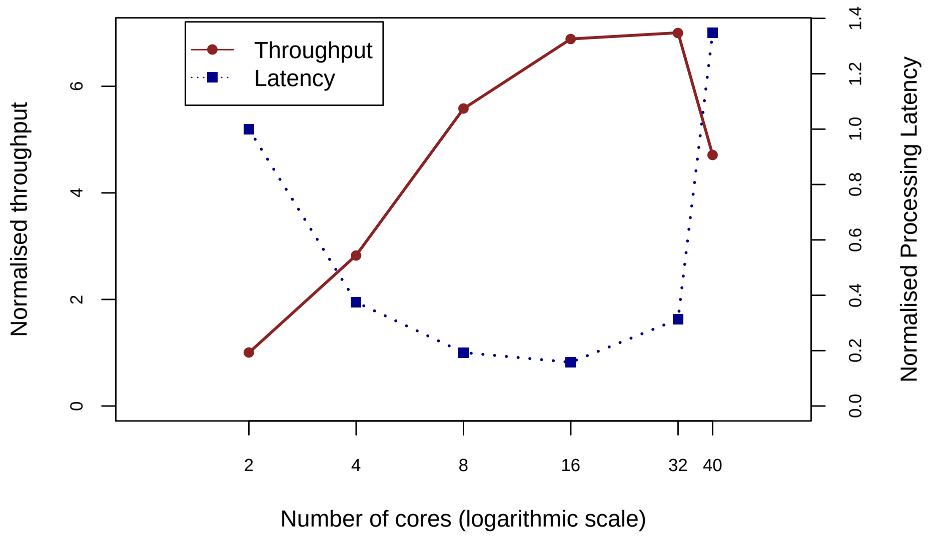


Figure 6.7 Scalability of FFT on Uncached-RA-LPEL

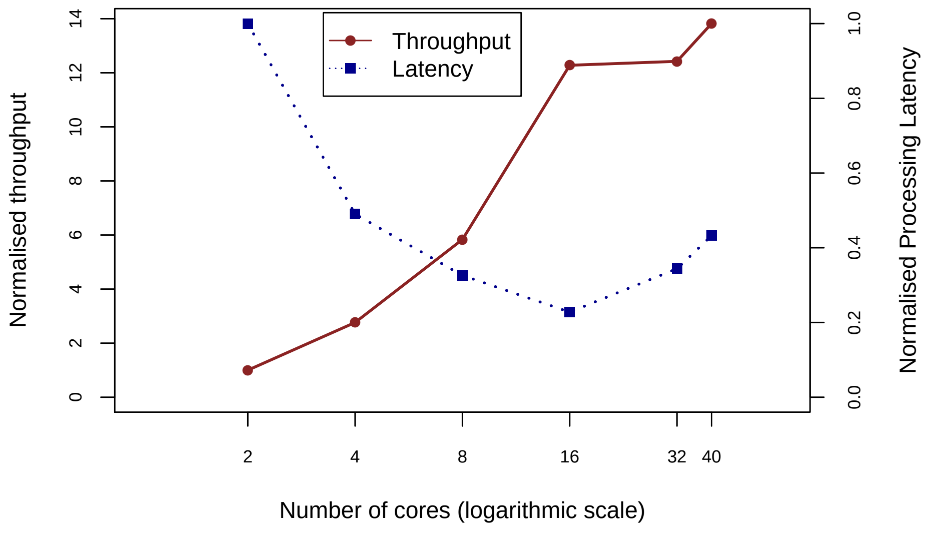


Figure 6.8 Scalability of DES on Uncached-RA-LPEL

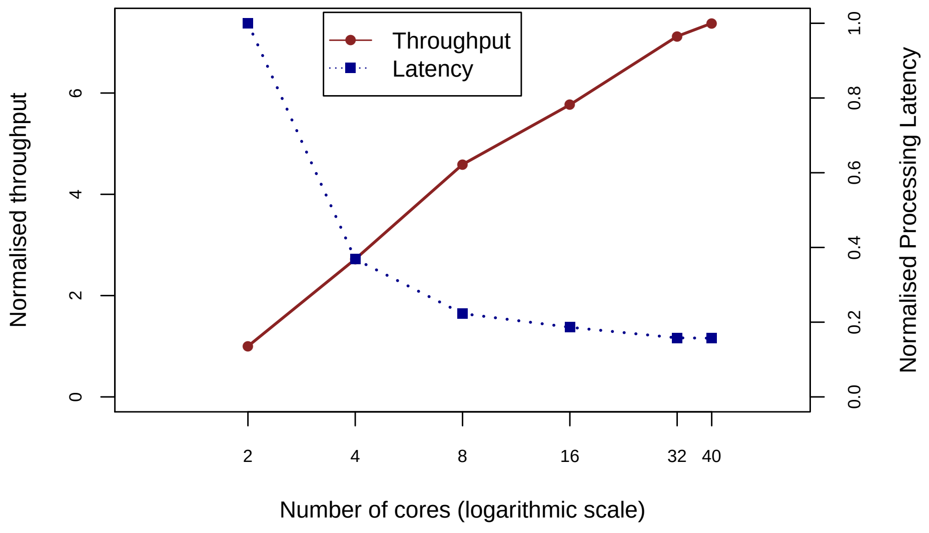


Figure 6.9 Scalability of Histogram on Uncached-RA-LPEL

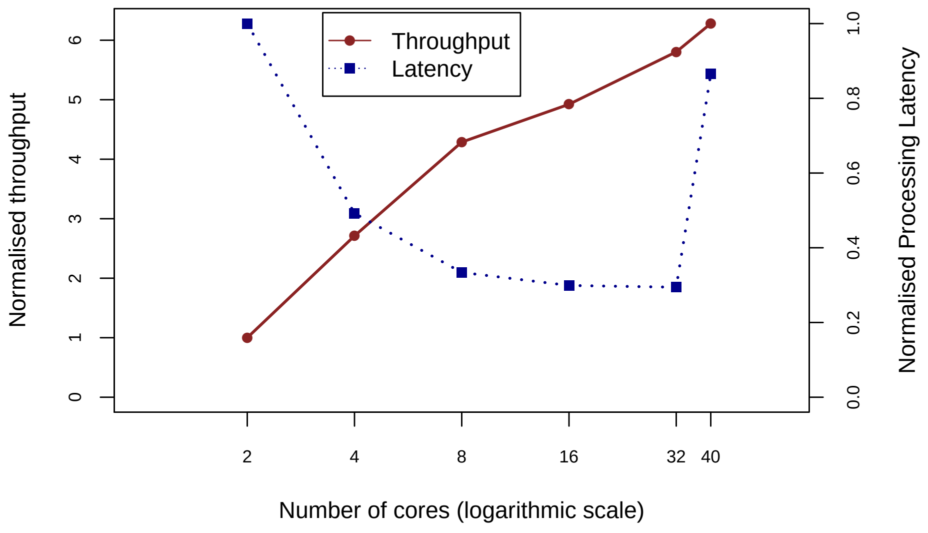


Figure 6.10 Scalability of Image Filter on Uncached-RA-LPEL

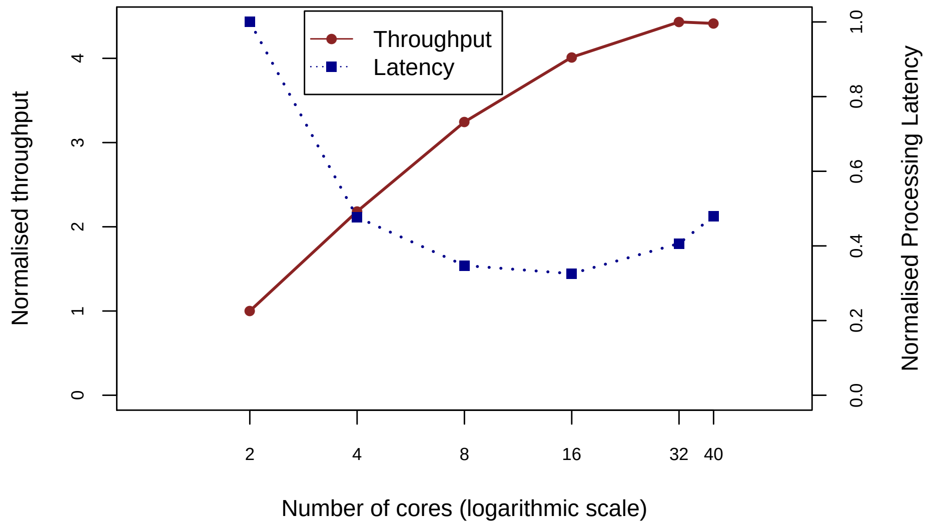


Figure 6.11 Scalability of Face Detection on Uncached-RA-LPEL

In contrast, DES requires extensive computation on a small amount of data. Each input message is 2 kB and each task takes 194 ms on average to process a message. For this reason, the throughput of DES scales better, as shown in Figure 6.8.

The latency depends on the immanent concurrency level of the stream program. Increasing the number of cores takes advantage of the concurrency within the stream program and helps to reduce the latency. However, more cores also imply higher communication costs, as tasks are spread among cores. Figure 6.7 and Figure 6.8 show that the latency decreases when we increase the number of cores up to 16. For 32 and 40 cores the communication overhead surpasses the benefit of concurrency. The latency of DES and FFT therefore does not scale well for 32 or 40 cores.

Figure 6.9 and Figure 6.10 show throughput and latency for histogram and image filter benchmark respectively. In contrast to DES and FFT, here we can see roughly linear scaling in throughput from 2 cores all the way to 40 cores. This was expected, as histogram and image filter are computationally more intensive than DES and FFT. For histogram the latency continues to decrease up to 40 cores. In contrast, a decrease in latency can be observed for image filter benchmark for up to 32 cores, after which it rises sharply. One reason can be the higher number of message queuing at the merge point in the stream network, which can be a bottleneck.

As can be seen in Figure 6.11, face detection is computationally expensive, in line with histogram and image filter, and as such we see a similar trend in throughput scalability. For the latency part, we can see substantial drop until 8 core, then slight change for 16 cores. From 16 cores onward, we can see a slight increase in the latency.

Benchmark	Task	Min (s)	Max (s)	Diff (%)
FFT	initP	1.1232	1.9954	77.65
	stepP	10.3226	15.2775	48.00
DES	initP	3.6370	6.8986	89.68
	subRound	4.7194	14.3530	204.13
	finalP	2.1357	4.0790	90.99
Histogram	<collect>	0.9477	1.5814	66.86
	<split>	0.8987	4.2756	375.74
	split	3.9660	5.0427	27.15
	calHist	22.3738	28.7144	28.34
Image Filter	<collect>	0.6123	27.0231	4313.34
	<filter>	0.1787	0.4919	175.19
	<parallel>	0.4500	1.5947	254.38
	<split>	0.3979	11.5076	2792.24
	filt	134.2071	470.1736	250.33
	split	1.0512	6.3557	504.62
Face Detection	classifier1	9.2898	15.7955	70.03
	classifier2	10.9571	15.3507	40.10
	classifier3	6.8608	11.9101	73.60

Table 6.1 Minimal and Maximal Task Execution Time for Benchmarks Running on 40 Cores

Table 6.1 shows the minimal and maximal execution time for each task in the benchmarks. Some of these tasks have multiple instances occurring in the separate parallel pipelines created by S-Net. We can see that all benchmarks show a considerable variation in execution times of tasks. This can be attributed to high work-load imbalance which depends highly on input messages. These numbers underline the need for a load balancing scheduler like the one we have presented.

The table shows that the <collect> task of the image filter benchmark has nearly 4000% variation on 40 cores (for 32 core run this variation is 495.30%). The <collect> task merges messages from multiple streams and forwards them to the subsequent component. Such a high variation indicates that at some point multiple messages were waiting to be merged, resulting in the sharp increase in latency seen in Figure 6.10.

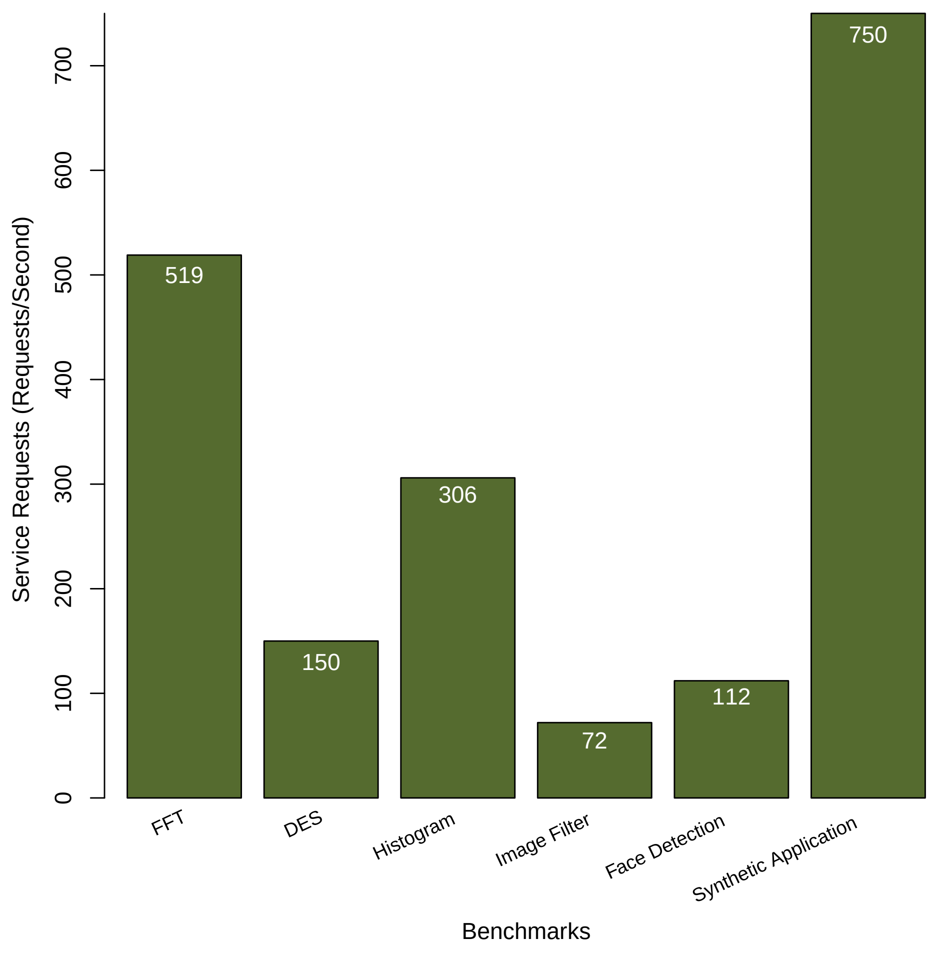


Figure 6.12 Average Service Requests Received per Second by the Conductor

6.3.1 Central Work Queue Bottleneck at the Conductor

In the centralised scheduler of RA-LPEL, the tasks which can be executed next, i.e. ready tasks, are stored in a queue called ***C**entral **T**ask **Q**ueue* (CTQ). This CTQ is maintained by the conductor. Since we only have one conductor which serves 40 workers, the natural question would be:

is having only one conductor for 40 workers causing a bottleneck?

We run an extra experiment in order to investigate if this is the case. For this experiment we compare service request received by the conductor with 40 cores executing as workers.

Figure 6.12 shows request received by the conductor with all the benchmark application. As we can see with 72 requests per second, the image filter sends least amount of requests. While with 519 requests per second, FFT sends the highest amount of requests per second. In order to see if 519 is causing a bottleneck at the conductor, we use a synthetic S-Net application, where tasks do not perform any work. The benchmark contains tasks that just read an integer value from its input stream and write this value back to its output stream. With this synthetic benchmark the conductor is capable to service 750 requests per second, which is nearly 50% higher than requests sent by the FFT benchmark. This clearly shows that our experiments did not suffer from bottleneck at the conductor.

Furthermore, the number of requests that the conductor receives is also dependent on the applications as well. For example, the number of requests will be lower for applications where tasks take considerable amount of time (e.g. couple of seconds) for completion, in comparison to the applications where task completion time is very small (e.g. couple of nanoseconds).

Although, bottleneck at the conductor still is of a concern to us, as we move to an architecture with a higher core count, the conductor may/will become a bottleneck. Still in our opinion it is not a very big problem. There are many architectures which come with far higher core counts than SCC, but the cores are arranged in an island fashion in similarity to the SCC. For example, Kalray ***M**assively **P**arallel **P**rocessor **A**rray* (MPPA)2-256 Boston [66] processor architecture has 256 cores. These 256 cores are arranged in 16 clusters, with each cluster containing 16 cores. In order to avoid the conductor becoming a bottleneck, we can run multiple instances of RA-LPEL, ideally one instance per cluster. In this case, one instance of the conductor is only responsible for 15 workers[1]. As we can see from Figure 6.12, it is highly unlikely that one conductor will become a bottleneck when there are only 15 workers.

