High Performance and Energy Optimal Parallel Programming on CPU and DSP based MPSoC

A Project Report

submitted by

ABHIPRAYAH TIWARI

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY, DELHI

JULY 2018

THESIS CERTIFICATE

This is to certify that the thesis titled "**High Performance and Energy Optimal Parallel Programming on CPU and DSP based MPSoC**", submitted by **Abhiprayah Tiwari**, to the Indraprastha Institute of Information Technology Delhi, for the award of the degree of Master of Technology, is an original research work carried out by him under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

Dr. Vivek Kumar
Supervisor
Assistant Professor
Dept. of Computer Science
IIIT-Delhi

Dr. Gaurav Mitra
Co-supervisor
Software Engineer
Texas Instruments
Sugar Land, Texas, USA

Place: New Delhi

Date: 27th July 2018

ACKNOWLEDGEMENTS

First and foremost, I offer my sincerest gratitude to my supervisor, Dr. Vivek Kumar and my co-supervisor Dr. Gaurav Mitra. This thesis would not have been possible without their continued support, patience, valuable suggestions and guidance. A special thanks to Anish Varghese for providing me access to Keystone II Hawking MPSoC at The Australian National University on which I carried out energy measurement related experiments. Finally, I would like to thank Texas Instruments for donating us a Keystone II Hawking MPSoC on which remainder of this work has been carried out.

ABSTRACT

With energy efficiency becoming a major concern in the HPC community, low-power alternatives such as heterogeneous Multi-Processor System-on-chip (MPSoC) are gaining popularity. These devices house several different kinds of accelerators on-chip including Digital Signal Processors (DSP) alongside CPU and GPU cores. Maximum performance from MPSoC devices can be achieved through the simultaneous use of all the processing elements. In addition to partitioning work across different processing elements being notoriously difficult on MPSoC platforms, achieving an energy-optimal partition is known to be non-trivial.

The Keystone II Hawking (K2H) Platform is one such MPSoC from Texas Instruments that consists of 4 cache coherent ARM cores and 8 TI DSP cores that are not cache coherent. This MPSoC provides high floating-point performance with low power consumption and is also being used in the nCore BrownDwarf supercomputer. Recent work on the K2H MPSoC describes a novel hybrid work-stealing runtime that supports concurrent execution of computation across all ARM and DSP cores. While this runtime scales well with loop-based parallelism, it offers limited scalability for recursive divide-and-conquer parallelism. Other recent work on K2H describes novel techniques to predict an energy-optimal work partition through the use of a simple energy usage model. However, this requires programmer effort in work partitioning and applying this energy model. This thesis aims to combine these two approaches while overcoming their inherent limitations.

The key contributions of my thesis are: 1) an implementation of a novel hybrid work-stealing runtime for the K2H MPSoC that supports high-performance execution of both loop-based and recursive divide-and-conquer parallelism; 2) a novel automated approach that uses this hybrid runtime for determining the energy-optimal partition; and 3) experimental analysis of the above claims by using several well-known benchmarks.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS **i**

ABSTRACT **ii**

LIST OF TABLES **vi**

LIST OF FIGURES **vii**

ABBREVIATIONS **viii**

1 INTRODUCTION **1**

1.1 Problem Statement 1

1.2 Scope and Contributions 2

1.3 Thesis Outline 3

2 BACKGROUND **5**

2.1 Work Stealing 5

2.2 TI-Keystone II SoC 7

2.3 *async-finish* Programming Model 9

2.4 HC-K2H 10

2.5 Energy Optimal Partitioning 11

2.6 Summary 13

3 EXPERIMENTAL METHODOLOGY **14**

3.1 Benchmarks 14

3.1.1 Parallel *for* Loop Benchmarks 14

3.1.2 Nested *async-finish* Benchmarks 16

3.1.3 Changes for avoiding False Sharing 17

3.1.4 Serial Versions of Benchmarks 17

3.2 Software Platform and Compiler Flags 17

3.3 Measurements 18

4 HIGH PERFORMANCE EXECUTION OF DIVIDE-AND-CONQUER APPLICATIONS 19

4.1 Introduction 19
4.2 Motivation 20
4.3 Insights 22
4.4 Implementation 22
4.4.1 Types of Deque 23
4.4.2 Main loop 23
4.4.3 Starting a finish scope 24
4.4.4 Asynchronous task creation 25
4.4.5 Ending a finish scope 26
4.4.6 Giving Tasks to free workers 28
4.4.7 Stealing and executing tasks all the tasks 30
4.4.8 Manual cache flushes for flat-finish based parallel *for* loops . 34
4.5 Experimental Evaluation 35
4.5.1 Speedup of U-RT over HC-K2H by using all ARM and DSP cores 35
4.5.2 Speedup over sequential execution 36
4.6 Summary 42

5 PRODUCTIVELY DETERMINING ENERGY-OPTIMAL PLACE OF EXECUTION 43

5.1 Introduction 43
5.2 Motivation 44
5.3 Insights 44
5.4 Implementation 45
5.4.1 Energy Model 45
5.4.2 Determining energy-optimal place of execution by using U-RT 46
5.4.3 Basic Helper Routines 47
5.4.4 Pre-tuning: Estimating Execution Times 48
5.4.5 Tuning Power Draw 50
5.4.6 Tuning Rate of Computations 51
5.5 Experimental Evaluation 51
5.6 Summary 55

6 RELATED WORK **56**
6.1 Parallel Programming Models for Heterogeneous Processors 56
6.2 Energy Efficiency on Heterogeneous Processors 57

7 CONCLUSION AND FUTURE WORK **58**
7.1 Conclusion 58
7.2 Future Work 59

A SOFTWARE INSTALLATION ON TI KEYSTONE-II **61**
A.1 Install MCSDK 61
A.2 Install MCSDK-HPC 61
A.3 Install TFTP server 61
A.4 Install NFS Server 62
A.5 EVM Setup 62
A.5.1 Basic setup of EVM 62
A.5.2 Install Terminal Software on PC for Serial Port Connection . 62
A.5.3 Bring up Linux on EVM using an NFS filesystem 63
A.6 Software Installation on File System of the EVM 64
A.6.1 Install HPC IPKs and CGTools on to the EVM 64
A.6.2 Install cross compilation tools 64
A.6.3 Setup HPC environment 65
A.6.4 Adding newer CMEM API 65

LIST OF TABLES

4.1 Speedup of ARM+DSP (12 workers) based execution over ARM only (4 workers) execution in HC-K2H 21

4.2 Speedup of U-RT relative to HC-K2H by using all ARM and DSP cores for nested `async−finish` parallelism 35

5.1 Time Taken during ARM, DSP and ARM+DSP execution 52

5.2 Energy consumed during ARM, DSP, ARM+DSP execution 52

5.3 Table comparing values computed using FORASYNC with values approximated by FORASYNC_ENERGY during profiling 54

5.4 Accuracy and Overhead Comparison of FORASYNC_ENERGY . . 55

LIST OF FIGURES

2.1 Work-Stealing Implementation 6

2.2 Keystone-II 66AK2H ARM-DSP SoC 8

2.3 An example code schema with `async` and `finish` 9

4.1 The communication structure of U-RT 28

4.2 Speedup obtained in U-RT relative to HC-K2H when all available ARM and DSP workers are used 36

4.3 Speedup obtained by using U-RT relative to sequential ARM execution 37

4.4 (Cont.) Speedup obtained by using U-RT relative to sequential ARM execution 40

4.4 Speedup obtained by using U-RT relative to sequential ARM execution 41

ABBREVIATIONS

SoC	System on a Chip
MPSoC	Multi-processor System on a Chip
TI	Texas Instruments
CPU	Central Processing Unit
DSP	Digital Signal Processor
FLOPS	Floating Point Operations per Second
GFLOPS	GigaFLOPS
SPP	Special Purpose Processor

CHAPTER 1

INTRODUCTION

This thesis addresses the problem of productively achieving high performance execution and automated discovery of energy-optimal place of execution on an MPSoC.