6.4 Power Optimisation Experiments

In this section we evaluate the efficiency of RA-LPEL in terms of energy usage by deploying different energy policies. For all the experiments described in this section we use all 40 cores of the SCC (We lose 4 cores as donors for the shared memory, and 4 further cores to model an external source/producer and sink/consumer for RSPs).

For these experiments we use a fixed input pattern consisting of 7000 messages for FFT and face detection, 2300 messages for DES and histogram, and 1800 messages for image filter. In order to see the effect of DVFS on energy consumption we deploy three different energy policies. The case when dynamic voltage and frequency scaling is enabled is denoted as DVFS. The case when voltage is fixed at maximum (1.1V) and only frequency is scaled is denoted as ***D**ynamic **F**requency **S**caling* (DFS). The last

[1]This would require a modification of the RA-LPEL, in terms of new communication protocol to allow conductor-to-conductor communication.

policy which does not scale either volt or frequency (maximal voltage and frequency with volt fixed at 1.1 V and frequency fixed at 800 MHz) is denoted as ***M**aximal **V**oltage and **F**requency* (MVF).

We also test our DVFS energy policy using different thresholds for overload (ol_{th}) and waiting workers (ww_{th}). We expected different thresholds to yield different behaviour with respect to energy efficiency, which would allow us to pick thresholds with a balanced behaviour, i.e. which would save a significant amount of energy without sacrificing too much computational performance.

6.4.1 Effectiveness of Power Saving

Table 6.2 summarises the main result of our experiments, indicating the total wall-clock time, the average power level, and the total energy consumption of each benchmark under three different energy policies: DVFS, DFS and MVF.

The total wall-clock time that each benchmark takes to run is roughly the same ($\sigma/\mu < 0.6\%$) under DVFS, DFS, and MVF, the reason being that we are considering a reactive scenario, where the pace of the system is not determined by the core frequency, but by the data input rate imposed by the environment. In other words, the system must complete a given workload within a specified wall-clock time. It cannot complete the workload significantly faster, because the input data only becomes available in real time, and it must not complete its workload significantly slower (the latter would indicate an abnormal overload situation).

Tiny differences in the wall-clock times are mainly due to variations in the dynamic scheduling of tasks: For example, even a slight difference in the execution time of a sole task can influence numerous subsequent scheduling decisions, like the assignment of individual tasks to particular cores, which can in turn cause changes in the memory access times of these tasks, due to the use of differing ***N**etwork **on** **C**hip* (NoC) routes.

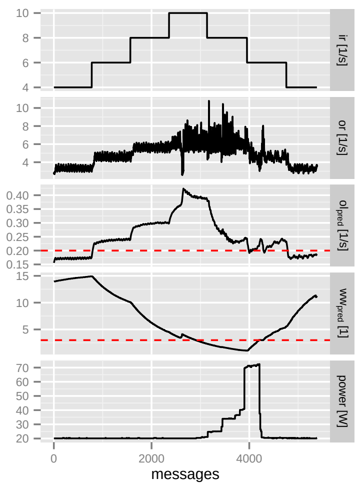
Our results indicate that the DVFS strategy cuts the energy consumption by, 58% for FFT, 35% for DES, 52% for histogram, 70% for image filter, and 42% for face detection benchmark. As expected, the exclusive use of frequency scaling in DFS without voltage scaling saves significantly less energy in all benchmarks.

6.4.2 Influence of Thresholds

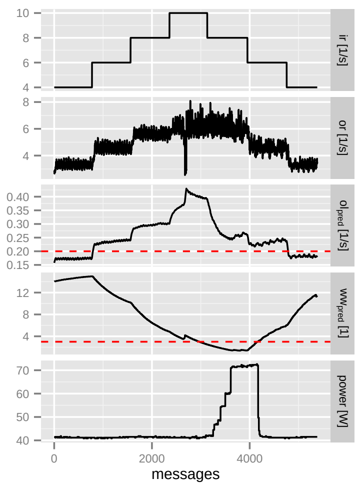
Figure 6.13, Figure 6.14, Figure 6.15, Figure 6.16, and Figure 6.17 show time series of our experiments under the three energy policies for each benchmark. Each sub-figure shows the progress of the input rate ir, the output rate or, the predicted overload ol_{pred}, the predicted number of waiting workers ww_{pred}, and power.

Benchmark	Policy	Wall-Clock Time [s]	Power [W]	Energy [kJ]
FFT	DVFS	1646.045	30.600	50.378
	DFS	1662.049	47.934	79.670
	MVF	1655.352	72.579	120.143
	σ^2	43.069	296.665	818.136
	σ/μ	0.40%	34.19%	34.30%
DES	DVFS	1447.779	46.625	67.507
	DFS	1448.767	56.862	82.399
	MVF	1451.745	71.838	104.291
	σ^2	2.842	107.199	228.231
	σ/μ	0.12%	17.72%	17.83%
Histogram	DVFS	1637.046	34.189	55.969
	DFS	1638.545	51.399	84.200
	MVF	1624.340	72.736	118.148
	σ^2	40.609	248.587	646.176
	σ/μ	0.39%	29.88%	29.52%
Image Filter	DVFS	1559.915	21.420	33.417
	DFS	1550.796	41.515	64.386
	MVF	1562.298	71.447	111.621
	σ^2	24.569	422.480	1033.988
	σ/μ	0.32%	45.89%	46.06%
Face Detection	DVFS	1797.958	42.044	75.757
	DFS	1774.602	53.726	95.513
	MVF	1776.495	74.030	131.508
	σ^2	112.186	174.648	532.678
	σ/μ	0.59%	23.35%	22.87%

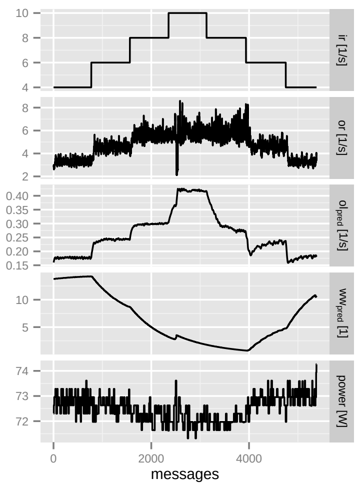
Table 6.2 Total Wall-clock Time, Average Power Level, and Total Energy Consumption of Each Benchmark Under Three Different Energy Policies, as Mean Over Five Runs



(a) DVFS

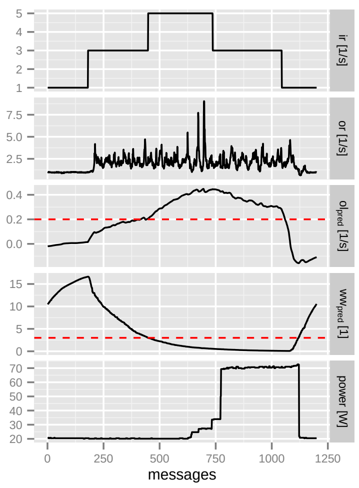


(b) DFS

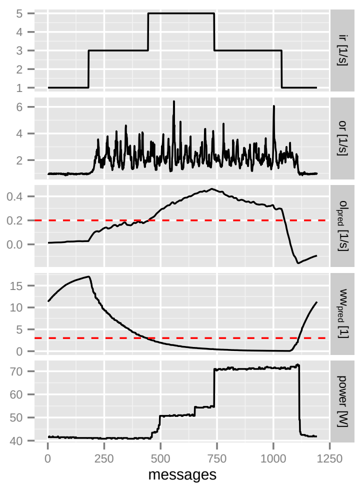


(c) MVF

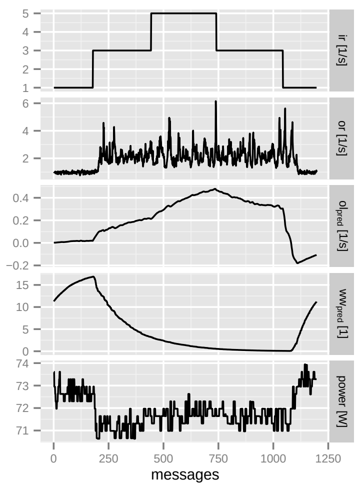
Figure 6.13 FFT with Thresholds $ol_{th} = 0.20$ and $ww_{th} = 3.0$



(a) DVFS



(b) DFS



(c) MVF

Figure 6.14 DES with Thresholds $ol_{th} = 0.20$ and $ww_{th} = 3.0$

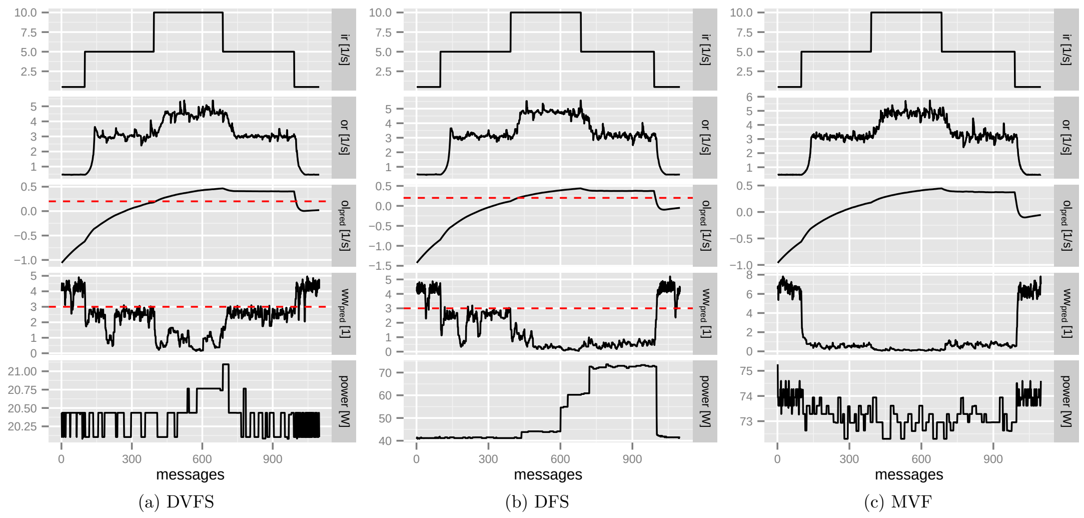


Figure 6.15 Histogram with Thresholds $ol_{th} = 0.20$ and $ww_{th} = 3.0$

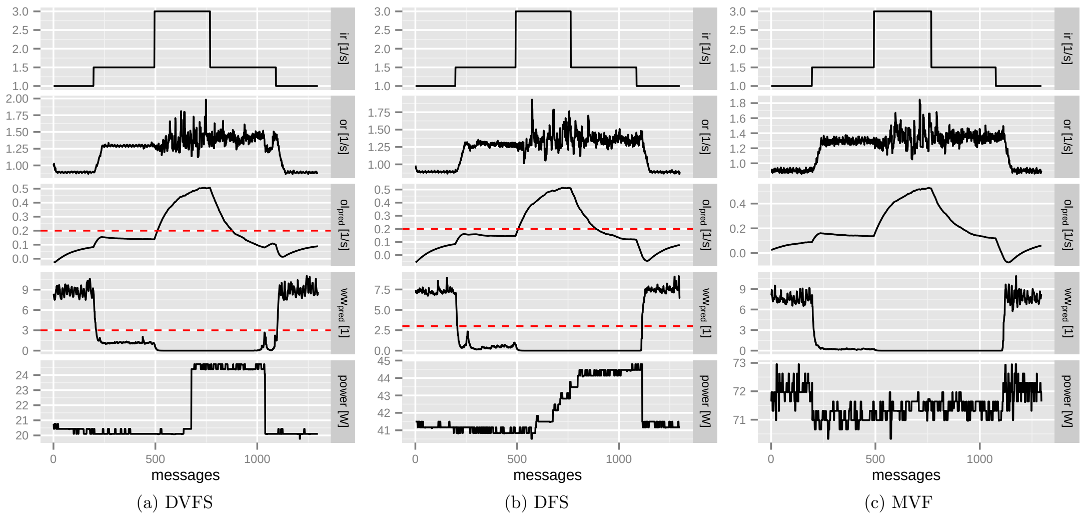
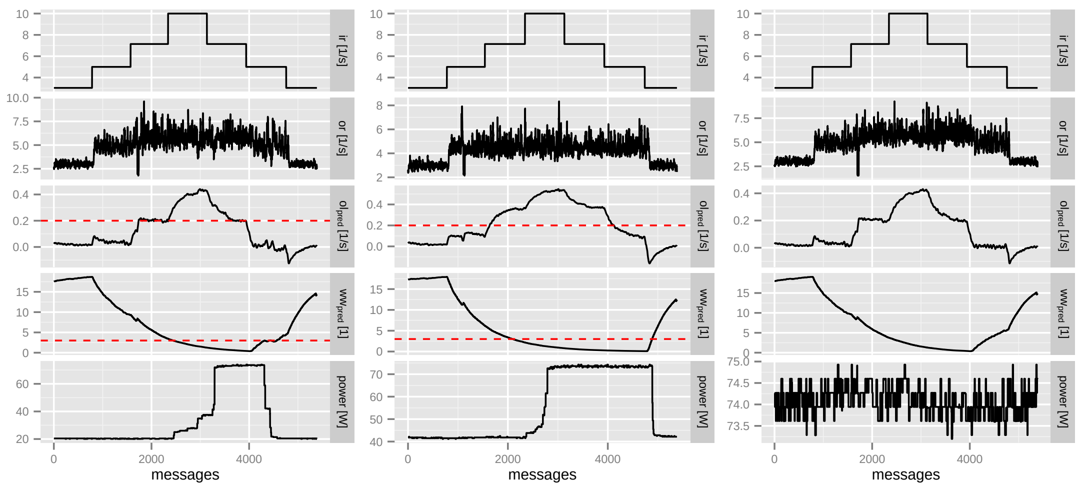


Figure 6.16 Image Filter with Thresholds $ol_{th} = 0.20$ and $ww_{th} = 3.0$



(a) DVFS (b) DFS (c) MVF

Figure 6.17 Face Detection with Thresholds $ol_{th} = 0.20$ and $ww_{th} = 3.0$

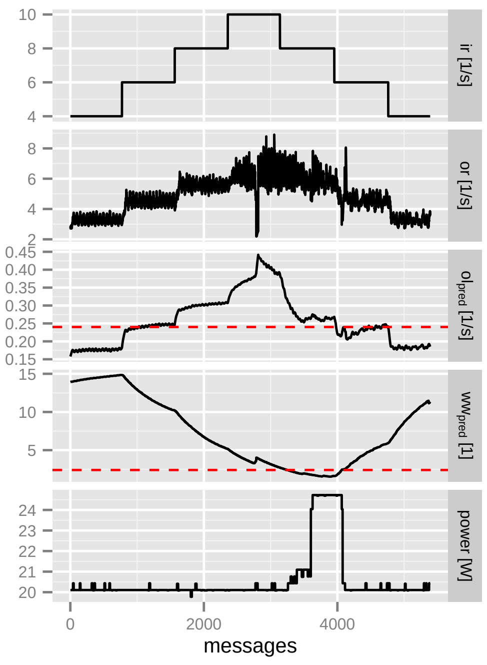
The red dashed lines mark our balanced overload threshold $ol_{th} = 0.2$ and our balanced waiting threshold $ww_{th} = 3.0$. In the topmost panel we can see our input rate pattern, in this case a synthetic ascending/descending step pattern. The panel below shows the corresponding output rate, which can be seen to carry a significant jitter. The next two panels show the predicted overload ol_{pred} and the predicted number of waiting workers ww_{pred}, respectively. In the last panel, we see that the power level for DVFS follows roughly the shape of the input rate pattern, although it is delayed. The pattern for DFS is similar, but the system operates at a higher power level, which is visible as a shift on the power axis. Moreover, the system reaches the maximal power level faster and stays there longer. For MVF, the power level is practically constant, and what can be seen in the panel is merely a slight jitter. Bakker et al. [10] have observed a similar fluctuation in power consumption of the SCC chip in idle mode. They ascribe this behaviour to the charging and discharging of a stabiliser capacitor in the voltage regulator circuits.

Table 6.3 summarises the results of our experimentation with different overload and waiting thresholds under the DVFS policy. For the same reason as given in Section 6.4.1 concerning Table 6.2, the total wall-clock time that each benchmark takes to run is roughly the same for all different chosen thresholds.

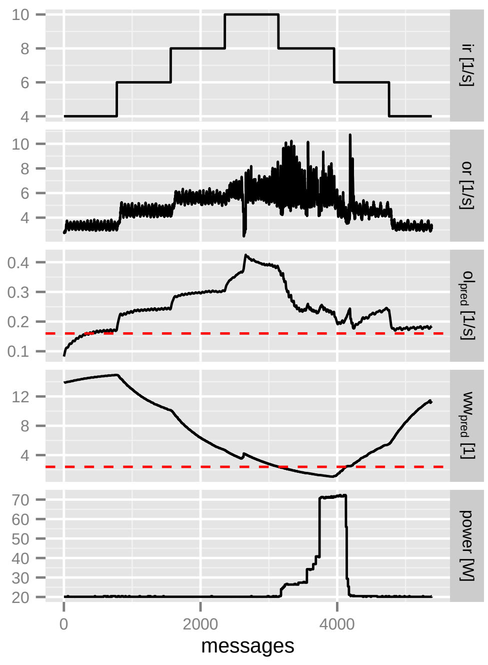
Let's have a closer look at the effect of the different parameter choices of the FFT benchmark. The combination of a high overload threshold of $ol_{th} = 0.24$ and a low waiting threshold of $ww_{th} = 2.4$, as seen in Figure 6.18(a), causes the system to increase its performance configuration at a very late point in time, and performance is decreased again shortly afterwards. Although the ol_{pred} is already higher than ol_{th}, the system does not engage DVFS to increase performance, the reason being that

Benchmark	ol_{th}	ww_{th}	Wall-Clock Time [s]	Power [W]	Energy [kJ]
FFT	0.24	2.4	1647.944	26.868	44.278
	0.16	2.4	1638.741	28.729	47.080
	0.20	3.0	1646.045	30.600	50.378
	0.24	3.6	1635.508	32.971	53.930
	0.16	3.6	1655.718	32.010	53.024
DES	0.24	2.4	1443.617	40.944	59.113
	0.16	2.4	1454.066	44.094	64.168
	0.20	3.0	1447.779	46.625	67.507
	0.24	3.6	1441.094	43.402	62.539
	0.16	3.6	1447.406	46.882	67.849
Histogram	0.24	2.4	1643.120	33.882	55.673
	0.16	2.4	1643.268	33.944	55.779
	0.20	3.0	1637.046	34.189	55.969
	0.24	3.6	1636.282	33.971	55.586
	0.16	3.6	1636.062	34.021	55.660
Image Filter	0.24	2.4	1568.998	20.212	31.712
	0.16	2.4	1541.283	21.159	32.596
	0.20	3.0	1559.915	21.420	33.417
	0.24	3.6	1539.534	20.971	32.281
	0.16	3.6	1575.832	20.845	32.852
Face Detection	0.24	2.4	1766.394	33.599	59.714
	0.16	2.4	1785.962	34.912	62.487
	0.20	3.0	1797.958	42.044	75.757
	0.24	3.6	1785.684	40.825	73.130
	0.16	3.6	1760.555	44.493	78.333

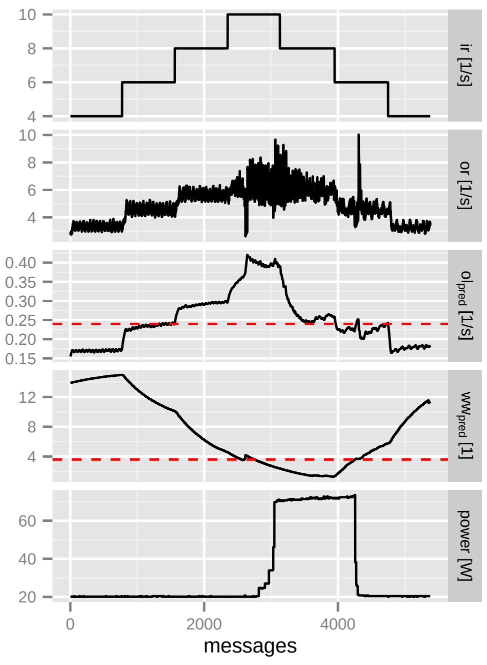
Table 6.3 Average Power Level, Wall-clock Time, and Total Energy Consumption for Different Threshold Values of ol_{th} and ww_{th} Under the DVFS Policy, as Mean Over Three Runs



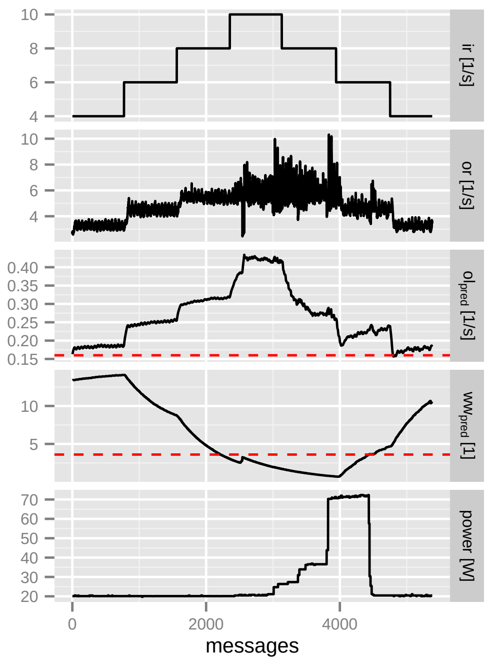
(a) $ol_{th} = 0.24, ww_{th} = 2.4$



(b) $ol_{th} = 0.16,\ W_{th} = 2.4$

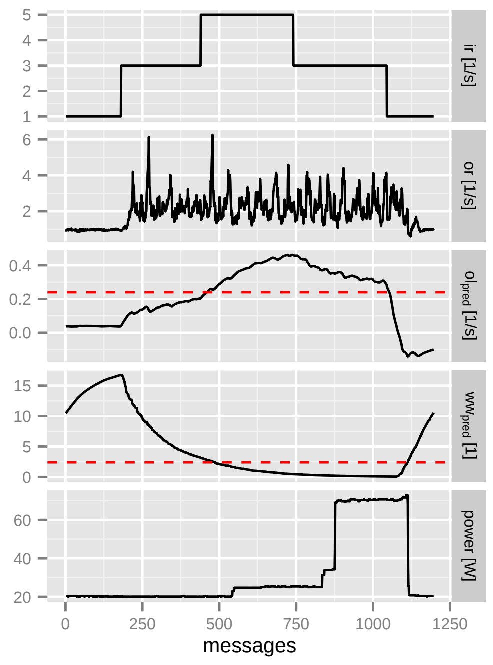


(c) $ol_{th} = 0.24, ww_{th} = 3.6$

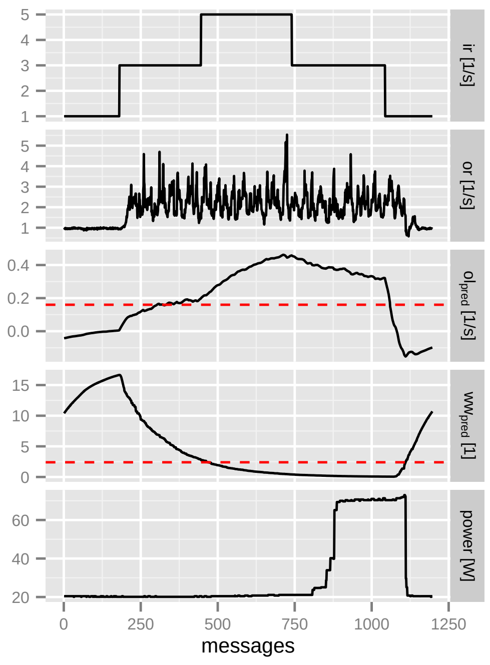


(d) $ol_{th} = 0.16, ww_{th} = 3.6$

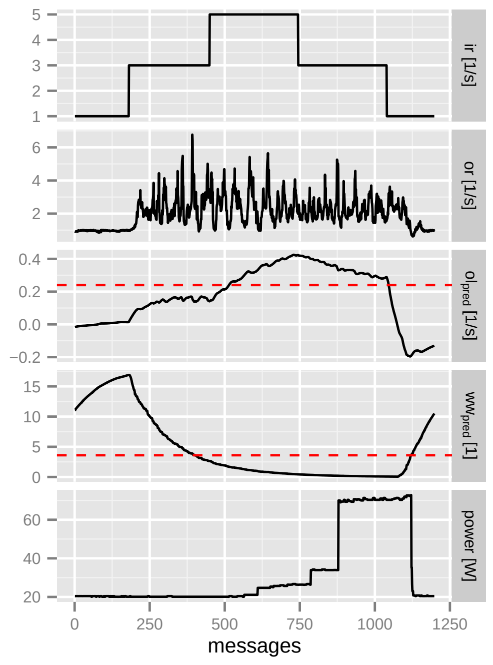
Figure 6.18 FFT with Different Choices of Thresholds ol_{th} and ww_{th} for DVFS



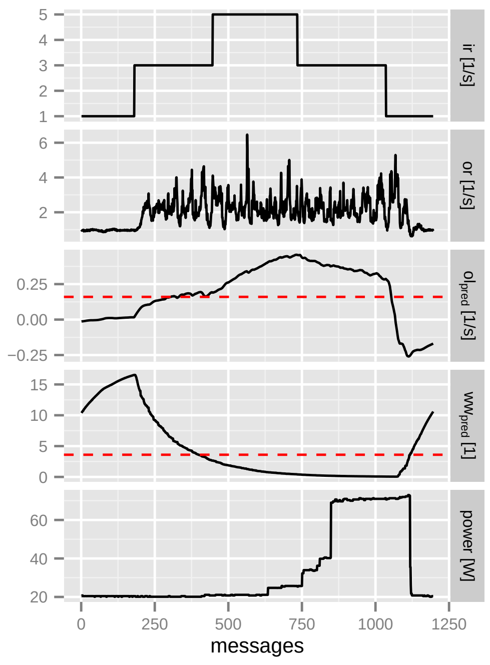
(a) $ol_{th} = 0.24, ww_{th} = 2.4$



(b) $ol_{th} = 0.16,\ ww_{th} = 2.4$

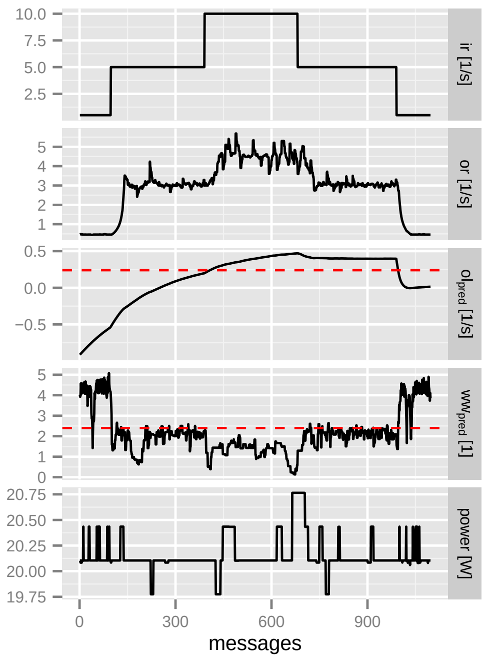


(c) $ol_{th} = 0.24, ww_{th} = 3.6$

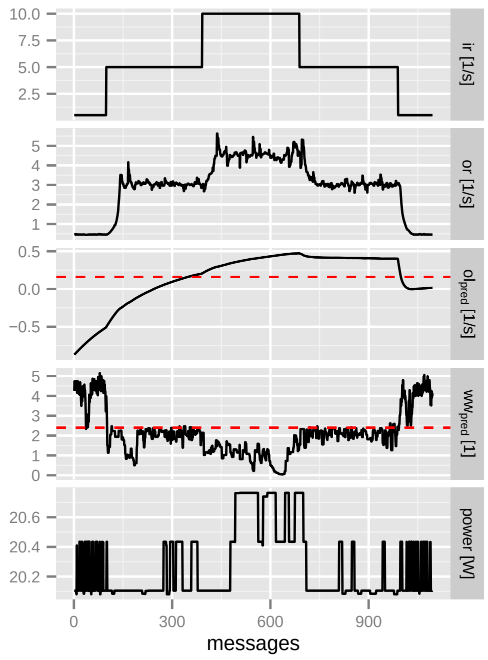


(d) $ol_{th} = 0.16, ww_{th} = 3.6$

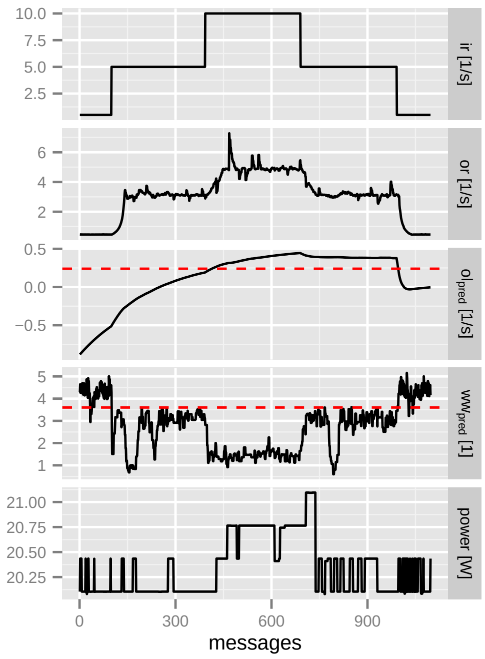
Figure 6.19 DES with Different Choices of Thresholds ol_{th} and ww_{th} for DVFS



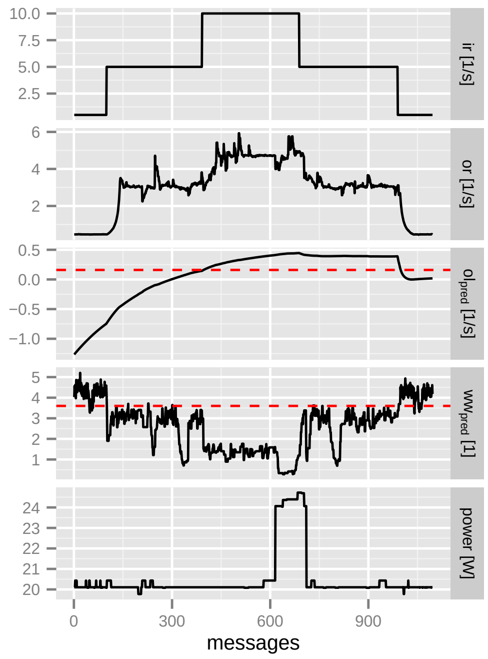
(a) $ol_{th} = 0.24, ww_{th} = 2.4$



(b) $ol_{th} = 0.16,\ W_{th} = 2.4$

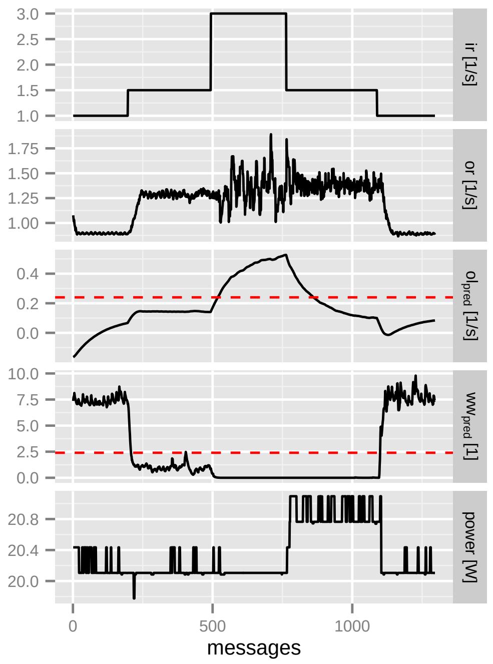


(c) $ol_{th} = 0.24, ww_{th} = 3.6$

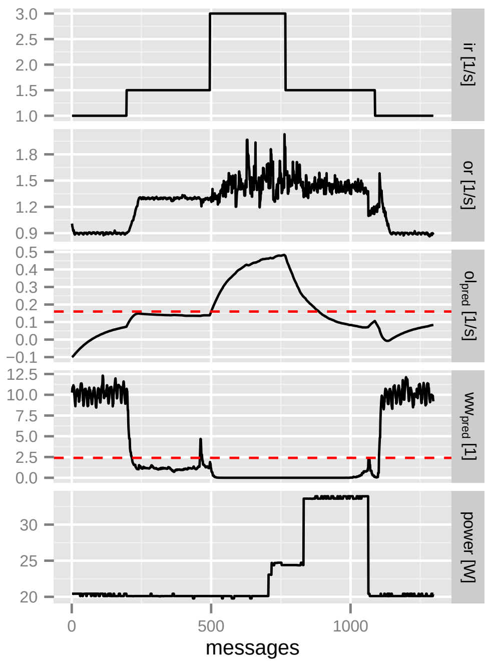


(d) $ol_{th} = 0.16, ww_{th} = 3.6$

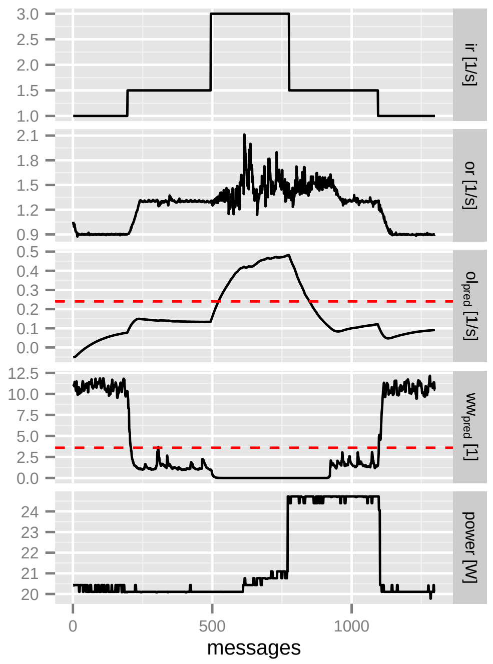
Figure 6.20 Histogram with Different Choices of Thresholds ol_{th} and ww_{th} for DVFS