1.1 Problem Statement

Energy efficiency in High Performance Computing (HPC) has become an important factor, leading to complex heterogeneous platforms which include specialized devices such as GPU, FPGA, and DSP. The use of specialized accelerators in HPC systems has grown rapidly after processors hit the frequency barrier in early 2005. Recently, HPC systems have started using Multiprocessor System-on-Chip (MPSoC) as they house a variety of processing elements on a single chip. Supercomputers in this category are China's Tianhe-2A (top500.org, 2017), Hewlett Packard Moonshot (Packard, 2014) and nCore BrownDwarf (nCore HPC, 2015). MPSoC on these recent supercomputers comprises of multicore DSPs as accelerators. MPSoC offer significantly better power efficiency, high bandwidth and low latency data communication across all heterogeneous processing elements of the MPSoC. However, such devices have certain drawbacks, e.g., use of complex memory hierarchy, low memory constraints, etc, which make writing parallel programs for these devices a difficult task.

With the advent of multicore processors, variants of async-finish programming model (Charles *et al.*, 2005; Cavé *et al.*, 2011; Frigo *et al.*, 1998) have gained a lot of popularity and are extremely well suited for high productivity parallel programming over multicore processors. In this programming model, programmers simply expose a parallel task by using an `async` keyword and task synchronization points by using a `finish` keyword. The problem of scheduling these async tasks is then delegated to an underlying work-stealing runtime (Blumofe and Leiserson, 1999). Kumar *et al.* (2015) proposed such an `async–finish` style parallel programming model for the Keystone-II Hawking MPSoC (K2H) MPSoC (Section 2.4). This MPSoC is

composed of 4 cache coherent ARM cores and 8 DSP cores that are not cache coherent. They implemented an underlying hybrid work-stealing runtime that abstracts away the hardware complexities from the user and performs high performance concurrent execution of async tasks across all ARM and DSP cores. While their runtime scales well with loop-based parallel program, it offers limited scalability for recursive divide-and-conquer parallelism where newly spawned child task should join with its parent post-termination.

Further, to achieve *optimal* energy usage on MPSoC, it is important to partition the computation among heterogeneous processors on MPSoC such that *energy-to-solution* is the main criteria rather than the *time-to-solution*. Mitra *et al.* (2016) recently proposed an energy usage model for heterogeneous systems which can be used to calculate and predict an energy-optimal work partition based solely on system characteristics and the computational rate of the application on each isolated device on MPSoC. However, this work lacks a hybrid load-balancing runtime and the programmer has to manually perform work-partitioning for determining the energy-optimal partition, hence hampering the programmer's productivity.

Thus, despite the growing popularity of MPSoCs, developing parallel applications based on power or performance goals on MPSoCs is still very challenging.

1.2 Scope and Contributions

The aim of my research is to mitigate the two major challenges for productively exploiting the parallelism provided by modern MPSoC processors— achieving high performance execution for divide-and-conquer applications and determining energy-optimal place of execution for parallel applications.

To do this, I chose `async–finish` approach as the representative of modern parallel programming models for multicore processors and TI's Keystone-II Hawking platform as the MPSoC processor. To evaluate hybrid work-stealing runtime, I chose the approach proposed by Kumar *et al.* (2015) and for determining the energy optimal place of execution, I chose the approach proposed by Mitra *et al.* (2016). Irrespective of these choices, the methodology and insights developed here should be applicable to other MPSoCs consisting of multicore CPUs and DSPs.

High performance execution of divide-and-conquer applications Traditionally it is harder to achieve both productivity and performance in divide-and-conquer parallel programming than for `for` loop based parallelism. However, with the advent of `async–finish` programming model with an underlying work-stealing runtime, it is now easy to achieve both productivity and performance in divide-and-conquer parallel programming as well. While it is easy to implement a work-stealing runtime for homogeneous multicore processors, implementing the same for MPSoCs is challenging due to the presence of heterogeneous processing elements. This thesis identifies the issues in performing hybrid work-stealing over heterogeneous processing elements in MPSoC for an `async–finish` based divide-and-conquer parallelism. We introduce a novel design of a hybrid work-stealing runtime for Keystone-II Hawking MPSoC that supports high performance execution of both divide-and-conquer and `for` loop parallelism.

Productively determining energy-optimal place of execution Performance and energy-optimal scheduling are two different goals (Mitra *et al.*, 2016). Depending upon the need, a user may require either of these objectives to be optimized. However, it is a daunting task to productively determine the energy-optimal work-partitioning for MPSoC. This thesis addresses this issue for loop based parallelism by introducing a novel approach that relies on an underlying hybrid work-stealing runtime to automatically determine the energy-optimal work-partition for Keystone-II Hawking MPSoC, i.e., the best place for achieving energy-optimality (all ARM cores, all DSP cores, or all ARM+DSP cores).

1.3 Thesis Outline

The body of this thesis is structured around the two key contributions outlined above.

Chapter 2 gives a brief overview of work-stealing runtime, the Keystone-II Hawking MPSoC, the `async–finish` programming model, the work by Kumar *et al.* (2015) and Mitra *et al.* (2016) on which this work is based.

Chapter 4 and 5 comprise the main body of the thesis, covering the two key contributions. Chapter 4 discusses the issues associated with divide-and-conquer parallel programming over the Keystone-II Hawking MPSoC and describes the implementation

and evaluation of our novel U-RT work-stealing runtime that supports high performance execution for both divide-and-conquer and `for` loop based parallel programming over Keystone-II Hawking MPSoC. Chapter 5 uses U-RT and presents a novel approach for determining energy-optimal place of execution for loop based parallelism.

Finally, chapter 7 concludes the thesis and provides directions for future research.

CHAPTER 2

BACKGROUND

This chapter provides relevant background information. The chapter starts with a brief discussion on work-stealing scheduling algorithm in Section 2.1. Section 2.2 provides an overview of Keystone-II Hawking MPSoC. Section 2.3 discusses the `async–finish` programming model. Section 2.4 discusses the HC-K2H from Kumar *et al.* (2015) and finally, section 2.5 concludes this chapter by discussing the energy optimal partitioning scheme from Mitra *et al.* (2016).

2.1 Work Stealing

Work-stealing (Blumofe and Leiserson, 1999) is a strategy for efficiently distributing work in a parallel system. The runtime maintains a pool of *worker threads*, each of which maintains a local set of *tasks*. When local work runs out, the worker becomes a *thief* and searches for a *victim* thread from which to *steal* work. A steal occurs when a thief takes work from a victim. The runtime provides the thief with the execution context of the stolen work, including the entry point and sufficient program state to initiate the computation. The runtime ensures that work is executed exactly once and that the state of the program reflects the contributions of all workers.

Work-stealing can be implemented using concurrent deque or private deque. Chase-Lev (Chase and Lev, 2005) concurrent work-stealing deque is the standard for concurrent deque work-stealing schedulers. Figure 2.1 shows the general implementation of a work-stealing algorithm. *Push* and *pop* are usually performed from the *head* of the deque, i.e., in LIFO (Last In First Out) order, whereas steals are performed from the tail, i.e., in FIFO (First In First Out) order. In this design, recently created tasks are executed by the victim whereas the oldest created tasks are stolen by the thief, thereby improving the locality. Moreover, in the case of divide-and-conquer applications, tasks found on the tail of deque (older tasks) will be of bigger computation than the one found at the

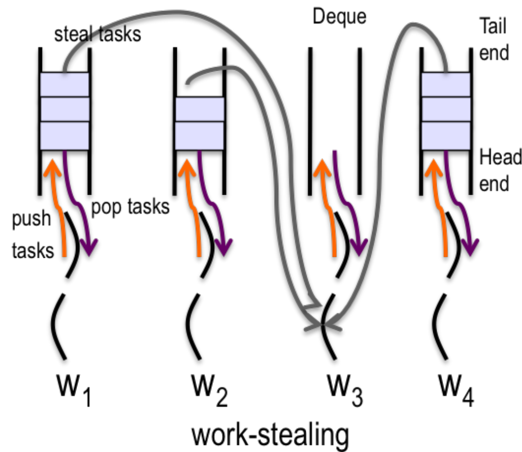


Figure 2.1: Work-Stealing Implementation

head. Hence, any steal will migrate a big chunk of computation to the thief. This will also lead to infrequent steal attempts from the thief.

Work-stealing schedulers can efficiently schedule both regular and irregular computations. This makes them a good choice for heterogeneous systems also as even though the computation may originally be regular, the differences in the architecture of different processors would give it the characteristic of an irregular computation. Randomized work-stealing schedulers come with guaranteed bounds on the expected execution time(Blumofe and Leiserson, 1999) and maintain good locality for recursively parallel computations.