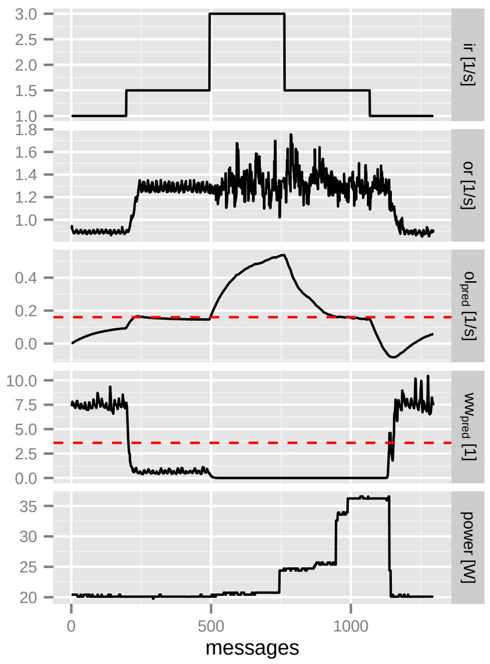
(a) $ol_{th} = 0.24, ww_{th} = 2.4$



(b) $ol_{th} = 0.16,\ W_{th} = 2.4$

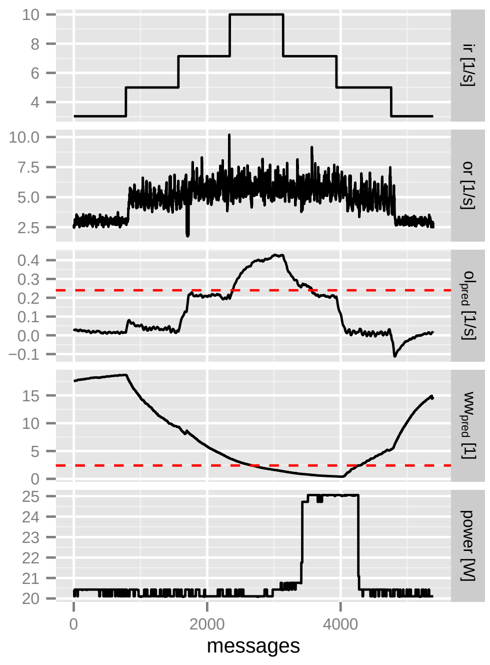


(c) $ol_{th} = 0.24, ww_{th} = 3.6$

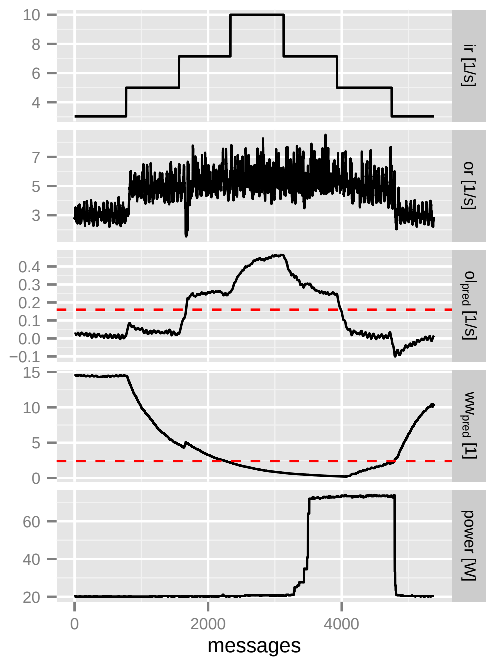


(d) $ol_{th} = 0.16, ww_{th} = 3.6$

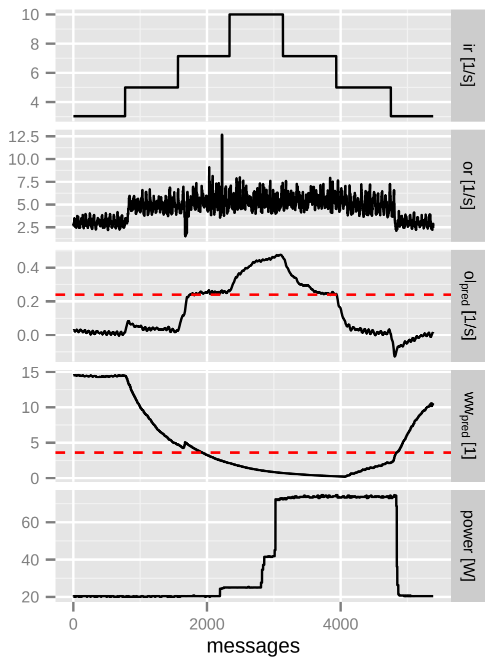
Figure 6.21 Image Filter with Different Choices of Thresholds ol_{th} and ww_{th} for DVFS



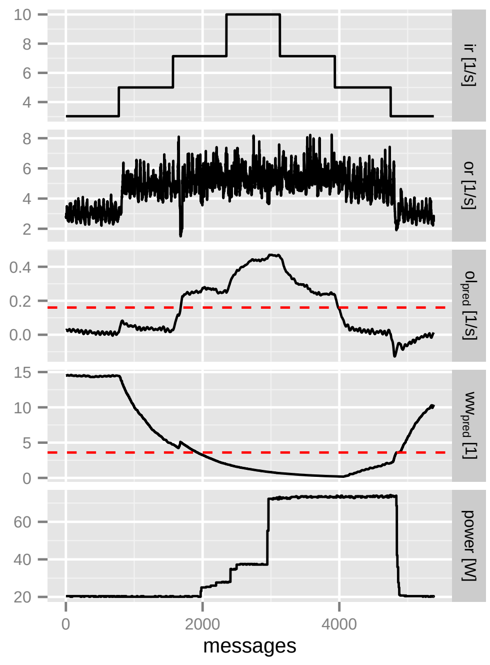
(a) $ol_{th} = 0.24, ww_{th} = 2.4$



(b) $ol_{th} = 0.16,\ ww_{th} = 2.4$



(c) $ol_{th} = 0.24, ww_{th} = 3.6$



(d) $ol_{th} = 0.16, ww_{th} = 3.6$

Figure 6.22 Face Detection with Different Choices of Thresholds ol_{th} and ww_{th} for DVFS

ww_{pred} is still lot lower than ww_{th}. Which means the overload that the system is experiencing can be/is dealt with by increasing the resource usage i.e. waiting workers. Once ww_{pred} is below threshold ww_{th} and if ol_{pred} is still above ol_{th} then DVFS is engaged. Lowering the overload threshold to $ol_{th} = 0.16$ makes the system reach its maximum performance state earlier and leave it later, as shown in Figure 6.18(b). If, in addition, the waiting threshold is raised to $ww_{th} = 3.6$, as shown in Figure 6.18(d), the system spends most of the time in its maximum performance state.

For the DES benchmark similar patterns to FFT can be seen in Figure 6.19. Figure 6.19(a) shows that ol_{pred} and ww_{pred} cross the thresholds at roughly the same time and we can see that power consumption starts to increase around same time (roughly around 500 messages). From Figure 6.19(c) we can see that even though ww_{pred} is below threshold around 375 messages, we do not see a corresponding change in power consumption, the reason being ol_{pred} has not reached the threshold yet. Once ol_{pred} crosses the threshold $ol_{th} = 0.24$ around 500 messages, we can see that power consumption starts to increase reflecting an increase in voltage/frequency.

we can observe different pattern for histogram benchmark in Figure 6.20. There is some subtle fluctuations in power consumption. This behaviour can be attributed to the fact that number of waiting worker ww_{pred} is hovering around threshold ww_{th}. As we can see from Figure 6.20(d), the fluctuation disappears when higher value is set for the threshold, i.e. $ww_{th} = 3.6$. Furthermore, the power consumption stays roughly the same for nearly all different threshold values for histogram and is much lower compared to the other benchmarks, i.e. FFT and DES. We can see nearly similar pattern—minus the power fluctuations—for image filter benchmark in Figure 6.21. The difference in the power consumption with varying value of thresholds is more noticeable than was for histogram benchmark. As can be observed in Figure 6.21(a) the higher value of thresholds ol_{th} and ww_{th} triggers small change in power consumption, whereas lowering the threshold ol_{th} to 0.16 allows ol_{pred} to reach it early at around 500 messages, but the spike in ww_{pred} around same time delays the decision to increase frequency, which happens around 600 messages as can be seen in Figure 6.21(b). The threshold ww_{th} is set to 3.6 as observed from Figure 6.21(d) and Figure 6.21(d). This means the the the frequency will be decreased at later time in comparison to the case in which the threashold is set at 2.4. The decrease in frequency is triggered around 1100 messages.

Figure 6.22 shows similar patterns for the Face Detection benchmark. Note that the power panel in Figure 6.18(a) Figure 6.22(a) has a different scale, indicating a very good choice of parameters, where the system never needs the maximal performance settings.

From all the experiments we can see that the waiting worker threshold ww_{th} determines the sensitivity towards low resources. If ww_{th} is too high, the speed of the processor (and thus the power consumption) might be increased earlier than necessary to handle the workload even though there are still some workers available which might be able to handle increased workload. If ww_{th} is too low, there is a risk that the system will not be able to handle the workload. The overload threshold ol_{th} determines the sensitivity towards high workload. If ol_{th} is too low, the speed of the cores (and thus the power consumption) might be kept higher than necessary to handle the work load. If ol_{th} is too high, there is a risk that system will not be able to handle the workload.

The above rules form a set of general guidelines for choosing suitable thresholds ww_{th} and ol_{th}. Notwithstanding, the concrete values are application-specific. In practice, these values could be determined using meta-heuristics to perform guided profiling of the application under consideration.

6.5 Chapter Summary

In this chapter we experimentally evaluated our approach. We ran two sets of experiments. The first set is to evaluate the efficiency of the RA-LPEL on the SCC in terms of throughput, latency and scalability. The second set was to evaluate the efficiency in terms of impact of DVFS and resulting reduction in power consumption.

We described our choices of use cases for the experiment. The use cases were implemented in a stream language S-Net, the uses cases were fast fourier transform, data encryption standard, colour histogram calculation, image filtering and face detection. The power optimisation experiments confirmed our theory that, even with simple DVFS strategy such as our, substantial (up to half or more for most of the benchmarks) energy reduction can be achieved.

Chapter 7

Related Work

Power management techniques to reduce energy consumption have been extensively studied in prior work. ***D****ynamic* ***V****oltage and* ***F****requency* ***S****caling* (DVFS) is a widely used technique to allow a trade-off between power consumption and performance. Our work focuses specifically on ***R****eactive* ***S****tream* ***P****rograms* (RSPs) running on many-core architecture. We have also developed HALLOC, a mechanism that provides ***O****perating* ***S****ystem* (OS) independent memory creation and management functionality. The novelty of HALLOC is that it allows memory (de)allocation and management across cores, each running a separate instance of an OS. This differs from previous work that we studied in the sense that most of the prior work has focused either on the multi-thread workload or multi-core running in a single system image setting, i.e. a single instance of an OS runs on all the cores.

In this chapter, we present an assortment of work based on similar ideas, or that in one way or other influenced our work. The work that is most closely related to ours, since it also focuses on S-Net and ***S****ingle-chip* ***C****loud* ***C****omputer* (SCC) is covered in § 7.1. Related work for memory management is presented in § 7.2. This is followed by related work in the DVFS arena in § 7.3. We conclude this chapter with a short summary in § 7.4.

7.1 Distributed S-Net on the SCC

Work by Verstraaten et al. [135, 136] is closely related to ours in the sense that it also uses the S-Net ***R****untime* ***S****ystem* (RTS) on the SCC. Although their approach is similar to ours, there are some fundamental differences. The main focus of their work was to provide a simplified way of programming the SCC and similar future architectures by using the S-Net, while we are targeting energy-efficient execution of the RSPs on the SCC and similar future many-core architectures.

Verstraaten et al. used distributed S-Net in their work. Traditionally, the distributed S-Net used ***M**essage **P**assing **I**nterface* (MPI) to transfer data between nodes when the stream crossed the node boundary, i.e. when the reader of a stream is not located on the same node where the writer is located. Verstraaten et al. employed SCC specific features, such as, ***L**ook**u**p **T**able* (LUT) remapping, ***M**essage **P**assing **B**uffer* (MPB) and interprocess interrupts to devise a communication mechanism that can be used instead of the MPI.

Regarding the programming side, a fundamental difference between our work and that of Verstraaten et al. is that their approach requires user annotations for the mapping of software components to cores, while our execution layer includes a dynamic load-balancing mechanism and thus does not need manual tuning of the schedule with such placement annotations. This is because Verstraaten et al. used a distributed version of S-Net with PTHREAD backend in their work, in which the mapping of the task/network-to-core is determined via static user annotations at compile time, as we already mentioned in Section 2.2. This can be difficult, as the end users have to manually annotate the source code and must have necessary knowledge of the topology of the targeted architecture. Furthermore, automatic system-wide load-balancing under dynamic demand is also very difficult to achieve, due to static task/network-to-core mapping. As mentioned, in our approach we do not need to use such annotations for the task/network-to-core mapping, allowing more portability across different architectures. In addition, our use of the centralised scheduler allows system-wide load-balancing under dynamic load demands.

On the technical side, for the LUT remapping they use 172 unused LUT pages and split them into two sections. The first section maps the physical memory, and using their custom `malloc` implementation, memory can be allocated in this section. The second section is used to map LUT pages sent by other cores dynamically. S-Net fields can be big and as such they are allocated in the first section. When field data needs to be sent across a node boundary, the sending core sends only an LUT entry, where data is located. On receiving core this LUT entry is mapped to its own second section, and as such, data can be transferred between cores without any need to copy to/from any buffer space. A small amount of data, such as S-Net metadata and LUT entry information is transmitted via MPB. Since there is no task migration taking place and apart from data nothing is shared between the cores, they were able to retain access to L2 caches of the SCC. We also use LUT remapping, but we do this statically, i.e. once LUT entries are mapped they are not changed during the execution of the program, this allowed us to create a shared memory on the SCC (where C pointers can be shared between cores as well). In addition, we use this shared memory to allocate everything, from the task structure to the S-Net field data. Furthermore, we assume shared everything approach and due to the lack of fine-grain control over L2 caches[1] on the SCC we had to disable L2 caches for our approach. The distributed version of S-Net runs several threads per core, such as the ***I****nput* ***M****anager* (IM), the ***O****utput* ***M****anager* (OM), the ***D****ata* ***F****etcher* (DF) and the worker thread, while in our approach the core executes only worker thread. So theoretically, there may be some difference in OS induced context switching overhead.

Although we would like to compare our approach with Verstraaten's in terms of throughput and scalability, this was not possible due to the problem with OS-level device drivers that were used by Verstraaten et al. and were made deprecated following the introduction of the newer version of the sccKit, and ***F****ield* ***P****rogrammable* ***G****ate* ***A****rray* (FPGA) hardware for the SCC controller. Also, the version of the Linux kernel used by Verstraaten et al. was replaced by a newer kernel (Linux kernel_3.1.4). This means that the mechanism that was employed to create device drivers at kernel level[2] was changed from *cdev* to *miscdevices*. Furthermore, there were also some changes in how interrupts were handled. Given that, with all these changes required at various levels, including kernel level, an attempt was made to get everything working, to compare the work of Verstraaten et al. with ours, but was not pursued any further [95].

It would have been interesting to have had the performance compared between these two approaches. But the inability in doing so is not a severe limitation, since our contributions are not purely performance focused. In our work we have developed an efficient, light-weight power management mechanism for RSPs on the SCC. Our power management mechanism should also be applicable to other future architectures that do not have the limitations caused by the SCC architecture.

7.2 Hierarchical Memory Management Mechanism

Our memory management mechanism HALLOC (described in Chapter 3.3) is specialised in allowing memory allocation/deallocation across different instance of OS executing on different cores on the ***N****etwork* ***o****n* ***C****hip* (NoC) architecture. Furthermore, our approach provides simple ***A****pplication* ***P****rogramming* ***I****nterface* (API) with functions like, scc_malloc and scc_free (similar to standard `malloc` and `free`), which abstracts away communication mechanism from the cores/processes and allows complex structures to be passed by reference, substantially simplifying the programming of distributed applications. The main distinction of HALLOC compared to other multi-

[1]The cache has to be flushed manually to ensure memory consistency, this cache flushing operation is expensive, costing approximately 1,000,000 cycles [9, 135, 136].