On weak memory models, concurrent deques require expensive memory-fences, which can degrade performance significantly. E.g., Frigo *et al.* (1998) found that for the work-stealing implementation in Cilk, half of the time is spent in executing memory fence instructions. Due to these limitations, there has been a lot of interest in work-stealing algorithms where non-concurrent, *private deques* replace the concurrent, shared deques, and processors explicitly communicate to balance load (Acar *et al.*, 2013). In such algorithms, each processor keeps its deque private, operating on its bottom end as usual. When a processor finds its deque to be empty, instead of manipulating a remote deque concurrently, it sends a message to a randomly chosen victim processor and waits for a response, which either includes one or more tasks or indicates that the victim's deque is empty. In order to respond to messages, each processor periodically polls, often driven by interrupts, its message queue. U-RT runtime discussed in this thesis utilizes private deques for improving the performance of recursive divide-and-

conquer algorithms on the K2H (Chapter 4).

2.2 TI-Keystone II SoC

Texas Instrument's KeyStone platforms provide an innovative platform that integrates RISC and DSP cores with application-specific co-processors and input/output peripherals. This design can achieve excellent performance for embedded DSP applications, but can also be used for general-purpose applications.

The work described in this paper was developed on TI's Keystone II platform (also referred to as Hawking), but the principles should be applicable to other CPU+SPP architectures. The Hawking platform contains 4 ARM Cortex-A15 processors, each running at up to 1.4 GHz and 8 TI C66x floating-point DSPs each running at up to 1.2GHz, all on a single low-power System-on-chip (SoC). The hardware also provides hardware queues, which can be used to communicate between ARM and DSP cores. There are two queue managers with 8192 queues per queue manager and 64 descriptor memory regions per queue manager. The hardware queues provide synchronized push operation from both ends of the queue, but the pop operation can only take place at the tail of the queue. Therefore, they differ from standard work-stealing deques which require synchronized pop operations on both sides and push operation on one side.

The ARM cores have 32KB of L1 cache each and 4MB of shared L2 cache while DSP cores can have up to 32KB of L1 cache and 1MB of L2 cache each. The caching hardware provides automatic cache-coherence between the ARM cores but the shared memory controller in Keystone devices neither maintains cache-coherency between two DSP cores nor between ARM and DSP cores. As such it is the responsibility of the software to maintain coherent views of memory when required. The L2 cache is inclusive of L1 cache on ARM side and thus all the data in the cache sub-system before the initiation of write-back of L2 cache is written to the main memory after the write-back has completed. Both DSP cores and ARM cores also have hardware pre-fetcher logic built into them. Thus, both processors can bring any data any time from the main memory into there cache system. DSPs provide a way to invalidate only the pre-fetched data, however, as far as we know ARM provides no special support for invalidating only the pre-fetched data. However, it is possible to turn off pre-fetching completely

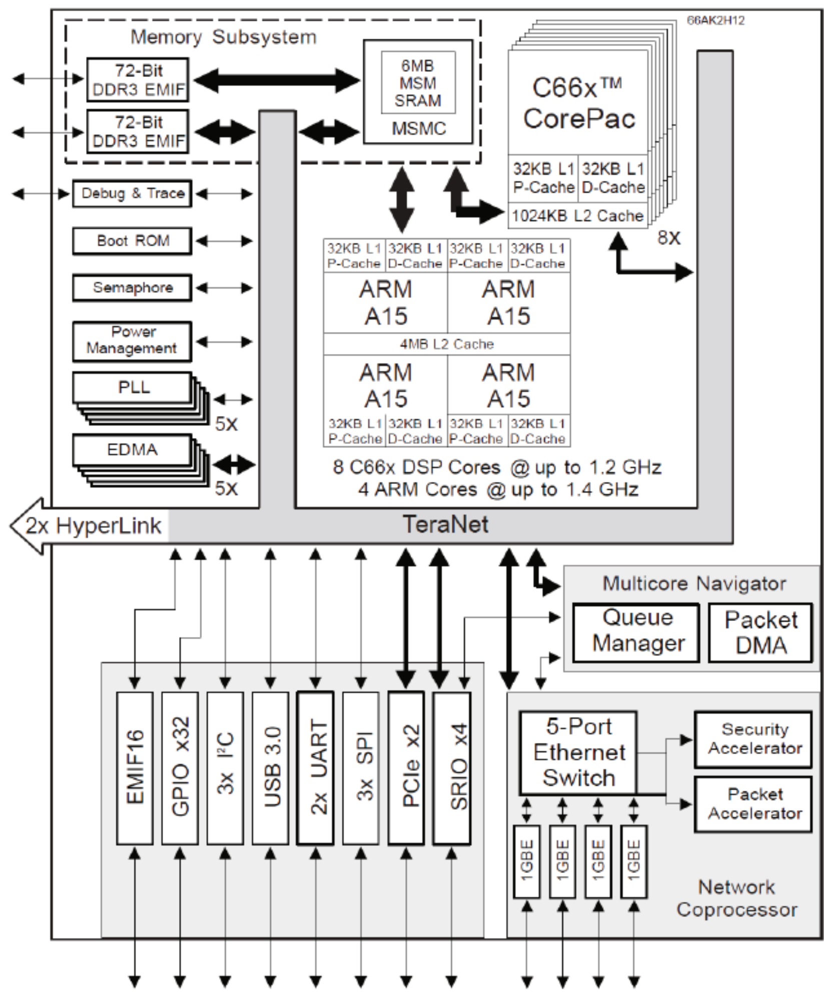


Figure 2.2: Keystone-II 66AK2H ARM-DSP SoC

on both the processors. The ARM cores only support write-back and write-back invalidate operations and main memory can be configured to be write-through, write-back or non-cache-able on a per-page basis (ARM Cortex A15 MPCore Processor Technical Reference Manual; TMS320C66x DSP CorePac User Guide; TMS320C66x DSP Cache User Guide).

2.3 *async-finish* Programming Model

`async-finish` constructs were first coined by X10 language Charles *et al.* (2005). Now it has been adopted by several other frameworks supporting task parallelism such as Habanero-Java (Imam and Sarkar, 2014; Cavé *et al.*, 2011), Habanero-C/C++ (Kumar *et al.*, 2014), and Java TryCatchWS (Kumar *et al.*, 2012). Some other well-known implementations that support variant of `async-finish` are Cilk (Blumofe and Leiserson, 1999) (`spawn-sync`) and Java `fork-join` framework (Lea, 2000).

Figure 2.3 shows a sample code written using `async-finish` programming model. The "`async` ⟨*stmt*⟩" clause creates a new task, which will execute ⟨*stmt*⟩ *asynchronously* (i.e., before, after, or in parallel) with the code which follows the `async` block (i.e., the remainder of the parent task). Figure 2.3 illustrates how the parent task, T_0, uses an `async` construct to create a child task T_1. Thus, `STMT1` in task T_1 and `STMT2` in task T_0 can potentially execute in parallel.

`async` is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including statement blocks, for-loop iterations, and function calls.

Finish is a generalized join operation. The statement `start_finish()` starts

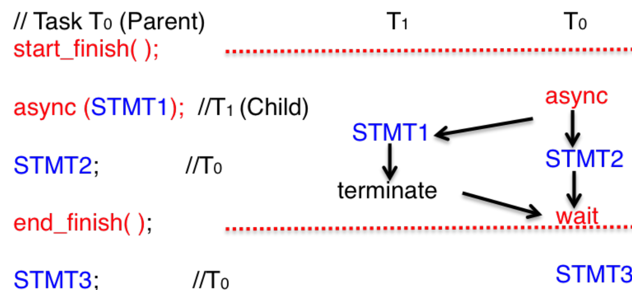


Figure 2.3: An example code schema with `async` and `finish`

a finish scope and the statement `end_finish` terminates this finish scope. Worker starting the finish scope will wait in the `end_finish` scope until all `async` transitively spawned tasks from this scope have completed. Each dynamic instance T_A of an `async` task has a unique *Immediately Enclosing Finish* (IEF) instance F of a finish statement during program execution, where F is the innermost finish containing T_A.

For example, the `start_finish` statement in Figure 2.3 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before `STMT3` starts executing. If T_1 were to create additional transitive tasks, these would become "grandchild" of T_0 and T_0 will wait for all to complete before executing `STMT3`. The power of these constructs comes from the ability to arbitrarily nest `async` and `finish` constructs.

2.4 HC-K2H

Kumar *et al.* (2015) presented a library-based implementation of `async–finish` programming model for the Keystone-II Hawking MPSoC that used an underlying hybrid work-stealing runtime for load balancing of tasks across ARM and DSP cores. HC-K2H is based over HClib. HC-K2H supports two constructs for parallel programming over the Keystone-II Hawking MPSoC: a) *forasync* construct for supporting `for` loop based parallelism, and b) `async–finish` for recursive divide-and-conquer parallelism. The tasks generated from these constructs are then scheduled over ARM and DSP cores by using a hybrid help-first work-stealing scheduler (Guo *et al.*, 2009). In this hybrid runtime, ARM cores (total 4) used a software deque that is based on THE protocol from Cilk language (Frigo *et al.*, 1998), whereas DSP cores used the hardware queue provided by Keystone-II Hawking MPSoC. Both ARM and DSP workers would first try an intra-arch steal before attempting an inter-arch steal. ARM workers could steal tasks from DSP workers through the hardware queue of DSP anytime, however, DSP processors could not steal directly from the ARM workers. ARM workers would keep checking if DSPs are idle and then put tasks to a shared hardware queue from where the DSP workers could steal. HC-K2H supported nesting of `finish` scopes, but the finish scopes were not shared across ARM and DSP cores. The reason for this decision was to allow each processor to use its own efficient synchronization mechanism to increment or decrement the current number of tasks available in the finish

scope. ARM workers used CAS atomic operations for counter updates whereas DSP workers used hardware semaphores. Whenever a task was transferred across architectures, a new *artificial finish scope* would get created and the task was executed inside that new finish scope. This means that this task and all its children would complete the execution before this finish scope had ended. After the completion of the finish scope, the ARM or DSP would message each other about task completion, so that the other processor could decrement its task counter. To facilitate the above, their work reserved two hardware queues for receiving the message when the other side had completed all the tasks in its artificial finish scope. The processors would check for the messages about task completion during their steal cycles.

The second problem that they had to face was to maintain coherent views of memory when executing the tasks across ARM and DSP cores. This is because DSP cores do not maintain cache coherency with other DSP cores or with the ARM processor (Section 2.2). To maintain a consistent memory view, they did a cache flush at the end of each finish scopes and artificial finish scopes on the ARM side and also during the execution of each task on DSP side.

2.5 Energy Optimal Partitioning

This section discusses the work presented in the paper by Mitra *et al.* (2016). Their work presented a model for predicting whether splitting the work across heterogeneous processors will be energy-optimal or not, along with a novel and cheap energy measurement framework. While this study presented means to predict energy-optimal partitioning, it required work partitioning of the application to be done by the programmer. Hence, programmer effort is required in judging out which work partitioning would be better for energy-optimal execution.

For their purposes, they configured their energy measurement framework to send a current reading every 10ms (this is set to 1ms for our purposes), which was later used for calculating the total energy consumed. The energy model was independent of the amount of work to be done and was only dependent on key system characteristics like power consumption of different processors and the ratios of the rate of computation of these processors. A brief description of their work from which our work in Chap-

ter 5 has been derived has been given below with slightly modified terminology for TI-Keystone II SoC.

Let the rate of computation of ARM be R_c and that of DSP be R_d. Let the power draw of ARM when it is executing be P_c^a and that of DSP be P_d^a. Let the power draw of ARM when it is idling be P_c^i and that of DSP be P_d^i. Let N be the total amount of work to be done and T_{ac} be the time spent by ARM in executing the computation and T_{ad} be the time spent by DSP in executing the computation. If f is the fraction of work to be given to ARM and T_s is the time to solution given by $max\left[T_{ac}, T_{ad}\right]$, then the total energy consumption can be given as,

$$E(f) = \left[\left(P_c^a - P_c^i\right)\frac{Nf}{R_c} + P_c^i T_s\right] + \left[\left(P_d^a - P_d^i\right)\frac{N(1-f)}{R_d} + P_d^i T_s\right] \tag{2.1}$$

The energy optimal work partition will be the value of f which minimizes equation 2.1. It's minimum must occur when $f = 0$ or when $f = 1$ or when $T_{ac} = T_{ag}$, i.e., when both ARM and DSP finish with no idle time, in which case f^* is given by $\frac{R_c}{R_c+R_d}$ assuming that both processors combined can give throughput which is equivalent to the sum of their individual throughputs. Then, using equation 2.1, points at which the minimum energy consumption may occur can be given as,

$$E(0) = N\left(\frac{P_d^a + P_c^i}{R_d}\right) \tag{2.2}$$

$$E(1) = N\left(\frac{P_d^i + P_c^a}{R_c}\right) \tag{2.3}$$

$$E(f^*) = N\left(\frac{P_c^a + P_d^a}{R_c + R_d}\right) \tag{2.4}$$

For f^* to be the minimum, we have $E(f^*) < E(0)$ and $E(f*) < E(1)$. Using equation 2.2, 2.3 and 2.4, we arrive at the following results,

$$\frac{P_c^a - P_c^i}{P_d^a + P_c^i} < \frac{R_c}{R_d} < \frac{P_c^a + P_d^i}{P_d^a - P_d^i} \tag{2.5}$$

Equation 2.5 presents a simple way to predict whether splitting the work would lead to lower energy consumption compared to executing it at only one processor or not and is

only dependent on power draw values of different processors and the ratio of their rates of computations.

2.6 Summary

This chapter introduces key background material. We provide an introduction to work-stealing, Keystone-II Hawking MPSoC, `async–finish` programming model, HC-K2H and energy optimal partitioning. Before we move to the heart of this thesis and its primary contributions, we first give an overview of our experimental methodology, in the next chapter.

CHAPTER 3

EXPERIMENTAL METHODOLOGY

This chapter describes the benchmarks, compilers, operating system, hardware, and performance measurement methodologies used in this thesis. To maintain coherency in experimental methodology across different chapters, a common set of benchmarks, workloads, and measurements is used for this thesis.

3.1 Benchmarks

For evaluating U-RT, two broad categories of benchmarks were required: a) regular parallel `for` loop based benchmarks and b) nested `async–finish` benchmarks. These benchmarks are as follows:

3.1.1 Parallel *for* Loop Benchmarks

BFS

BFS is a parallel implementation of BFS graph traversal algorithm. It has multiple iteration for the same parallel `for` loop but the computation reduces in each iteration. Input size of 4 million was used. Adopted from Rodinia benchmark suite (Che *et al.*, 2009).

B+Tree

B+Tree is a parallel tree search algorithm for searching point based and range based queries. Input size of 65535 was used. Adopted from Rodinia benchmark suite (Che *et al.*, 2009).

Hotspot

Hotspot is a structured grid physics simulation benchmark and the same parallel `for` loop is executed multiple times in it. The following options were passed to control size: (4096 4096 10 4 ../temp_4096 ../power_4096). Adopted from Rodinia benchmark suite (Che *et al.*, 2009).

LUD

LU decomposition is a dense linear algebra benchmark used for calculating the solution to a set of linear equations. The same parallel loops get called multiple times in this application but each time the amount of computation in the loop is reduced. Input size of 4096 was used. Adopted from Rodinia benchmark suite (Che *et al.*, 2009).

LavaMD

LavaMD is a coarse granular parallel loop benchmark which calculates the particle potential and relocation due to mutual forces between particles within a large 3D space. NUM_PAR_PER_BOX was set to 96 with the default input size. Adopted from Rodinia benchmark suite (Che *et al.*, 2009).

Particle Filter

Particle filter is a benchmark consisting of multiple small and big kernels, each of which run multiple times. The following options were passed to control size: (-x 128 -y 128 -z 10 -np 100000). Adopted from Rodinia benchmark suite (Che *et al.*, 2009).

SRAD

SRAD is a compute-intensive benchmark with a good mix of multiple types of arithmetic operations in the compute kernel. In this benchmark also, the same loop runs multiple times. Following command line options were used to run the benchmark: (4096 4096 0 127 0 127 2 0.5 2). Adopted from Rodinia benchmark suite (Che *et al.*, 2009).