[2]The device drivers are used to map memory in different configurations e.g. MPB or not, cached, or uncached. Verstraaten et al. used one of the custom device drivers they created.

threaded allocators is that, HALLOC does not only provide memory management functionality, but in addition it also includes a mechanism that creates shared memory that works seamlessly on top of multiple cores of the SCC, with each cores running an independent instance of an OS. As such, this shared memory only exist at an application level, i.e. the OS is not even aware of the existence of this extra memory available. As the closest related work, we have identified some work that provide some sort of support for the shared memory either on SCC or work in similar manner to our approach.

In Intel's [79] ***P****rivately* ***O****wned* ***P****ublic* ***Sh****ared* ***M****emory* (POP-SHM) approach for the SCC, each core offers some of its private memory to create a pool of shared memory to share data with other cores. POP-SHM provides an API to create the shared object. Access to these objects has to be guarded by POP-SHM library calls, so that the library can take care of coherency by flushing the caches. The approach provides only a mechanism to share data. To perform computations the data must be copied to core local private memory first, which is cumbersome in case of multi-layer software. ***M****emory* ***E****fficient* ***Sh****aring* (MESH) [111] is a framework for memory-efficient sharing. It uses remote method invocation to pass access to a shared object between cores. The MESH framework uses POP-SHM to provide a shared memory. It provides a higher level of abstraction than POP-SHM.

S*oftware* ***M****anaged* ***C****ache-coherence* (SMC) [140] is a library for the SCC that provides coherent, shared, virtual memory, but it is the responsibility of the programmer to ensure that data is placed in the shared region and that operations to shared data are guarded by release/acquire calls.

Hoard [17] targets multithreaded applications and aims to avoid false sharing and being efficient. Hoard maintains one global heap and per-processor heaps; each thread can access only its heap and the global heap. When usage of per-processor heap's drops below a certain threshold, a fixed-size chunk of its memory is moved to global head to be used by another processor. Hoard allocates per-processor memory in what is called *superblock*. Each block allocated by Hoard has a superblock as an owner, to which it returns in case of a de-allocation. Another example of memory allocator algorithm with a per-processor private heap with owner is LKmalloc [85]. The allocator in Vee and Hsu [133] behaves in similar manner to Hoard, as de-allocated memory is returned when it meets a certain threshold.

In [5] authors present a memory allocator called *scalloc* that is fast, multi-core scalable, and provides low-fragmentation. The allocator is made-up of two parts: a frontend to manage memory in spans and a backend to manage empty spans. Spans are the same concept as superblocks in Hoard [17]. The spans are organised in 29 different size classes, ranging from 16 bytes to 1 MiB. Any request for memory over

1 MiB is allocated directly from OS using *mmap*. Span-pool is a global concurrent data structure that holds spans in different pools using arrays and stack. Each span is used to fulfil memory requests in terms of blocks, when all the blocks in span are freed, i.e. span has no allocated block, it is returned to span-pool. With regard to memory allocation and deallocation we have a similar approach, for example notion of ownership of memory block and separate lists to hold memory blocks that needs to be freed. For example, a block is added to a local free list when allocation was done by same core, or added to a remote free list otherwise. In the case of *scalloc*, these will be threads—not cores. The main difference in our approach is that our allocator works across different instances of OS, uses less complex data structure and can handle allocation bigger than 1 MiB in size.

7.3 Energy Optimisation

DVFS has attracted several research projects for its applications in efficient power management. The majority of the work in this area aims to minimise energy consumption by dynamically adjusting the voltage and frequency of the ***C****entral* ***P****rocessing* ***U****nit* (CPU) depending on the state of the system at different levels. As currently our work targets DVFS at the CPU/core level, we will restrict related work discussed in this section to the same level.

By observing memory operations like *memget*, *memput*, and *memcpy*, the authors in [42] identify slack periods of ***P****artitioned* ***G****lobal* ***A****ddress* ***S****pace* (PGAS) applications during these operations. By exploiting these slack periods, the authors propose a power management middleware to allow adjusting the power configuration at both PGAS application and runtime layers. The runtime layer makes decisions for power configurations based on the combination of predefined thresholds and adaptive thresholds based on the history. At the application layer, the middleware allows the programmer to use language extensions to define different power/performance policies, including maximum performance, balance power/performance, minimum power and maximum power reduction. The middleware is facilitated with a power controller to dynamically adjust the voltage and frequency, depending on the power configuration. To achieve the desired configuration, the power controller can operate in different modes, including adjusting both the voltage and frequency (DVFS), or adjusting only the frequency (***D****ynamic* ***F****requency* ***S****caling* (DFS)).

Thrifty Barrier [90], exploits barrier synchronisation imbalance in parallel applications to reduce energy consumption. In the thrifty barrier, when a thread reaches the barrier if it is not the last one to arrive it is put into sleep mode. In addition, being not the last thread to arrive at the barrier, the estimated wait time that the thread has to spin-loop aids the decision to put the thread into sleep. Also, this estimated wait time with a cache invalidation message from coherency protocol is used to wake up the sleeping thread when the last thread enters the barrier. The base assumption in this study is that computation phases that surround a particular barrier tend to perform the same type of computation every time they are executed. This work targets only the barrier synchronisation points in parallel application; any parallel programming model in which this points are not explicit can not use this approach.

Cai et al. proposed a mechanism to save energy consumption based on the identification of critical threads in parallel regions [26]. These threads are detected by the meeting point thread characterisation mechanism and are executed on cores with maximum frequency. The voltage and frequency are scaled down for cores executing other non-critical threads. *Meeting points* only target the application following ***S****ingle* ***P****rogram* ***M****ultiple* ***D****ata* (SPMD) semantics. They insert meeting points—a barrier like, although, unlike a barrier, not stopping a thread from progressing past the point—at the end of each parallel loop. Each thread compares its own progress with other threads based on these meeting points, and slows down if it is ahead. The main assumption in this work is that the total number of times each thread visits the meeting point is roughly the same. This work is limited to the class of applications that exhibit SPMD semantics.

Several works focus on dynamic power adaptation for MPI applications by observing the behaviours of MPI calls. For example, Lim et al. proposed a method which aims to dynamically reduce the power state (p-state) of CPUs during communication phases where computational is not extensive [93]. This approach uses different training algorithms to identify communication regions in the MPI program and to derive the appropriate p-state of for each region. A shifting component is then used to determine when the MPI program enters and leaves a communication region. This component is also responsible for changing the p-state to the desired level. The p-state is changed by writing the appropriate values of the ***F****requency* ***Id****entifier* (FID) and ***V****oltage* ***Id****entifier* (VID) to the ***M****odel* ***S****pecific* ***R****egister* (MSR).

Similarly, Iannou et al. proposed a method that aims to detect recurring communication and execution phases in MPI applications [61].The phase detector is designed by instrumenting MPI calls to measure the execution time of each call and the execution time of the program in between calls. The instrumentation information is then used in the so-called *supermaximal repeat string algorithm* [54] to detect different phases in the MPI program. The work also proposes a hierarchical power controller to automatically adjust the voltage and frequency during each detected phase. The adjustment is determined based not only on the information of the local power domains but also on the previously observed execution times of the phase on multiple frequency and voltage domains.

Likewise, Freeh and Lowenthal propose a profile directed scheme for power reduction in MPI programs [41]. They divide the program into phases based on trace data collected during a profile run of the application. In order to construct the phases, a program is divided into blocks identified by MPI operations, and change in memory pressure (measured as operations per miss). Two adjacent blocks are merged into a phase if their corresponding memory pressure is within some threshold. Each phase is then assigned a voltage, frequency value. As another example, Kappiah et al. proposed a method which exploits inter-node slack to detect non-bottleneck nodes [67]. The frequency of these nodes is scaled down so that their computations are potentially completed at the same time with bottleneck nodes.

Sassolas et al. present a hybrid DVFS and ***D**ynamic **P**ower **M**anagement* (DPM) technique to save energy in pipelines streaming applications [118]. The main focus of this work is that the computation is done in stages in a pipeline manner. Based on that knowledge, the observation they make is that throughput in streaming application is constrained by the duration of its slowest stage. So if other stages slow down, power consumption can be reduced. A monitor keeps account of the utilisation of a communication buffer, and based on two thresholds will decide if the producer is executing fast or slowly. Based on the position in the streaming pipeline, a priority is assigned to the task for scheduling purpose. Based on utilisation of a communication buffer, one of the DVFS modes—there are two modes used in this study: Turbo and Half-turbo—is chosen. Any unused resources and resources executing blocked tasks are put into *Deep Idle*, a DPM mode. Similarly, Alimonda et al. described their approach to applying DVFS to data-flow application [6]. Their work assumed that the mapping of tasks to ***P**rocessing **E**lements* (PEs), as well as their interconnection is already given. In order to make the DVFS decision they also monitored ***F**irst **I**n **F**irst **O**ut* (FIFO) buffer occupancy. Although unlike work by Sassolas et al. they did not use DPM.

As a static power adjustment approach, Rountree et al. used linear programming to derive the offline schedule for CPU frequency so that the energy saving is optimal [115]. The linear programming solver relies on the application communication trace to identify the critical path of execution. Using the power characteristics of the cluster, the solver derives a schedule that ensures that the node executing the critical path is never slowed down. Although this approach provides a nearly optimal schedule, it requires a complete trace of the program at each different frequency level. In addition, the linear programming solver is too slow to be efficiently utilised at runtime for dynamic adaptation. To overcome these problems, the authors combine this static approach and dynamic slack prediction into the *Adagio* RTS [116]. Adagio first generates a static schedule based on the predicted execution time. The schedule is dynamically adapted during the runtime based on the prediction of critical paths.

Wang and Lu proposed a threshold-based approach to reduce energy consumption for heterogeneous clusters [137]. First an offline analysis is performed to generate thresholds. These thresholds are then used to divide the workload into several ranges with different power configurations. At runtime, the workload for cluster is measured and, based on this measurement, future workload is predicted. The predicted workload is then used to decide the range and appropriate power configuration to be used. Similarly, Chen at al. proposed a DVFS scheme for heterogeneous clusters that satisfies ***Q**uality **o**f **S**ervice* (QoS) requirements [29]. They use approximation algorithms to analyse trade-off of time/space complexity and power consumption minimisation for three different QoS models. The latest survey by Bambagini and Marinoni [11] presents in-depth analysis of state-of-the-art energy-aware scheduling algorithms for real-time systems, including DVFS, DPM and hybrid approaches.

Chen et al. have presented a DVFS (and also static power consumption) approach that targets special multi-core platforms with time-triggered communication, i.e. ***T**ime-**D**ivision **M**ultiple **A**ccess* (TDMA) [28]. Our approach is not limited to such TDMA-based architectures. Also, the SCC used in our experiments does not have a time-triggered NoC. In addition, TDMA schemes are normally applied for highly dependable architectures, where the focus is more on dependability rather than efficient resource utilisation.

Poellabauer et al. have developed a feedback-based DVFS approach that aims to reduce the processor speed during ***I**nput/**O**utput* (I/O)-intensive phases where the maximum processor speed cannot be fully exploited [110]. The detection is made by observing the ratio of data cache misses per instruction, which they call ***M**emory **A**ccess **R**ate* (MAR). A high MAR denotes an I/O-intensive period with lots of accesses to data memory, while a low MAR denotes a computation-intensive period. The separation of execution is phased into I/O and computation works effectively for a single core. With multiple cores these phases tend to be out of sync among the cores, which reduces the chance that all cores are doing I/O at the same time.

Bini et al. have presented a DVFS framework for real-time systems [19]. This approach is based on the calculation of the so-called optimal processor speed which would allow all deadlines to be met. They then use two discrete speeds, one slightly below and the other slightly above the optimal speed, which are used to approximate the optimal speed, such that it is guaranteed that no deadline is missed.

In all the related work described above, the system state is observed at either the OS level or MPI level, or does not work for multi-core or works for specific real-time scheduling only. In contrast, in our approach we extract the system state at the application level, and we do not have to rely on specific task scheduling methods or NoC message routing schema.

As our approach specialises on the domain of streaming applications, we automatically adapt the power configuration to cope with the dynamic changes of the system's workload in an energy-efficient way. To do so, we use the correlation of the input and the output rates to detect overload situations in the system. In addition, we use monitoring information about idling cores to detect underload situations. In contrast to approaches where I/O is the deciding factor to reduce speed at I/O phases, e.g. MPI phase and barrier based approaches, we rely on temporal overload/underload situations in the system. Moreover, in contrast to some of the approaches discussed above, we do not derive an offline schedule that satisfies certain power consumption constraints. This is because for a dynamic load-balancing centralised scheduler such as ours it is not possible to use static schedules. In addition, we are targeting RSPs, where workload is dependent on the environment, and as such likely to have high variation. In some of the related work described above, the results are acquired by means of simulation. In contrast, we use an actual hardware platform to obtain results. Nevertheless, our approach is also hardware independent, i.e. it does not require any specific performance registers, and as such, is adapted to a new hardware platform without the need for any additional measures/modifications.

The DVFS related work discussed in this section can be summarised as:

Uniform vs. Core-based Control The majority of DVFS research assumes the power controllability of individual computing elements, for example, Cai et al. [26], Li et al. [90], Gamell et al. [42], Lim et al. [93], Iannou et al. [61], Poellabauer et al. [110], Sassolas et al. [118], and Alimonda et al. [6]. This is also the case for Chen at al. [29] with the focus on server farms.

For many-core processors like the SCC, such an individual control of computing elements is not applicable. While the SCC allows frequency switching at the tile level, the more important voltage setting can only be done at the level of individual voltage islands of four tiles each. Thus, we support an approach that does not rely on controllability of individual nodes, but rather does a processor-wide adaptation of power settings.

Dynamic vs. Static Scheduling Some DVFS approaches focus on static control of power settings, for example, Kappiah et al. [67], Rountree et al. [115],[116], and Chen et al. [28]. In contrast, our approach focuses on the support of dynamic scheduling problems.

Soft vs. Hard Real-Time Some DVFS approaches target real-time systems, deploying scheduling methods that rely on the knowledge of the computational demand of individual tasks in order to allow response-time analysis, for example, Wang and Lu [137], Bambagini and Marinoni [11], or Bini et al. [19]. Our approach does not rely on the knowledge of computational demands of individual tasks. However, on the downside, our approach is not applicable for frameworks with hard real-time requirements.

7.4 Chapter Summary

In this chapter, we have presented a range of work related to two aspects: memory allocation and management, and power consumption reduction by means of DVFS. To the best of our knowledge, there has been no specialised memory allocator that covers the non-single system image OS, and provide abstraction that hides details of the underlying communication mechanism. With simple API functions like, scc_malloc and scc_free (similar to standard `malloc` and `free`) our approach substantially simplifies the programming of distributed architectures. We focus on the domain of RSPs with dynamic program structures and variable workload with asynchronous communication. This allows us to use a simple matrix like correlation of the input rate and the output rate, and idling cores to drive DVFS related decision to automatically adapt the power configuration to cope with dynamic changes of the system's workload in an energy-efficient way.

Chapter 8

Conclusion and Outlook

With the advent of many-core architectures, general computing has been forced to become increasingly parallel to leverage the parallelism offered by architectures. Furthermore, industry has moved from raw performance to *performance per watt*. Reflecting the future trend there is a need for heavy computation, but power consumption is a first-class citizen as a constraint.

Motivated by these developments, this dissertation has presented the design and implementation of a resource-aware execution layer for ***R**eactive **S**tream **P**rograms* (RSPs) called ***R**esource-**A**ware **L**ight-weight **P**arallel **E**xecution **L**ayer* (RA-LPEL). Achieving more performance per watt is the main goal of the presented work. This chapter concludes the thesis by summarising the main features and contributions and discusses possible future work.

8.1 Thesis Summary

With the advent of multi-core/many-core architectures in everyday computing, programs have to run concurrently on more than one core to achieve maximum performance. This many-core shift in hardware forces the general computing paradigm to become increasingly parallel, to leverage maximum processing power and to improve application performance. In the wake of green computing, *performance per watt* has become widely used metric.

One way to program these many-core systems is to use *stream processing paradigm*. In the the stream processing paradigm, programs are constructed by computational nodes connected by streams. Streams are communication channels to transfer sequences of data between computation nodes. The stream processing model is well-suited for modern many-core architectures, as it embraces distributed processes that interact solely through explicit data streams. The stream programming relieves programmers from the burden of identification and exposition of the concurrency and simplifies the concurrency management on modern many-core architectures due to their implicit synchronisation. Furthermore, high-level coordination languages based on principles from data-flow programming allow software engineers to rapidly build maintainable parallel applications from sequential building blocks.

A stream program that continuously respond to external inputs and proceses virtually infinite sequences of data is referred to as a ***R**eactive **S**tream **P**rogram* (RSP). The RSPs work in tandem with their environment, and the load imposed by the environment may vary over time. This provides a unique opportunity to maximise the performance per watt. For example, if the system load imposed by the environment varies over time, dynamic power management techniques like ***D**ynamic **V**oltage and **F**requency **S**caling* (DVFS) can be used to effectively reduce/increase the voltage and frequency to allow energy-efficient execution of programs. Saving the power at the right time, without violating the system's throughput and latency constraints is a hard problem, as the future resource requirements can be unpredictable for many important applications.

Motivated by these developments, this dissertation has presented the design and implementation of RA-LPEL, which employs light-weight DVFS mechanism to support the energy-efficient execution of RSPs on the many-core architectures.

We introduced the ***S**ingle-chip **C**loud **C**omputer* (SCC) many-core processor and S-Net stream programming language as a concrete hardware and software vehicles to evaluate our approach. Next, we described our effort of bringing an existing execution layer for RSPs to the SCC. The details included a memory creation and allocation mechanism called ***H**ierarchical **Alloc**ator* (HALLOC) for the SCC. HALLOC allows memory management across distributed cores of the SCC, each running a separate instance of an ***O**perating **S**ystem* (OS). The main distinction of HALLOC is that it is not simply a memory manager; instead it is a complete mechanism that creates the shared memory—the instance of an OS running on the SCC is not aware of the existence of this shared memory—at the application level and provides functionality to manage (de)allocations.

We then provided taxonomy of the resource-aware execution together with formal definition of load ranges for RSPs with different arrival rates. We extended these definitions by including the impact the DVFS has on the matrices (throughput and latency). Next we describe RA-LPEL, a resource-aware execution layer for the RSPs. We described our use of the correlation between the input rate and the output rate to detect overload situations in the system. We also used monitoring information about idling cores to detect underload situations. Afterwards, we described the heuristics which use these temporal overload/underload situations in the system to derive and apply DVFS decisions.

We performed benchmarking to obtain results regarding two aspects of RA-LPEL. In the first aspect we studied performance in terms of scalability of throughput/latency for varying number of cores. We also compared our shared-memory approach with existing distributed backend as well. In the second aspect, we covered our main aim of investigating the impact of DVFS on the energy efficiency.

We identified throughput and the waiting worker as indicators of resource utilisation of RSPs. These metrics are independent of hardware, i.e. although the exact value might depend on the hardware architecture, the concept does not, and as such they are general enough to cover various architectures. We used throughput and ideal workers to derive policies that ensure power is saved at the right time without affecting performance.

The measurements showed a substantial gain (power consumption reduced nearly down to half in some cases) in energy efficiency for the scenario in which the DVFS was employed, compared with one without the DVFS. When only ***D**ynamic **F**requency **S**caling* (DFS) was enabled, there was reduction in the power consumption, but not as significantly as was achieved with DVFS.

8.2 Conclusion

We have presented "RA-LPEL", a resource-aware light-weight parallel execution layer, which aims to maximise the energy efficiency (performance per watt) of the RSPs on many-core architectures. In order to achieve that aim we also developed "HALLOC", a shared memory creation and management mechanism.

Porting an existing execution layer (***L**ight-weight **P**arallel **E**xecution **L**ayer* (LPEL) as a ground work) to the SCC seemed straight-forward, but proved to be more difficult than anticipated. One of the major problems we faced was that LPEL was designed for shared memory architectures. While the SCC provides some form of support for shared memory, e.g. ***M**essage **P**assing **B**uffer* (MPB), it was not sufficient for our purpose due to its size (8 KiB per core). In order to solve this problem we devised "HALLOC", a memory creation and management scheme, which provided nearly 2.5 GiB of shared memory, but on the down side, we had to lose four cores of the SCC. We had to overcome many problems while working on HALLOC, e.g. working out the order in which memory mapping and ***L**ookup **T**able* (LUT) remapping occurs, creating separation between two different memory (de)allocation mechanisms working together to avoid memory bugs, and problems with the caches.

One of the other problems we faced was how to handle cache coherency. The SCC does not provide cache coherency and since we assume shared everything, we had to disable caches of the SCC. This was due to multiple reasons. One of the reasons was the lack of L2 cache invalidation, i.e. that if the memory is shared and caches are used, the programmer has to manually flush the caches, a costly operation (costing approximately 1,000,000 cycles [9, 135, 136]). There was no way to partially flush the L2 cache. In addition, in the centralised scheduler used in our work, task-to-core mapping is changed very frequently for the purpose of the load-balance. In this case the cost of frequent cache-flushing overhead outweighs any gain in performance. For future non-cache coherent architectures, it would be desirable to have fine grain control over caches to avoid the problem related to the memory consistency and to keep the cost of cache flushing to a minimum.

The lack of debugging tools was another big problem. Memory analysis tools like Valgrind could have been very useful while designing HALLOC, but due to the SCC specific problems it would not work as expected. As an example, the instruction CL1INVMB is only available in the SCC and as such would be a problem. The shared memory created by HALLOC was above OS and as such it proved to be a problem to conduct a memory leak analysis with Valgrind. Although debugging with GDB was possible, it was very cumbersome. In the case of using GDB we had to connect to the SCC cores via ***Secure Shell*** (SSH), which meant that we would get 44 different consoles, one per core. One can imagine the problem when you have to click through 44 windows, every time you step forward/back to see what is happening.

On the power management side of the SCC, one of the promising aspects was that we could turn off cores to reduce power consumption. Unfortunately, when cores were turned off at runtime it was not possible to bring them back to normal operation. A hard reset of the SCC platform was required to resume normal operation. Another problem was that while voltage can be set for an island (made up of 8 cores) it was not possible to get readings for power consumption at this level, i.e. the SCC provides voltage and current readings only for the entire chip.

On the SCC, frequency scaling is fast (roughly 20 clock cycles) and will require synchronisation[1] between only 2 cores—a tile is frequency island—while voltage scaling is slow (roughly 40 ms) and will require synchronisation of 8 cores (4 tiles form a voltage island). This can be a problem when the workload is not balanced, as by the time synchronisation is complete for 8 cores of an island and then 6 islands of the SCC, the workload and tasl-to-core mapping may have already changed, rendering the validity of the calculation moot. An ideal power management would scale voltage and

[1]As we can only set the frequency for 2 cores, we have to make sure that voltage is set to the need of the core running with the highest frequency to ensure problem free operations.

frequency per core. However, this level of granularity would require a large amount of the die area and complex wiring to have a core local power supply.

Overall there are some problems with the SCC that need to be revisited/solved for future many-core architectures, e.g. the limited amount of ***T**est-and-**S**et* (T&S) registers (they are used to implement atomic locks), the rather small size of MPB memory, the limitation of a single outstanding memory operation (described in Section 2.5.4.2), the costly L2 cache flush operation (with no support for partial flush), relatively poor performance of some components, e.g. memory, and the aged feature set of the P54C architecture.

The main lesson we learned from this work was that while the SCC has many interesting features, it also has multiple problems that hold it back. In addition, the SCC was a research architecture and is no longer maintained, but the lessons we have learned will be useful for future many-core architectures.

We worked on a low level of abstraction in order to develop HALLOC, which provides others to use a familiar shared memory programming model to program the SCC without worrying about an underlying memory model. Furthermore, we devised a way to have a shared memory that is above OS, i.e. created and managed at application level. If not all, then part of the implementation will be useful for future work on the next generation of architectures. At the University of Hertfordshire we have already acquired a Kalray MPPA2-256 [66], a many-core processor with 256 cores. The cores in this architecture are arranged in 16 clusters, with each cluster consisting of 16 cores (similar to the islands on the SCC). While the MPPA2-256 has a distributed memory architectures, each cluster within behaves as a 16-core ***S**ymmetric **M**ulti**p**rocessing* (SMP) system, and as such we believe that the work we have done will provide a crucial base and open up more avenues to drive further research in the direction of energy-efficient execution of RSPs on future many-core architectures.

As our approach specialises on the domain of streaming applications, we use RSP-specific properties to automatically adapt the power configuration to cope with dynamic changes of the system's workload in an energy-efficient way. The RA-LPEL demonstrated the effectiveness of simple matrices and light-weight DVFS policies. The performance indicators we used are independent of any specific hardware architecture and as such will be useful for any future many-core architectures.

8.2.1 Research Questions: a Retrospective

With all the work described in this thesis, our aim was to answer the research questions set at the beginning of this study:

> *Is it possible to improve the adaptive resource utilisation and improve the energy efficiency of RSPs on many-core platforms by exploiting knowledge about the states of the system?*

This question was split into multiple sub-questions:

1. What is an efficient way to port an existing RSP execution layer to the SCC?
2. What are meaningful performance indicators to identify the workload situation of an RSP on a many-core processor?
3. Is it possible to have these performance indicators to be independent of any specific hardware feature of that many-core processor?
4. What are adequate strategies to optimise the performance per watt of RSPs on many-core platforms?
5. Is it possible to design DVFS strategies that are light-weight and simple, but still adequate to provide substantial reduction in energy consumption of RSPs?

Many of these sub-questions are dependent on each other. As an example, to answer the Question 1, we have to consider the properties this particular execution layer provides and how it would help to answer subsequent questions, e.g. Question 2 and help to identify performance indicators for RSPs.

We answered the Question 1 with the development of the first RSP execution layer for the many-core processor SCC that retains the shared memory programming model.[2] In order to retain shared memory programming model, we devised the HALLOC, a novel hierarchical memory creation and management mechanism for the SCC.

With the identification of, throughput, and number of waiting worker, as a meaningful performance metrics to indicate the workload situation, and the resource utilisation of RSPs, we answered the Question 2. These metrics that we identified are independent of the hardware i.e. although the exact value might depend on the hardware architecture, the concept does not, and as such they are general enough to cover various architectures, with this insight we answered the Question 3.

We answered Question 4, with the development of the ***R**esource-**A**ware **L**ight-weight **P**arallel **E**xecution **L**ayer* (RA-LPEL), which used throughput and ideal workers to derive heuristics that governed the DVFS related decisions. These heuristics allowed to save power at the right-time without affecting the performance constraints.

[2]There has been before a successful attempt to port a stream-processing environment to the SCC [135]. This approach was made for a concrete coordination language, called S-Net. Our execution layer is not bound to a particular coordination language, and did not require refactoring the code for the sake of avoiding shared memory communication.

We evaluated the efficiency of the RA-LPEL on the actual hardware, i.e. the SCC; in order to answer Question 5. With the experimental results obtained on the SCC, we confirmed that our light-weight and simple DVFS strategy—based on throughput and idle workers—substantially reduced the energy consumption of the RSPs, nearly down to half in some cases

Lastly, by answering all the sub-questions, we can say that, *it is* possible to improve the adaptive resource utilisation and improve the energy efficiency of RSPs on many-core platforms by exploiting knowledge about the states of the system, i.e. throughput and the number of idle workers.

8.3 Outlook

Within this dissertation, we have presented the current state of our research in optimising the performance of RSPs on many-core platforms. We have introduced approaches to exploit the knowledge of RSPs to effectively allow DVFS to reduce the power consumption without violating the system's throughput and latency constraints. Although the results presented in Chapter 6 have demonstrated the effectiveness of our approach, there are numerous was for further development.

It might be useful to conduct further studies to identify extra properties of the RSPs that can be used as the indicators of the system state. Currently we use throughput and resource utilisation, i.e. the number of waiting workers to derive the system state. This indicator could be modified to include the amount of time workers have to wait, although this would mean that the overhead would increase with the number of workers, as the waiting time for all the workers has to be calculated on each decision step. In contrast, the overhead of our current metric—the number of waiting workers—is constant. This could provide useful insights, for example, whether the more complex metrics are better indicators of the performance of the RSPs or not. Additionally, another interesting approach to pursue is to change the steps of how much the frequency/voltage change, e.g. instead of changing frequency one step at time, the difference can be calculated as % and ask to change the frequency accordingly in % as well.

Temperature has an impact on power consumption. Since the SCC provides a temperature sensor it would be interesting to extend the RA-LPEL and see how this extra information could be used to optimise energy usage even further.

Currently we manually use a *hill climbing* approach to see the effect of the different threshold values on power consumption. In practice, these values could be determined using meta-heuristics to perform guided profiling of the application under consideration. In this case, these meta-heuristics would have to be deployed during runtime. One option is to embed an optimisation technique such as hill climb or a genetic algorithm into RA-LPEL itself. This could be a promising direction that would allow a dynamic adaptation of threshold values at the runtime.

Another interesting direction is to have a conductor operating at the voltage island level, i.e. in the case of the SCC, one RA-LPEL conductor that would manage seven workers and multiple—ideally one per island—conductors operating on the chip. This would allow a more fine-grained DVFS at the island level, although this is simply a specialisation of our approach, in which we consider the SCC as just one island. Having multiple conductors will not allow the simple load balancing approach that we currently get with a centralised scheduler. Furthermore, this approach will require partitioning and mapping of RSP network to different islands—which is more difficult than it seems, as it also requires multiple changes to the centralised scheduler, e.g. to make it topology aware, need to develop new protocol for conductor-to-conductor communication, investigation into cost/benefit analysis at runtime (in order to allow task migration between different instances of a conductor for the purpose of load balance or DVFS).

Currently the conductor is in charge of turning the knobs to allow DVFS. With a higher number of workers the conductor will not scale sufficiently and will begin to struggle with making DVFS-related decisions and performing its scheduling duties. Which may very well be the case with Kalray MPPA2-256, with 256 cores. One way to resolve this problem is to implement the DVFS optimisation as a separate component and not as a part of the conductor.

Another interesting avenue to pursue is to apply our approach to other ***Runtime** Systems* (RTSs), e.g. FRONT. Since FRONT also targets shared memory architectures and we already provide the shared memory on the SCC, it would be an interesting study to investigate if similar increase in energy-efficiency can be observed for RSPs with different RTS with minimum implementation efforts.

Bibliography

[1] Ahmed Ben Achballah and Slim Ben Saoud. A Survey of Network-On-Chip Tools. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 4(9), December 2013. doi: 10.14569/IJACSA.2013.040910. Available at: http://dx.doi.org/10.14569/IJACSA.2013.040910.

[2] William B Ackerman. Data Flow Languages. *IEEE Computer*, 15(2):15–25, February 1982. ISSN 0018-9162. doi: 10.1109/MC.1982.1653938. Available at: http://dx.doi.org/10.1109/MC.1982.1653938.

[3] Anant Agarwal. The Tile Processor: A 64-Core Multicore for Embedded Processing. In *In 11th Annual Workshop on High Performance Embedded Computing (HPEC)*, 2007. Available at: https://www.researchgate.net/publication/265527779_The_Tile_Processor_A_64-Core_Multicore_for_Embedded_Processing.

[4] Ankur Agarwal, Cyril Iskander, and Ravi Shankar. Survey of Network on Chip (NoC) Architectures & Contributions. *Journal of Engineering, Computing and Architecture*, 3(1):21–27, January 2009. Available at: https://www.researchgate.net/publication/238536425_Survey_of_Network_on_Chip_NoC_Architectures_Contributions.

[5] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. Fast, Multicore-Scalable, Low-Fragmentation Memory Allocation through Large Virtual Memory and Global Data Structures. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 451–469, New York, NY, USA, 2015. ACM. doi: 10.1145/2814270.2814294. Available at: http://doi.acm.org/10.1145/2814270.2814294.

[6] Andrea Alimonda, Salvatore Carta, Andrea Acquaviva, Alessandro Pisano, and Luca Benini. A Feedback-based Approach to DVFS in Data-flow Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(11):1691–1704, November 2009. ISSN 0278-0070. doi: 10.1109/TCAD.2009.2030439. Available at: http://dx.doi.org/10.1109/TCAD.2009.2030439.

[7] James H. Anderson and Sanjoy K. Baruah. Energy-aware Implementation of Hard-real-time Systems upon Multiprocessor Platforms. In Seong-Moo Yoo and Hee Yong Youn, editors, *In Proceedings of the ISCA 16th International Conference on Parallel and Distributed Computing Systems*, pages 430–435. ISCA, August 2003. ISBN 1-880843-48-X. Available at: https://www.

researchgate.net/publication/2837907_Energy-Aware_Implementation_of_Hard-Real-Time_Systems_Upon_Multiprocessor_Platforms.

[8] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[9] Roy Bakker. Exploring the Intel Single-chip Cloud Computer and its possibilities for SVP. Master's thesis, University of Amsterdam, 2011. Available at: https://staff.fnwi.uva.nl/r.bakker/MSc_thesis_Roy_Bakker.pdf.

[10] Roy Bakker, Michiel W. van Tol, and Andy D. Pimentel. Emulating Asymmetric MPSoCs on the Intel SCC Many-core Processor. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 520–527, Washington, DC, USA, February 2014. Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/pdp.2014.104. Available at: http://dx.doi.org/10.1109/pdp.2014.104.