Tiled Matmul

Tiled matrix multiplication is an implementation of matrix multiplication which uses tiling along with a parallel for loop to achieve high performance. Tiling increases the cache locality of the benchmark and makes it mostly compute intensive. The matrices of size 4096×4160 and 4160×4032 were multiplied. Adopted from HClib distribution (Kumar *et al.*, 2014).

3.1.2 Nested *async-finish* Benchmarks

Jacobi

Jacobi performs iterative mesh relaxation with barriers and the input size of matrices were 4096×4096 and the number of steps was 10. Based on an algorithm taken from the Java ForkJoin framework (Lea, 2000).

Matmul

Matmul implements a cache-oblivious matrix multiplication algorithm. Thus, it is not possible to increase the granularity of the benchmark arbitrarily as the performance could degrade with increase in granularity due to the sub-matrices not being able to fit in the cache. Likewise, too fine granularity will have a similar effect due to increased overheads. Both the matrices multiplied had 1024 rows and columns. Adopted from Cilk distribution (Frigo *et al.*, 1998).

Mergesort

Mergesort is a custom made simple mergesort algorithm and as such it has no parallel merge phase and suffers from lack of scalability. Size of the array was 16,777,216.

Cilksort

Cilksort is essentially a mergesort in which merging is also done in parallel. This allows the benchmark to scale well with much higher processing units compared to mergesort.

Size of the array was 16,777,216. Adopted from Cilk distribution (Frigo *et al.*, 1998).

3.1.3 Changes for avoiding False Sharing

On the Keystone-II Hawking MPSoC, to obtain correct results from the parallel execution across ARM and DSP cores, it is mandatory to ensure that the shared data accesses across ARM and DSP is free from false-sharing (Intel Software Developer Zone). On the Keystone-II Hawking MPSoC, the L1 cache line size is different for ARM and DSP (64 and 128 bytes respectively). DSPs are not cache coherent with ARM cores and hence explicit cache write-back by DSP runtime can easily corrupt the data shared with ARM if proper data layout is missing. For removing false sharing in parallel `for` loop benchmarks, we used either loop tiling or padding of shared data structures. In case of nested `async–finish` benchmarks, either the place of division of work was changed (if possible) or task granularity was changed. If nothing worked then padding was done.

3.1.4 Serial Versions of Benchmarks

The serial version of each benchmark is obtained either by removing all `async–finish` model based constructs from the benchmark or by having the only single (root) `async–finish` construct remaining. Thus, the serial version of each benchmark has no task creation or runtime overhead associated with it. Sometimes, this version has also been referred to as serial elision version in the thesis.

3.2 Software Platform and Compiler Flags

The latest available release of MCSDK-HPC (MCSDK-HPC) for the Keystone-II Hawking MPSoC was installed and all the experiments were performed on it. For ARM, Linaro ARM compiler, having gcc version 4.7 was used and for DSP optimizing compiler cl6x version 8.0.3 was used.

The compiler flags for ARM were: -O3 -mcpu=cortex-a15 -mfpu=vfpv4 -mfloat-abi=hard. NEON SIMD floating point extensions were not used as they were not fully IEEE compliant before ARMv8 (ARM Floating Point) and the Keystone-II Hawking

MPSoC has ARMv7a architecture processors.

The compiler flags for DSP were: –abi=eabi -mv6600 -op3 -ma –multithread -O3. Out of the 1MB L2 cache available to DSP, 512KB was configured as L2 cache and the rest was configured as SRAM.

3.3 Measurements

Parallel `for` loop benchmarks were executed for 10 iterations and execution time reported is an average of these 10 iterations. For energy related experiments average of 7 iterations are reported. Execution time reported for nested `async–finish` benchmarks is an average of 20 iterations for each benchmark.

CHAPTER 4

HIGH PERFORMANCE EXECUTION OF DIVIDE-AND-CONQUER APPLICATIONS

Recall from Section 2.1 that work-stealing is a very efficient strategy for distributing work in a parallel system. It has also been used for hybrid load balancing over heterogeneous processing elements in Keystone-II Hawking MPSoC (Section 2.4). However, nested `async–finish` parallelism on Keystone-II Hawking MPSoC incurs overheads as compared to `for` loop parallelism. This overhead is visible while using all heterogeneous processing elements in MPSoC. This chapter analyzes the source of these overheads and introduces the design of a novel hybrid work-stealing runtime that does not exhibit these overheads.

The rest of this chapter is structured as follows. Section 4.2 identifies the motivation for this chapter, Section 4.3 describes our insights in designing U-RT, a novel hybrid work-stealing runtime. Section 4.4 describes the design and implementation of U-RT. Section 4.5 describes the experimental evaluation of U-RT by using several well-known benchmarks, and finally section 4.6 concludes this chapter.

4.1 Introduction

`async–finish` based programming model is widely used for achieving high productivity and high performance parallel programming over multicore processors (Section 2.3). Kumar *et al.* (2015) presented an `async–finish` based parallel programming model for the Keystone-II Hawking MPSoC with an underlying hybrid work-stealing runtime (HC-K2H) capable of scheduling computation across both ARM and DSP cores, thereby abstracting away the hardware complexities from the programmers. While they were able to demonstrate good scalability for parallel loops using this runtime, the scalability was limited for recursive divide-and-conquer parallel programs while using all ARM and DSP cores. This chapter identifies the key source of overhead in HC-K2H that limited the scalability of recursive divide-and-conquer parallel

programs on all ARM and DSP cores of the Keystone-II Hawking MPSoC. It further discusses the design, implementation and evaluation of a novel runtime U-RT, that extends the ideas proposed in Acar *et al.* (2013) for performing hybrid work-stealing on the Keystone-II Hawking MPSoC.

The principal contributions in this chapter are as follows: a) a study of the sources of overhead in the implementation of HC-K2H; b) design and implementation of U-RT for the Keystone-II Hawking MPSoC that shows good scalability of both loop-based and divide-and-conquer parallelism; and c) a detailed experimental evaluation of U-RT by using several well-known benchmarks.

4.2 Motivation

Algorithm 4.1: Mergesort algorithm parallelized using `async–finish` model

```
1 procedure MERGESORT(a, p, r)
2   if p < r then
3     q ← (p + r) / 2
4     START_FINISH()
5     ASYNC(MERGESORT(a, p, q))
6     ASYNC(MERGESORT(a, q + 1, r))
7     END_FINISH()
8     MERGE(a, p, q, r)
```

Algorithm 4.1 shows the pseudo-code of mergesort algorithm parallelized using `async–finish` parallel programming model. Mergesort provides a perfect example of how a divide-and-conquer algorithm is parallelized using `async–finish` model. Most of the pseudocode is similar to original mergesort pseudocode and in fact, removing ASYNC, START_FINISH and END_FINISH calls would give us a serial version of mergesort. In lines 5 and 6, we identify tasks which can be executed in parallel with other tasks. Lines 4 and 7 identify the finish scope of the tasks, where START_FINISH starts the finish scope and END_FINISH ends the finish scope. All the tasks and their sub-tasks generated in the finish scope and its children finish scopes must complete execution before the control reaches out of END_FINISH. In simple words, a runtime capable of handling nested `async–finish` parallelism should be capable of providing the results inside the finish scope, to the computation which follows the finish scope. While a simple global atomic counter would have been sufficient in the case of a `for`

loop, a counter local to each finish scope is required for mergesort as a tree of finish scopes is created and synchronization has to take place at each join point (Section 2.3).

For cache coherent multicore processors, a simple check whether all pending tasks generated within the START_FINISH and END_FINISH scope have completed is sufficient to transfer the control after END_FINISH. All results calculated by `async` tasks within this `finish` scope can be safely accessed by statements after END_FINISH. However, this approach will not work for MPSoCs such as Keystone-II Hawking MPSoC because it does not have hardware support for maintaining cache-coherency across all processors.