[11] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. Energy-Aware Scheduling for Real-Time Systems: A Survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):1–34, January 2016. ISSN 1539-9087. doi: 10.1145/2808231. Available at: http://doi.acm.org/10.1145/2808231.

[12] David Bednárek, Jirı Dokulil, Jakub Yaghob, and F Zavoral. The bobox project parallelization framework and server for data processing. *Charles University in Prague, Technical Report*, 1, 2011. Available at: http://www.ksi.mff.cuni.cz/content/techreports/TR2011-1-Bobox.pdf.

[13] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John F. Brown III, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, Washington, DC, USA, February 2008. Institute of Electrical & Electronics Engineers (IEEE). ISBN 978-1-4244-2010-0. doi: 10.1109/ISSCC.2008.4523070. Available at: http://dx.doi.org/10.1109/ISSCC.2008.4523070.

[14] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, January 2002. ISSN 0018-9162. doi: 10.1109/2.976921. Available at: http://dx.doi.org/10.1109/2.976921.

[15] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, ISLPED '99, pages 206–211, New York, NY, USA, 1999. ACM.

ISBN 1-58113-133-X. doi: 10.1145/313817.313927. Available at: http://doi.acm.org/10.1145/313817.313927.

[16] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, Jun 2000. doi: 10.1109/92.845896. Available at: http://dx.doi.org/10.1109/92.845896.

[17] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *ACM Sigplan Notices*, 35(11):117–128, November 2000. ISSN 0362-1340. doi: 10.1145/356989.357000. Available at: http://doi.acm.org/10.1145/356989.357000.

[18] Gérard Berry. Real Time Programming: Special Purpose or General Purpose Languages. In *IFIP Congress*, pages 11–17, 1989. Available at: https://hal.inria.fr/inria-00075494.

[19] Enrico Bini, Giorgio Buttazzo, and Giuseppe Lipari. Minimizing CPU Energy in Real-time Systems with Discrete Speed Management. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(4):31:1–31:23, July 2009. ISSN 1539-9087. doi: 10.1145/1550987.1550994. Available at: http://doi.acm.org/10.1145/1550987.1550994.

[20] Atman Binstock. Powering the Rift, May 2015. Available at: https://www.oculus.com/en-us/blog/powering-the-rift/.

[21] Tobias Bjerregaard and Shankar Mahadevan. A Survey of Research and Practices of Network-on-chip. *ACM Computing Surveys*, 38(1), June 2006. ISSN 0360-0300. doi: 10.1145/1132952.1132953. Available at: http://doi.acm.org/10.1145/1132952.1132953.

[22] Shekhar Borkar. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: 10.1145/1278480.1278667. Available at: http://doi.acm.org/10.1145/1278480.1278667.

[23] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54:67–77, May 2011. ISSN 0001-0782. doi: 10.1145/1941487.1941507. Available at: http://doi.acm.org/10.1145/1941487.1941507.

[24] Andries Brouwer and Michael Kerrisk. Linux Programmer's Manual, 2004. Available at: http://man7.org/linux/man-pages/man2/mmap.2.html. [Online, accessed 1-Apr-2012].

[25] William H. Burge. Stream Processing Functions. *IBM Journal of Research and Development*, 19(1):12–25, January 1975. ISSN 0018-8646. doi: 10.1147/rd.191.0012. Available at: http://dx.doi.org/10.1147/rd.191.0012.

[26] Qiong Cai, José González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. Meeting Points: Using Thread Criticality to Adapt Multicore Hardware to Parallel Regions. In *Proceedings of the*

17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, pages 240–249, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454149. Available at: http://doi.acm.org/10.1145/1454115.1454149.

[27] Roger D. Chamberlain, Mark A. Franklin, Eric J. Tyson, James H. Buckley, Jeremy Buhler, Greg Galloway, Saurabh Gayen, Michael Hall, E.F. Berkley Shands, and Naveen Singla. Auto-Pipe: Streaming Applications on Architecturally Diverse Systems. *IEEE Computer*, 43(3):42–49, March 2010. ISSN 0018-9162. doi: 10.1109/mc.2010.62. Available at: http://dx.doi.org/10.1109/MC.2010.62.

[28] Gang Chen, Kai Huang, and Alois Knoll. Energy Optimization for Real-time Multiprocessor System-on-chip with Optimal DVFS and DPM Combination. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):1–21, March 2014. ISSN 1539-9087. doi: 10.1145/2567935. Available at: http://doi.acm.org/10.1145/2567935.

[29] Jian-Jia Chen, Kai Huang, and Lothar Thiele. Dynamic Frequency Scaling Schemes for Heterogeneous Clusters under Quality of Service Requirements. *Journal of Information Science and Engineering*, 28(6):1073–1090, 2012. Available at: https://www.researchgate.net/publication/285769401_Dynamic_Frequency_Scaling_Schemes_for_Heterogeneous_Clusters_under_Quality_of_Service_Requirements.

[30] George Chrysos. Intel® Xeon Phi ™ X100 Family Coprocessor - the Architecture. *Intel Whitepaper*, 2012. Available at: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[31] William T Cochran, James W Cooley, David L Favin, Howard D Helms, Reg A Kaenel, William W Lang, George C Maling Jr, David E Nelson, Charles M Rader, and Peter D Welch. What is the Fast Fourier Transform? *Proceedings of the IEEE*, 55(10):1664–1674, October 1967. ISSN 0018-9219. doi: 10.1109/PROC.1967.5957. Available at: http://dx.doi.org/10.1109/PROC.1967.5957.

[32] Intel Corporation. Intel ® Xeon Phi ™ Coprocessor, 2012. Available at: https://software.intel.com/en-us/mic-developer.

[33] Intel Corporation. Intel ® Xeon Phi ™ Coprocessor System Software Development Guide, March 2014. Available at: https://software.intel.com/sites/default/files/managed/09/07/xeon-phi-coprocessor-system-software-developers-guide.pdf.

[34] Jack B. Dennis. Stream Data Types for Signal Processing. In Guang R. Gao, Lubomir Bic, and Jean-Luc Gaudiot, editors, *Advanced Topics in Dataflow Computing and Multithreading*, pages 87–101. IEEE Computer Society Press, 1995. Available at: https://www.researchgate.net/publication/2758318_Stream_Data_Types_for_Signal_Processing.

[35] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108. Available at: http://doi.acm.org/10.1145/2000064.2000108.

[36] Zbyněk Falt, David Bednárek, Martin Kruliš, Jakub Yaghob, and Filip Zavoral. Bobolang: A Language for Parallel Streaming Applications. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 311–314, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2749-7. doi: 10.1145/2600212.2600711. Available at: http://doi.acm.org/10.1145/2600212.2600711.

[37] Jianbin Fang, Ana Lucia Varbanescu, Henk Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. An Empirical Study of Intel Xeon Phi. *CoRR*, abs/1310.5842, 2013. Available at: http://arxiv.org/abs/1310.5842.

[38] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0, September 2012. Available at: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[39] John Franco, Daniel P. Friedman, and Steven D. Johnson. Multi-way Streams in Scheme. *Journal of Computer Languages*, 15(2):109–125, April 1990. ISSN 0096-0551. doi: 10.1016/0096-0551(90)90014-G. Available at: http://dx.doi.org/10.1016/0096-0551(90)90014-G.

[40] Mark A Franklin, Eric J Tyson, James Buckley, Patrick Crowley, and John Maschmeyer. Auto-pipe and the X Language: A pipeline Design Tool and Description Language. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2006*, pages 10–pp, Washington, DC, USA, April 2006. Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/ipdps.2006.1639353. Available at: http://dx.doi.org/10.1109/IPDPS.2006.1639353.

[41] Vincent W. Freeh and David K. Lowenthal. Using Multiple Energy Gears in MPI Programs on a Power-scalable Cluster. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 164–173, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065967. Available at: http://doi.acm.org/10.1145/1065944.1065967.

[42] Marc Gamell, Ivan Rodero, Manish Parashar, and Rajeev Muralidhar. Exploring Cross-layer Power Management for PGAS Applications on the SCC Platform. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'12, pages 235–246, New York, NY, USA, 2012. Association for Computing Machinery (ACM). ISBN 978-1-4503-0805-2. doi: 10.1145/2287076.2287113. Available at: http://doi.acm.org/10.1145/2287076.2287113.

[43] Bert Gijsbers. An Efficient Scalable Work-Stealing Runtime System for the S-Net Coordination Language. Master's thesis, University of Amsterdam, 2013. Available at: http://dare.uva.nl/document/477383.

[44] Bert Gijsbers and Clemens Grelck. An Efficient Scalable Runtime System for Macro Data Flow Processing Using S-Net. *International Journal of*

Parallel Programming, 42(6):988–1011, December 2014. ISSN 0885-7458. doi: 10.1007/s10766-013-0271-8. Available at: http://dx.doi.org/10.1007/s10766-013-0271-8.

[45] Peter Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM Whitepaper*, pages 1–8, September 2011.

[46] Clemens Grelck and Frank Penczek. Implementing S-Net: A Typed Stream Processing Language, Part I: Compilation, Code Generation and Deployment V0.4. Technical report, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2007. Available at: http://www.snet-home.org/?page_id=7.

[47] Clemens Grelck and Frank Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*, pages 60–79. Springer, 2008. ISBN 978-3-642-24451-3. doi: 10.1007/978-3-642-24452-0_4. Available at: http://dx.doi.org/10.1007/978-3-642-24452-0_4.

[48] Clemens Grelck and Sven-Bodo Scholz. SAC—A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming*, 34(4):383–427, July 2006. ISSN 1573-7640. doi: 10.1007/s10766-006-0018-x. Available at: http://dx.doi.org/10.1007/s10766-006-0018-x.

[49] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(02):221–237, June 2008. doi: 10.1142/S0129626408003351. Available at: http://dx.doi.org/10.1142/S0129626408003351.

[50] Clemens Grelck, Jukka Julku, and Frank Penczek. Distributed S-Net: Design and Implementation. In Marco Morazán, editor, *21st International Symposium on Implementation and Application of Functional Languages (IFL'09)*, pages 39–54. Seton Hall University, 2009. Available at: https://www.researchgate.net/publication/228767665_Distributed_S-Net_design_and_implementation.

[51] Clemens Grelck, Jukka Julku, and Frank Penczek. Distributed S-Net: Cluster and Grid Computing Without the Hassle. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, CCGRID '12, pages 410–418, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4691-9. doi: 10.1109/CCGrid.2012.140. Available at: http://dx.doi.org/10.1109/CCGrid.2012.140.

[52] Clemens Grelck, Alex (eds): Shafarenko, Frank Penczek, Jukka Julku, Haoxuan Cai, Philip Hölzenspies, Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, and Bert Gijsbers. S-Net Language Report 2.1. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2013. Available at: http://www.snet-home.org/?page_id=7.

[53] Philipp Gschwandtner, Thomas Fahringer, and Radu Prodan. Performance analysis and benchmarking of the Intel SCC. In *2011 IEEE International Conference on Cluster Computing*, pages 139–149, Washington, DC, USA, September 2011. Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/CLUSTER.2011.24. Available at: http://dx.doi.org/10.1109/CLUSTER.2011.24.

[54] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK, 1997. ISBN 0-521-58519-8.

[55] Linley Gwennap. Adapteva: More Flops, Less Watts. *Microprocessor Report*, 6 (13):11–02, 2011. Available at: http://www.adapteva.com/wp-content/uploads/2012/08/adapteva_mpr.pdf.

[56] David Harel and Amir Pnueli. Logics and Models of Concurrent Systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag, New York, NY, USA, 1985. ISBN 0-387-15181-8. Available at: http://dl.acm.org/citation.cfm?id=101969.101990.

[57] Fouzhan Hosseini, Amir Fijany, and Jean-Guy Fontaine. Highly Parallel Implementation of Harris Corner Detector on CSX SIMD Architecture. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops: HeteroPar, HPCC, HiBB, CoreGrid, UCHPC, HPCF, PROPER, CCPI, VHPC, Ischia, Italy, August 31–September 3, 2010, Revised Selected Papers*, pages 137–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21878-1. doi: 10.1007/978-3-642-21878-1_17. Available at: http://dx.doi.org/10.1007/978-3-642-21878-1_17.

[58] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-Core IA-32 Message-passing Processor with DVFS in 45nm CMOS. In *2010 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 108–109, Washington, DC, USA, February 2010. Institute of Electrical & Electronics Engineers (IEEE). ISBN 978-1-4244-6033-5. doi: 10.1109/ISSCC.2010.5434077. Available at: http://dx.doi.org/10.1109/ISSCC.2010.5434077.

[59] Adapteva Inc. Epiphany Architecture Reference, 2008. Available at: http://www.adapteva.com/docs/epiphany_arch_ref.pdf. [Online, Version 14.03.11, accessed 1-Apr-2015].

[60] Institute of Electrical and Electronic Engineers. Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language). IEEE Standard

1003.1c–1995, Institute of Electrical & Electronics Engineers (IEEE), New York City, New York, USA, 1995. also ISO/IEC 9945-1:1990b.

[61] Nikolas Ioannou, Michael Kauschke, Matthias Gries, and Marcelo Cintra. Phase-Based Application-Driven Hierarchical Power Management on the Single-chip Cloud Computer. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 131–142, Washington, DC, USA, October 2011. Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/pact.2011.19. Available at: http://dx.doi.org/10.1109/pact.2011.19.

[62] Jim Jeffers and James Reinders. *Intel ® Xeon Phi ™ Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. ISBN 9780124104143, 9780124104945.

[63] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, March 2004. ISSN 0360-0300. doi: 10.1145/1013208.1013209. Available at: http://doi.acm.org/10.1145/1013208.1013209.

[64] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland. Available at: https://www.researchgate.net/publication/221329680_The_Semantics_of_Simple_Language_for_Parallel_Programming.

[65] Gilles Kahn and David B. MacQueen. Coroutines and Networks of Parallel Processes. In *IFIP Congress*, pages 993–998, 1977. Available at: https://www.researchgate.net/publication/29626306_Coroutines_and_Networks_of_Parallel_Processes.

[66] KALRAY Corporation. KALRAY MPPA MANYCORE Flyer, 2012. http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf accessed 10-01-2015.

[67] Nandini Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 33–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi: 10.1109/SC.2005.39. Available at: http://dx.doi.org/10.1109/SC.2005.39.

[68] Nilesh Karavadara, Simon Folie, Michael Zolda, Vu Thien Nga Nguyen, and Raimund Kirner. A Power-Aware Framework for Executing Streaming Programs on Networks-on-Chip. In *Proc. International Workshop on Performance, Power and Predictability of Many-Core Embedded Systems (3PMCES'14)*. Electronic Chips and Systems Design Initiative (ECSI), March 2014. Available at: https://www.researchgate.net/publication/281650850_A_Power-Aware_Framework_for_Executing_Streaming_Programs_on_Networks-on-Chip.

[69] Nilesh Karavadara, Michael Zolda, Vu Thien Nga Nguyen, Jens Knoop, and Raimund Kirner. Dynamic Power Management for Reactive Stream Processing on the SCC Tiled Architecture. *EURASIP Journal on Embedded Systems*, 2016

(1):1–17, June 2016. ISSN 1687-3963. doi: 10.1186/s13639-016-0035-9. Available at: http://dx.doi.org/10.1186/s13639-016-0035-9.

[70] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 1st edition, 2008. ISBN 1598292080, 9781598292084.

[71] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall Englewood Cliffs, 2 edition, 1988. ISBN 0-13-110362-8.

[72] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, San Francisco, USA, 2010. ISBN 978-1-59327-220-3.

[73] Intel Labs. Manycore Applications Research Community (MARC), 2010. Available at: http://communities.intel.com/community/marc.

[74] Intel Labs. Bypass Causes Data Corruption in the MPB. Technical report, Intel Corporation, August 2010. Available at: http://communities.intel.com/docs/DOC-5405. Bugzilla: http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=46.

[75] Intel Labs. SCC External Architecture Specification (EAS). Technical report, Intel Corporation, November 2010. Available at: https://communities.intel.com/docs/DOC-5852. Revision 1.1.

[76] Intel Labs. Shared Memory on Rock Creek. Technical report, Intel Corporation, October 2010. Available at: https://communities.intel.com/docs/DOC-5644.

[77] Intel Labs. *The SCC Programmer's Guide V(1.0)*. Intel Corporation, 2010. Available at: https://communities.intel.com/docs/DOC-5684.

[78] Intel Labs. Using the RCCE Power Management Calls. Technical report, Intel Labs, September 2011. Available at: https://communities.intel.com/docs/DOC-18982. Revision 1.1.

[79] Intel Labs. The sccKit 1.4.0 User's Guide. Technical report, Intel Corporation, March 2011. Revision 1.0.

[80] Intel Labs. The sccKit 1.4.x User's Guide. Technical report, Intel Corporation, November 2011. Available at: https://communities.intel.com/docs/DOC-6241. Revision 1.1.