Recall from Section 2.2 that DSPs in the Keystone-II Hawking MPSoC don't support concurrent software queues that could efficiently and safely be accessed directly by any of the other DSPs. Instead, the Keystone-II Hawking MPSoC supports a fixed number of hardware queues that could concurrently be accessed by any of the DSP or ARM cores. HC-K2H runtime used these hardware queues (one per DSP) to store tasks at a victim DSP (Section 2.4). These tasks can directly be stolen by any ARM or DSP thieves, i.e., some of the `async` tasks generated within a `finish` scope can execute at cache coherent ARM cores while some of them could execute at cache incoherent DSP cores. Hence, to ensure that results computed by `async` tasks within a `finish` scope has visibility after END_FINISH, HC-K2H runtime has to explicitly perform cache write back and invalidation operation at two places: a) at a DSP worker before and after it has completed the execution of an `async` task, and b) on ARM worker during the execution of each and every END_FINISH.

Table 4.1: Speedup of ARM+DSP (12 workers) based execution over ARM only (4 workers) execution in HC-K2H

Benchmark Name	Speedup Relative to 4 ARM Workers
Mergesort	0.837
Jacobi	0.055
Matmul	2.540
Cilk Sort	0.913

Although this approach allowed the concurrent execution of recursive divide-and-conquer computations across all ARM and DSP cores, it adversely affected the parallel performance and the same is reflected in Figure 4.1. As we can observe, concurrent

execution of Jacobi across all ARM and DSP cores ended up being almost $19\times$ slower compared to ARM only execution. In fact, other than Matmul, none of the benchmarks are showing any benefit of using all ARM and DSP cores for parallel execution.

Another limitation of using hardware queues in HC-K2H is that they can only be configured as FIFO-only or LIFO-only queue, unlike the software deques at ARM that allow LIFO policy at one end (for *pop*) while FIFO policy at the other end of the deque (for *steal*). In FIFO-only configuration of hardware queues, *pop* operation would significantly increase the memory requirement as it would be similar to doing a breadth-first search compared to a depth-first search. In LIFO-only configuration, *steals* would fetch smaller computation to the thief and would result in frequent stealing.

4.3 Insights

The insight we use is that a cache write back and invalidation (cache flush) operation is not required at the end of each *end_finish* API execution at ARM cores and also after every `async` execution at DSP cores. Instead, the cache flush should only be done when stealing happens between two cache-incoherent processors. The goal then becomes to minimize the steals happening during the entire execution of a parallel program. As explained in section 2.1 this is already achieved at ARM cores by using work-stealing software deques in HC-K2H. However, the same cannot be achieved by using hardware queues at DSP side due to the limitation explained in previous section. To overcome this limitation and to implement cache-coherency efficiently through the runtime, we rely on private software deque implementation (Acar *et al.*, 2013) at DSP cores.

4.4 Implementation

In this section, we will discuss in detail the design of U-RT. For the sake of clarity and for a fair comparison, the design of HC-K2H is also presented along with U-RT. There are differences between ARM side and DSP side implementation on both the runtimes for achieving the same functionality, thus to make comprehending the algorithms easier we adopt the following convention for writing the algorithms:

1. **`if`** *ARM* indicates that the code in the body of the **`if`** condition is for the ARM side of the runtime.

2. **`if`** *DSP* indicates that the code in the body of the **`if`** condition is for the DSP side of the runtime.

3. **`if`** *U-RT* indicates that the code in the body of the **`if`** condition is for the new runtime, i.e. U-RT.

4. **`if`** *HC-K2H* indicates that the code in the body of the **`if`** condition is for the old runtime, i.e. HC-K2H.

4.4.1 Types of Deque

U-RT has three different kinds of data structures to store `async` tasks for achieving the goal of high performance execution of both nested `async–finish` and `for` loop () parallelism: a) similar to HC-K2H, each ARM core maintains a work-stealing software deque for high performance load balancing across ARM cores, b) similar to HC-K2H, shared hardware queues for managing all inter-arch steals and communications, and c) unlike HC-K2H, private deque implemented on fast-shared memory for DSP cores (L2 SRAM and MSMC memory).

4.4.2 Main loop

Algorithm 4.2: Main loop of workers

```
1 procedure WORKER_ROUTINE
2   while shutdown ≠ true do
3     FIND_AND_EXECUTE_TASK()
```

Algorithm 4.2 describes the main loop of workers. Except for the master worker, each worker directly executes this loop once the program is launched. Master worker will execute this loop within the END_FINISH if there are pending tasks in the immediate `finish` scope. Succinctly put, in this routine the workers simply poll for getting a task and then execute it. This is the skeleton code for most of the dynamic load-balancing runtimes. The key to determining the efficiency of the runtime lies in the FIND_AND_EXECUTE_TASK procedure, whose implementation we discuss later in section 4.4.7.

Rest of the chapter is organized to make understanding the flow of the runtime easier as it follows the order in which each of these constructs will be encountered in parallelized mergesort (Algorithm 4.1). Algorithm 4.3 represents a generalized nested

Algorithm 4.3: Possible structure of nested `async−finish` parallelism

```
1  procedure COMPUTE
2    SEQUENTIAL_WORK()
3    START_FINISH()
4    ASYNC(COMPUTE)
5    START_FINISH()
6    ASYNC(COMPUTE)
7    COMPUTE()
8    END_FINISH()
9    ASYNC(COMPUTE)
10   COMPUTE()
11   END_FINISH()
```

`async−finish` program. We will often refer algorithm 4.3 to illustrate how the runtime maintains coherency automatically and what are the conditions to consider.

4.4.3 Starting a finish scope

Algorithm 4.4: Comparision between *start finish* implementation in U-RT and HC-K2H

```
1  procedure START_FINISH
2    wid ← WORKER_ID()
3    finish ← ALLOCATE_FINISH()
4    finish.counter ← 0
5    if U-RT then
6      finish.flush ← true
7      finish.gave_task ← 0
8    finish.parent ← GET_CURRENT_FINISH(wid)
9    SET_CURRENT_FINISH(wid, finish)
10   ATOMIC_INC(finish.parent)
```

The first runtime construct the master worker encounters when executing algorithm 4.1 is the START_FINISH construct. START_FINISH is used for starting a finish scope, whose use is detailed in section 2.3. Line 4 initializes the number of tasks in this finish scope to be zero. The counter will get atomically incremented whenever tasks are added into this finish scope, to count the number of pending tasks.

Line 5 details the changes done in U-RT to efficiently maintain cache-coherence during task transfers in divide-and-conquer parallelism. Two new variables, *flush* and *gave_task* are introduced. *flush* determines whether a cache-writeback-invalidate will be required before sending a task in this finish-scope to another cache-incoherent worker or not. It is set to true initially as there could possibly be some sequential work done before this finish scope started, whose result may be required by tasks present in the finish scope. Line 3 in algorithm 4.3 tells how such a condition may arise.

As discussed in section 2.2, modern processors like ARM and even DSPs have pre-fetching logic built into the hardware, using which they can pick up stale data (not yet written back) from the memory. Thus, we use the *gave_task* variable to tell us whether we need to invalidate the pre-fetch or not. The variable is incremented whenever we give a task to another processor which is cache-incoherent with the current processor. This is discussed in more detail in section 4.4.5.

Line 8 sets the parent finish scope of the current finish scope. Setting the parent finish scope is important in nested `async-finish` parallelism since we have a tree of finish scopes and after this finish scope ends, all tasks following this will belong to its parent finish scope. Thus, we would need to set the current finish scope to parent finish scope when this finish scope ends. Line 9 sets the current finish scope to this finish scope ensuring that all tasks generated by this worker after this finish scope and before the end of the finish scope, belong to this finish scope.

4.4.4 Asynchronous task creation

Next construct encountered by the master thread is the ASYNC construct, which is used for exposing parallelism in the code. Line 3 associates the parallel task with it's finish scope. Whenever this task completes execution, we decrement the counter of the finish scope of this task by 1. For now, we increment the counter as shown in the next line to indicate the addition of this task to this finish scope.