[81] Intel Labs. Activating L2 Reset Halts Core. Technical report, Intel Corporation, May 2011. Available at: http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=215.

[82] Intel Labs. Cache flushing with shared memory unreliable. Technical report, Intel Corporation, April 2011. Available at: http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=195.

[83] Intel Labs. The SCC Platform Overview. Technical report, Intel Corporation, January 2012. Available at: https://communities.intel.com/docs/DOC-5512. Revision 0.8.

[84] Peter J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965. ISSN 0001-0782. doi: 10.1145/363744.363749. Available at: http://doi.acm.org/10.1145/363744.363749.

[85] Per-Åke Larson and Murali Krishnan. Memory Allocation for Long-running Server Applications. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 176–185, New York, NY, USA, 1998. ACM. ISBN 1-58113-114-3. doi: 10.1145/286860.286880. Available at: http://doi.acm.org/10.1145/286860.286880.

[86] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. ISSN 0018-9219. doi: 10.1109/proc.1987.13876. Available at: http://dx.doi.org/10.1109/PROC.1987.13876.

[87] Edward A. Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987. doi: 10.1109/TC.1987.5009446. Available at: http://dx.doi.org/10.1109/TC.1987.5009446.

[88] Edward A Lee and Thomas M Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. ISSN 0018-9219. doi: 10.1109/5.381846. Available at: http://dx.doi.org/10.1109/5.381846.

[89] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, ISLPED '99, pages 267–269, New York, NY, USA, 1999. ACM. ISBN 1-58113-133-X. doi: 10.1145/313817.313944. Available at: http://doi.acm.org/10.1145/313817.313944.

[90] Jian Li, Jose F. Martinez, and Michael C. Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 14–23, Madrid, Spain, 2004. IEEE Computer Society. ISBN 0-7695-2053-7. doi: 10.1109/HPCA.2004.10018. Available at: http://dx.doi.org/10.1109/HPCA.2004.10018.

[91] Davide Libenzi. *GNU Portable Coroutine Library, Version 1.12*, 2012. Available at: http://www.xmailserver.org/libpcl.html.

[92] David J Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys (CSUR)*, 25(3): 303–338, September 1993. ISSN 0360-0300. doi: 10.1145/158439.158907. Available at: http://doi.acm.org/10.1145/158439.158907.

[93] Min Yeol Lim, Vincent W Freeh, and David K Lowenthal. Adaptive, Transparent Frequency, and Voltage Scaling of Communication Phases in MPI Programs. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 14–14, New York, NY, USA, December 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188567. Available at: http://doi.acm.org/10.1145/1188455.1188567.

[94] Many-core Applications Research Community. RCCE_iset_power function causes cores to be unstable. https://communities.intel.com/thread/26799 accessed 19-Apr-2014, 2011.

[95] Many-core Applications Research Community. The New SCC Linux Modifications. https://communities.intel.com/thread/28313 accessed 16-May-2013, 2012.

[96] Donald W. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *SIAM Journal on Applied Mathematics*, 11(2):431–441, June 1963. doi: 10.1137/0111030. Available at: http://dx.doi.org/10.1137/0111030.

[97] Tim Mattson and Rob van der Wijngaart. RCCE: a Small Library for Many-Core Communication. Technical report, Intel Labs, January 2011. Available at: https://communities.intel.com/docs/DOC-5628. Software Version 2.0-release; Document Version 0.90.

[98] Microsoft. The Manycore Shift White Paper. Technical report, Microsoft Corporation, 2007. Available at: https://www.microsoft.com/en-gb/download/details.aspx?id=17702.

[99] Sparsh Mittal. A Survey of Architectural Techniques for Improving Cache Power Efficiency. *Sustainable Computing: Informatics and Systems*, 4(1): 33–43, March 2014. ISSN 2210-5379. doi: 10.1016/j.suscom.2013.11.001. Available at: http://dx.doi.org/10.1016/j.suscom.2013.11.001.

[100] Sparsh Mittal and Jeffrey S. Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2): 19:1–19:23, August 2014. ISSN 0360-0300. doi: 10.1145/2636342. Available at: http://doi.acm.org/10.1145/2636342.

[101] Gordon E Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, January 1998. doi: 10.1109/jproc.1998.658762. Available at: http://dx.doi.org/10.1109/jproc.1998.658762.

[102] Trevor Mudge. Power: A First-Class Architectural Design Constraint. *IEEE Computer*, 34(4):52–58, April 2001. ISSN 0018-9162. doi: 10.1109/2.917539. Available at: http://dx.doi.org/10.1109/2.917539.

[103] Vu Thien Nga Nguyen. *An Efficient Execution Model for Reactive Stream Programs*. PhD thesis, University of Hertfordshire, 2014. Available at: http://hdl.handle.net/2299/16361.

[104] Vu Thien Nga Nguyen and Raimund Kirner. Demand-Based Scheduling Priorities for Performance Optimisation of Stream Programs on Parallel Platforms. In Joanna Kołodziej, Beniamino Martino, Domenico Talia, and Kaiqi Xiong, editors, *Algorithms and Architectures for Parallel Processing - 13th International Conference, ICA3PP 2013, Vietri sul Mare, Italy, December 18-20, 2013, Proceedings, Part I*, volume 8285 of *Lecture Notes in Computer Science*, pages 357–369. Springer International Publishing, 2013. ISBN 978-3-319-03858-2. doi: 10.1007/978-3-319-03859-9_31. Available at: http://dx.doi.org/10.1007/978-3-319-03859-9.

[105] National Institute of Standards and Technology. FIPS-46: Data Encryption Standard (DES), 1979. Available at: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf. Revised as FIPS 46-1:1988, FIPS 46-2:1993, FIPS 46-3:1999.

[106] Andreas Olofsson, Tomas Nordström, and Zain-ul-Abdin. Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany. In Michael B. Matthews, editor, *48th Asilomar Conference on Signals, Systems and Computers (ACSSC)*, pages 1719–1726, Washington, DC, USA, November 2014. Institute of Electrical & Electronics Engineers (IEEE). ISBN 978-1-4799-8297-4. doi: 10.1109/ACSSC.2014.7094761. Available at: http://dx.doi.org/10.1109/ACSSC.2014.7094761.

[107] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle: Do Nothing, Efficiently. In *Proceedings of the Linux Symposium*, volume 2, pages 119–125. Citeseer, 2007. Available at: http://www.scs.stanford.edu/~abelay/pdf/ols2007v2.pdf.

[108] Ritesh Parikh, Reetuparna Das, and Valeria Bertacco. Power-Aware NoCs Through Routing and Topology Reconfiguration. In *Proceedings of the 51st Annual Design Automation Conference*, DAC'14, pages 162:1–162:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2730-5. doi: 10.1145/2593069.2593187. Available at: http://doi.acm.org/10.1145/2593069.2593187.

[109] Trevor Pering, Tom Burd, and Robert Brodersen. Dynamic Voltage Scaling and The Design of a Low-Power Microprocessor System. In *Proceedings of the Power Driven Microarchitecture Workshop, attached to ISCA98*, pages 96–101, June 1998. Available at: https://www.researchgate.net/publication/2691157_Dynamic_Voltage_Scaling_and_the_Design_of_a_Low-Power_Microprocessor_System.

[110] Christian Poellabauer, Leo Singleton, and Karsten Schwan. Feedback-Based Dynamic Voltage and Frequency Scaling for Memory-Bound Real-Time Applications. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, RTAS '05, pages 234–243, Washington, DC, USA, March 2005. Institute of Electrical & Electronics Engineers (IEEE). ISBN 0-7695-2302-1. doi: 10.1109/RTAS.2005.23. Available at: http://dx.doi.org/10.1109/RTAS.2005.23.

[111] Thomas Prescher, Randolf Rotta, and Jörg Nolte. Flexible Sharing and Replication Mechanisms for Hybrid Memory Architectures. In *Proc. 4th Many-core Applications Research Community (MARC) Symposium. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam*, volume 55, pages 67–72, 2012. Available at: https://www.researchgate.net/publication/265219243_Flexible_Sharing_and_Replication_Mechanisms_for_Hybrid_Memory_Architectures.

[112] Daniel Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria, February 2010. Available at: https://www.researchgate.net/publication/262104326_A_Light-Weight_Parallel_Execution_Layer_for_Shared-Memory_Stream_Processing.

[113] James Reinders. An Overview of Programming for Intel ® Xeon ® Processors and Intel ® Xeon Phi ™ Coprocessors. *Intel Corporation, Santa Clara*, 2012. Available at: https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-\coprocessors_1.pdf.

[114] Randolf Rotta. On Efficient Message Passing on the Intel SCC. In Diana Göhringer, Michael Hübner, and Jürgen Becker, editors, *Proceedings of the 3rd Many-core Applications Research Community (MARC) Symposium*, volume 7598, pages 53–58. KIT Scientific Publishing, Karlsruhe, 2011. ISBN 978-3-86644-717-2. doi: 10.5445/KSP/1000023937. Available at: http://doi.org/10.5445/KSP/1000023937.

[115] Barry Rountree, David K. Lowenthal, Shelby Funk, Vincent W. Freeh, Bronis R. de Supinski, and Martin Schulz. Bounding Energy Consumption in Large-scale MPI Programs. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 1–9, New York, NY, USA, November 2007. ACM. ISBN 978-1-59593-764-3. doi: 10.1145/1362622.1362688. Available at: http://doi.acm.org/10.1145/1362622.1362688.

[116] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542340. Available at: http://doi.acm.org/10.1145/1542275.1542340.

[117] Praveen Salihundam, Shailendra Jain, Tiju Jacob, Shasi Kumar, Vasantha Erraguntla, Yatin Hoskote, Sriram R. Vangal, Gregory Ruhl, and Nitin Borkar. A 2 Tb/s 6$\times$4 Mesh Network for a Single-chip Cloud Computer With DVFS in 45 nm CMOS. *IEEE Journal of Solid-State Circuits*, 46(4):757–766, April 2011. ISSN 0018-9200. doi: 10.1109/JSSC.2011.2108121. Available at: http://dx.doi.org/10.1109/JSSC.2011.2108121.

[118] Tanguy Sassolas, Nicolas Ventroux, Nassima Boudouani, and Guillaume Blanc. A Power-aware Online Scheduling Algorithm for Streaming Applications in Embedded MPSoC. In René van Leuken and Gilles Sicard, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation: 20th International Workshop, PATMOS 2010, Grenoble, France, September 7-10, 2010, Revised Selected Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-17752-1. doi: 10.1007/978-3-642-17752-1_1. Available at: http://dx.doi.org/10.1007/978-3-642-17752-1_1.

[119] Bryan Schauer. Multicore Processors–A Necessity. *ProQuest Discovery Guides*, pages 1–14, 2008. Available at: http://www.engr.siu.edu/staff1/botrosn/mywebpage/Multicore.pdf.

[120] Jayanth Srinivasan. An Overview of Static Power Dissipation. *CiteSeer public search engine and digital libraries for scientific and academic papers in the fields of computer and information science*, pages 1–7, 2011. Available at: https://www.researchgate.net/publication/2406179_An_Overview_of_Static_Power_Dissipation.

[121] Mircea R. Stan and Wayne P. Burleson. Bus-Invert Coding for Low-Power I/O. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(1): 49–58, March 1995. ISSN 1063-8210. doi: 10.1109/92.365453. Available at: http://dx.doi.org/10.1109/92.365453.

[122] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990. doi: 10.1109/2.55497. Available at: http://dx.doi.org/10.1109/2.55497.

[123] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7): 491–541, July 1997. doi: 10.1007/s002360050095. Available at: http://dx.doi.org/10.1007/s002360050095.

[124] ClearSpeed Technology. CSX Processors Architecture, White Paper, 2007. Available at: https://www.cct.lsu.edu/~scheinin/Parallel/ClearSpeed_Architecture_Whitepaper_Feb07v2. [Online, accessed 20-Jan-2016].

[125] ClearSpeed Technology. CSX700 Floating Point Processor Datasheet, January 2011. Available at: http://www.clearspeed.com/products/documents/CSX700_Datasheet_Rev1E.pdf. [Online, accessed 20-Jan-2016].

[126] ClearSpeed Technology. CSX700 64-bit Floating Point Array Processor, January 2013. Available at: http://www.clearspeed.com/products/documents/csx700_product_brief.pdf. [Online, accessed 20-Jan-2016].

[127] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4. Available at: http://dl.acm.org/citation.cfm?id=647478.727935.

[128] William Frederick Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2009. Available at: http://hdl.handle.net/1721.1/46793.

[129] Osman S Unsal and Israel Koren. System-Level Power-Aware Design Techniques in Real-Time Systems. *Proceedings of the IEEE*, 91(7):1055–1069, July 2003. doi: 10.1109/JPROC.2003.814617. Available at: http://dx.doi.org/10.1109/JPROC.2003.814617.

[130] Giorgio Luigi Valentini, Walter Lassonde, Samee Ullah Khan, Nasro Min-Allah, Sajjad A. Madani, Juan Li, Limin Zhang, Lizhe Wang, Nasir Ghani, Joanna Kolodziej, Hongxiang Li, Albert Y. Zomaya, Cheng-Zhong Xu, Pavan Balaji, Abhinav Vishnu, Fredric Pinel, Johnatan E. Pecero, Dzmitry Kliazovich, and Pascal Bouvry. An Overview of Energy Efficiency Techniques in Cluster Computing Systems. *Cluster Computing*, 16(1):3–15, 2011. ISSN 1573-7543. doi: 10.1007/s10586-011-0171-x. Available at: http://dx.doi.org/10.1007/s10586-011-0171-x.

[131] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight Communications on Intel's Single-chip Cloud Computer Processor. *SIGOPS Operating Systems Review*, 45:73–83, February 2011. ISSN 0163-5980. doi:

10.1145/1945023.1945033. Available at: http://doi.acm.org/10.1145/1945023.1945033.

[132] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P. Rendell. Programming the Adapteva Epiphany 64-core Network-on-chip Coprocessor. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 984–992, Washington, DC, USA, May 2014. Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/IPDPSW.2014.112. Available at: http://dx.doi.org/10.1109/IPDPSW.2014.112.

[133] Voon-Yee Vee and Wen-Jing Hsu. A Scalable and Efficient Storage Allocator on Shared-Memory Multiprocessors. In *Proceedings of Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN '99)*, pages 230–235, Washington, DC, USA, 1999. Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/ISPAN.1999.778944. Available at: http://dx.doi.org/10.1109/ISPAN.1999.778944.

[134] Vasanth Venkatachalam and Michael Franz. Power Reduction Techniques for Microprocessor Systems. *ACM Computing Surveys (CSUR)*, 37(3):195–237, September 2005. ISSN 0360-0300. doi: 10.1145/1108956.1108957. Available at: http://doi.acm.org/10.1145/1108956.1108957.

[135] Merijn Verstraaten. High-level Programming of the Single-chip Cloud Computer with S-Net. Master's thesis, University of Amsterdam, 2012.

[136] Merijn Verstraaten, Clemens Grelck, Michiel W. van Tol, Roy Bakker, and Chris R. Jesshope. On Mapping Distributed S-Net to the 48-core Intel SCC Processor. In Diana Göhringer, Michael Hübner, and Jürgen Becker, editors, *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*, pages 41–46. KIT Scientific Publishing, Karlsruhe, 2011. ISBN 978-3-86644-717-2. Available at: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937.

[137] Leping Wang and Ying Lu. An Efficient Threshold-Based Power Management Mechanism for Heterogeneous Soft Real-Time Clusters. In *2008 Real-Time Systems Symposium*, pages 323–332, Washington, DC, USA, November 2008. Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/RTSS.2008.31. Available at: http://dx.doi.org/10.1109/RTSS.2008.31.

[138] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, September 2007. ISSN 0272-1732. doi: 10.1109/MM.2007.89. Available at: http://dx.doi.org/10.1109/MM.2007.89.

[139] Paul G. Whiting and Robert S. V. Pascoe. A History of Data-Flow Languages. *IEEE Annals of the History of Computing*, 16(4):38–59, December 1994. ISSN 1058-6180. doi: 10.1109/85.329757. Available at: http://dx.doi.org/10.1109/85.329757.

[140] Xiaocheng Zhou, Hu Chen, Sai Luo, Ying Gao, Shoumeng Yan, Wei Liu, Brian Lewis, and Bratin Saha. A Case for Software Managed Coherence in Manycore Processors. In *Proc. HotPar '10 (2nd USENIX Workshop on*

Hot Topics in Parallelism), Berkeley, CA, June 2010. Usenix Assoc. Available at: https://www.researchgate.net/publication/229043708_A_Case_for_Software_Managed_Coherence_in_Many-core_Processors.

[141] Dakai Zhu and Hakan Aydin. Reliability-Aware Energy Management for Periodic Real-Time Tasks. *IEEE Transactions on Computers*, 58(10):1382–1397, October 2009. doi: 10.1109/TC.2009.56. Available at: http://dx.doi.org/10.1109/TC.2009.56.