The **`if`** condition in line 5 provides one minor difference between U-RT and HC-K2H for the ARM side. In HC-K2H, the ARM workers check if DSPs are sitting idle or not, and push a task to shared hardware queue 1(sharedHWQ1) for the DSPs to steal it if they are sitting idle. This check is done by using shared memory and by using

Algorithm 4.5: Comparision between *async* implementation in U-RT and HC-K2H

```
1  procedure ASYNC(task)
2    heap_copy ← ALLOCATE_AND_INITIALIZE(task)
3    heap_copy.finish ← GET_CURRENT_FINISH(WORKER_ID())
4    ATOMIC_INC(heap_copy.finish.counter)
5    if ARM then
6      if U-RT then
7        PUSH_TO_DEQUE(heap_copy)
8      if HC-K2H then
9        if DSP_NEED_TASK() then
10         PUSH_TO_SHARED_HWQ1(heap_copy)
11       else
12         PUSH_TO_DEQUE(heap_copy)
13   if DSP then
14     PUSH_TO_DEQUE(heap_copy)
15     if U-RT then
16       UPDATE_TASK_AVAILABILITY()
17       COMMUNICATE()
```

cache-coherency operations frequently. In U-RT, the ARM workers do not give tasks to DSPs in ASYNC as a cache-writeback during ASYNC (explained in section 4.4.6) would stall task creation. Thus, this work is delayed for later.

On the DSP side, since HC-K2H used concurrent deques, it was able to push tasks directly to its queues. However, since we are using private deques on DSP side, we need to indicate our availability of tasks to other DSP workers and then send a task if it is required. Call to UPDATE_TASK_AVAILABILITY updates the task availability of the current DSP worker in shared memory. COMMUNICATE function is used for giving tasks to free workers and is explained in section 4.4.6.

4.4.5 Ending a finish scope

The next construct encountered by runtime in the execution of parallel mergesort is END_FINISH. The role of END_FINISH is to block and keep on executing tasks(Line 4) till all the tasks in the current finish scope have completed execution. Once that is done, both the HC-K2H and U-RT take different approaches to maintain cache coherency. HC-K2H always writeback-invalidates the cache at the end of each finish scope to maintain cache-coherency(Line 17). U-RT needs to take care of two separate issues when handling the end of finish scope. The first issue is when a task

Algorithm 4.6: Comparision between *end finish* implementation in U-RT and HC-K2H

```
1  procedure END_FINISH
2    wid ← WORKER_ID()
3    current_finish ← GET_CURRENT_FINISH(wid)
4    while current_finish.counter ≠ 0 do
5      FIND_AND_EXECUTE_TASK()
6    if U-RT then
7      if current_finish.gave_task > 0 then
8        INVALIDATE_PREFETCH()
9      if current_finish.parent ≠ NULL then
10       if ARM and current_finish.gave_task > 0 then
11         return
12       parent_finish ← current_finish.parent
13       START_CRITICAL_SECTION(flush)
14       parent_finish.flush ← true
15       END_CRITICAL_SECTION(flush)
16   if HC-K2H then
17     CACHE_WRITEBACK_INVALIDATE()
18   ATOMIC_DEC(current_finish.parent)
19   SET_CURRENT_FINISH(current_finish.parent)
20   DEALLOCATE_FINISH(current_finish)
```

from this finish scope was given to a cache incoherent worker. In this case, due to pre-fetching the worker encountering the finish scope needs to invalidate its pre-fetch since the caching hardware may have picked up some stale data from when the other worker had not written back its cache and was still in the midst of executing the task. This case is handled in line 7.

The second issue is that if a task is stolen after this finish scope ends, then depending upon the program structure, task getting created after this finish scope ends may need the data which had been computed inside the finish scope. Thus, we set the *flush* of parent finish scope to true to make sure that before giving such a task to another cache-incoherent processor a cache-writeback-invalidate always takes place. We also assume that the code written by the user will have no data races or false sharing issues. These two conditions ensure that no two workers can write to the same cache-line at the same time. We don't need a cache-writeback-invalidate in END_FINISH as we already writeback-invalidate our cache before giving a task to a cache-incoherent worker and delegate flush required at end of finish scope to the *flush* flag in the parent finish scope. One could say that we are being lazy in doing our cache-writebacks in the hope that we may not have to do it later, i.e. a steal may not happen for this or its parent finish scope.

This section ends the basic needs of any parallel runtime. In the sections below, we detail our work-stealing runtime procedures more thoroughly.

4.4.6 Giving Tasks to free workers

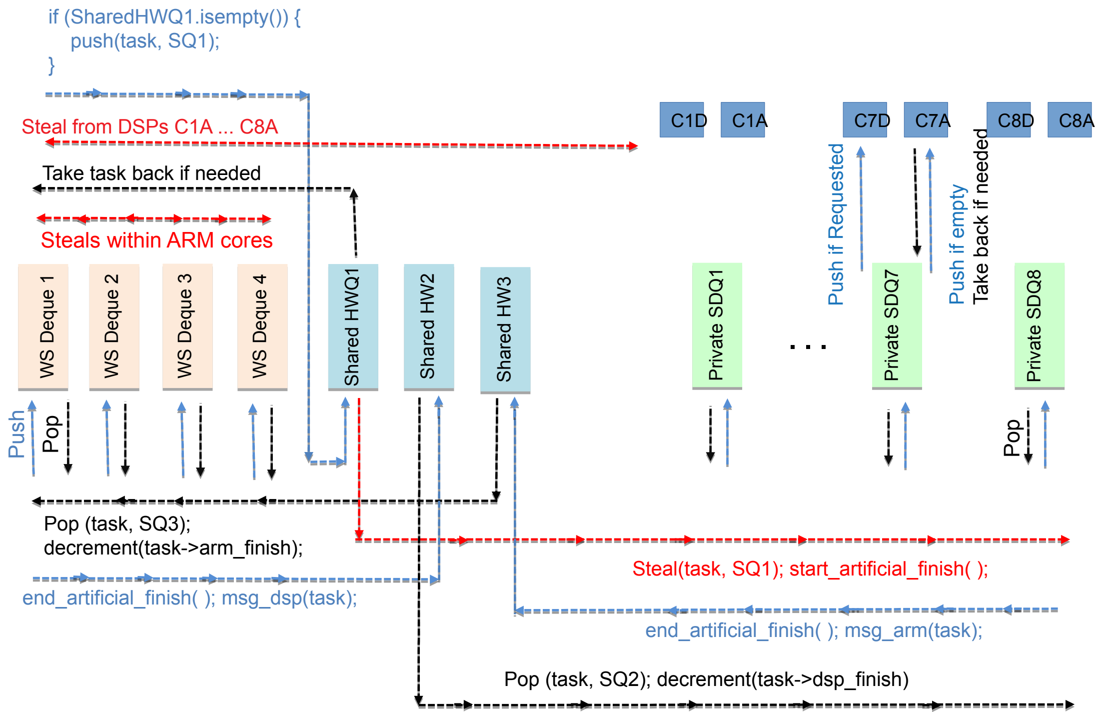


Figure 4.1: The communication structure of U-RT

Figure 4.1 shows the communication structure of U-RT. Parts of this figure will be explained in this section and parts in section 4.4.7, where we discuss how load-balancing actually takes place. In figure 4.1, C1D and C1A are communication cells for the first DSP worker. C1D is used for transferring tasks to other DSP workers and is implemented in fast shared memory. C1A is used for transferring tasks to ARM workers. Each DSP has its own private communication cells. The first DSP keeps a task in C1A if it finds the hardware queue to be empty as this is an indication that ARM workers have likely completed their share of computation and are now looking to take others computation. A similar procedure is followed by other DSPs also. DSP side follows the receiver-initiated model presented in Acar *et al.* (2013), where each DSP worker must request another DSP worker if it needs a task. This details the basic functioning of the COMMUNICATE algorithm on the DSP side. The ARM side only

checks whether the shared task deque, i.e. sharedHWQ1 is empty or not. If it finds it to be empty then its puts a task in it for the DSP workers. The code for COMMUNICATE is given in Algorithm 4.7

Algorithm 4.7: Communicate procedure used in U-RT

```
1  procedure COMMUNICATE
2   if U-RT then
3     if ARM then
4       if HWQ1_EMPTY() = false then
5         return
6     if DSP then
7       if ARM_STEAL_QUEUE_EMPTY() = false and DSP_REQUESTED_TASK
           () = false then
8         return
9     task ← DEQUE_POP_TOP()
10    if task = NULL then
11      return
12    finish ← task.finish
13    ATOMIC_INC(finish.gave_task)
14    START_CRITICAL_SECTION(flush)
15    if finish.flush = true then
16      CACHE_WRITEBACK_INVALIDATE()
17      finish.flush ← false
18    END_CRITICAL_SECTION(flush)
19    if ARM then
20      PUSH_TO_SHARED_HWQ1(task)
21    if DSP then
22      if ARM_STEAL_QUEUE_EMPTY() = true then
23        PUSH_TASK_FOR_ARM(task)
24      if DSP_REQUESTED_TASK() = true then
25        SEND_TASK_TO_DSP()
```

Before giving a task to a processor with which we do not maintain cache-coherency, we atomically increment *gave_task* (line 13, algorithm 4.7) flag to indicate that we have given a task to a processor which does not maintain cache-coherency with us. Its use is described in section 4.4.5. If condition, in line 15 in COMMUNICATE algorithm checks whether a cache flush is required or not, and if required then we set *flush* to false after doing a cache-writeback-invalidate to indicate that a cache flush will not be required when sending a task from this finish scope again. This technique saves a decent number of cache flushes in the case when there are multiple ASYNC calls inside the finish scope. It does have a catch although, in that doing some sequential work followed by an ASYNC call and the same repeated again would cause the technique to fail as no writeback would take place after second sequential work. Such cases are rare, but we

still provide two manual workarounds. First being to write a START_FINISH followed by END_FINISH to indicate to the runtime that a cache flush is required before sending the next task. The second solution is to simply encapsulate everything below this sequential work in a finish scope to indicate the same to the runtime.

COMMUNICATE procedure was not required by HC-K2H as it was using concurrent hardware-deques and had no requirement for giving tasks as the processors could simply take them whenever they wanted. An exception to this was the private link for transferring task from ARM to DSP, the logic of which has already been explained in section 4.4.4.

4.4.7 Stealing and executing tasks all the tasks

Algorithm 4.8: Procedure for finding and executing task

```
1  procedure FIND_AND_EXECUTE_TASK(task)
2    task ← GRAB_TASK_FROM_RUNTIME()
3    if task = null then
4      return
5    if U-RT then
6      if DSP then
7        UPDATE_TASK_AVAILABILITY()
8    COMMUNICATE()
9    EXECUTE_TASK(task)
10   if task.arm_stolen then
11     MSG_ARM_TASK_COMPLETED(task)
12   else if task.dsp_stolen then
13     MSG_DSP_TASK_COMPLETED(task)
```

The sole duty of algorithm 4.8 is to finish the execution of all tasks. To do this it relies on procedures GRAB_TASK_FROM_RUNTIME and EXECUTE_TASK, both detailed below. Just like in HC-K2H we do not carry over the finish scope of ARM to DSP and thus when a task stolen from the other processor completes, we message it to convey its completion. The other processor then decrements the task counter of the task's finish scope.

Grabbing tasks

Procedure GRAB_TASK_FROM_RUNTIME for U-RT is best explained using figure 4.1. The main job of GRAB_TASK_FROM_RUNTIME is to find some task to execute. For this, each processor first tries to pop from its local deque to see if it has some task available or not. If there is no task available, then each processor tries to take the task which it may have kept for inter-architecture steals. In ARM case, this would mean taking task back from sharedHWQ1 and for DSPs, it would mean taking task back from communication cell reserved for ARM processor. In doing so each processor also decrements the *gave_task* flag of the finish scope of the task. If even this does not succeed, then just like HC-K2H, first a steal operation is performed within the same architecture, failing on which finally an inter-architecture steal is carried out.

For HC-K2H, this procedure is much simpler. The workers first try to pop from their deque (or hardware queue), failing which they try an intra-architecture steal. If even that fails, then finally inter-arch stealing is performed to get tasks.

Executing tasks

Algorithm 4.9 states the procedure for executing a task. First, it is important to identify whether the current task and the sub-tasks created by it should be completed before proceeding further or not. This may be required to either writeback the cache after execution of the task or for sending the message about the completion of this task. If it is required then we must encapsulate the task in a finish scope so that the task itself and all the sub-tasks generated by it finish before we proceed further to writeback the cache or to convey the message about task completion. In HC-K2H this would only have to be done when an inter-architecture steal happened as then we would writeback-invalidate our cache and also send a message about the task's completion. It was not required when steals happened within the same architecture, even if they happened among the cache-incoherent DSP cores. The reason for this lies in line 16 of the algorithm, as the DSP always writeback-invalidate their cache during task execution. To remove this overhead, U-RT also creates these artificial finish scopes when a task is stolen from a processor with which our cache is incoherent, and finally, it writeback-invalidates its cache in END_ARTIFICIAL_FINISH procedure. Doing this is more efficient as a task

Algorithm 4.9: Comparision between task execution procedure in both runtimes

```
1  procedure EXECUTE_TASK(task)
2    if ARM then
3      if task.dsp_stolen = true then
4        create_finish ← true
5    if DSP then
6      if U-RT then
7        if task.arm_stolen = true or task.dsp_stolen = true then
8          create_finish ← true
9      if HC-K2H then
10       if task.arm_stolen = true then
11         create_finish ← true
12   finish ← task.finish
13   if create_finish = true then
14     START_ARTIFICIAL_FINISH(task)
15   if HC-K2H then
16     if DSP then
17       CACHE_WRITEBACK_INVALIDATE()
18   task.func()
19   if HC-K2H then
20     if DSP then
21       CACHE_WRITEBACK_INVALIDATE()
22   if create_finish = true then
23     END_ARTIFICIAL_FINISH()
24   ATOMIC_DEC(finish.counter)
```

being executed is likely to generate many more tasks and we would save the overhead of cache-writeback there.

Algorithm 4.10: Comparision between *START_ARTIFICIAL_FINISH* implementation in U-RT and HC-K2H

```
1  procedure START_ARTIFICIAL_FINISH(task)
2    wid ← WORKER_ID()
3    finish ← ALLOCATE_FINISH()
4    finish.counter ← 0
5    INVALIDATE_PREFETCH()
6    if U-RT then
7      finish.flush ← false
8      finish.gave_task ← 0
9    finish.parent ← NULL
10   task.finish ← finish
11   ATOMIC_INC(task.finish.counter)
12   SET_CURRENT_FINISH(wid, finish)
```

Algorithm 4.10 is similar to START_FINISH algorithm. The main difference is that we must invalidate our pre-fetch and U-RT also sets the *flush* flag to false since there could be no work done by this worker which it may have in its memory and which may be required by further tasks. The parent of the finish scope is also set to *NULL* since there is no logical parent for such a finish scope.

Algorithm 4.11: Comparision between *END_ARTIFICIAL_FINISH* implementation in U-RT and HC-K2H

```
1  procedure END_ARTIFICIAL_FINISH
2    wid ← WORKER_ID()
3    current_finish ← GET_CURRENT_FINISH(wid)
4    while current_finish.counter ≠ 0 do
5      FIND_AND_EXECUTE_TASK()
6    if U-RT then
7      CACHE_WRITEBACK_INVALIDATE()
8    if HC-K2H then
9      if ARM then
10       CACHE_WRITEBACK_INVALIDATE()
11   DEALLOCATE_FINISH(current_finish)
```

Algorithm 4.11 is similar to END_FINISH, except that it also writeback-invalidates the cache to make the result of the computation available to the original processor. The only exception in HC-K2H is that cache-writeback is not done on DSP side as DSPs always writeback their cache during task execution.

Finally, the finish scope's task counter is decremented by EXECUTE_TASK to indicate the completion of the task.

4.4.8 Manual cache flushes for flat-finish based parallel *for* loops

While our insight of taking a lazy approach in cache flushing works well for recursive divide-and-conquer parallelism, for parallel `for` loops a pair of user invoked cache flushes are required outside the flat-finish scope for the correct working of U-RT (as shown in Figure 4.12). To see why, we need to understand when all could a cache flush be required in parallel `for` loops. On the ARM side, we need to writeback-invalidate the cache followed by invalidating the pre-fetch on DSP side. This will make the memory view of processors coherent before starting the execution of parallel `for` loop. After the execution of `for` loop completes, the DSP side needs to writeback-invalidate their cache followed by invalidating the pre-fetch on ARM side. Those are all the cache operations that will be required on the execution of a parallel `for` loop. However, we would still be doing a cache-writeback-invalidate on artificial finish scopes which get created when we steal from another cache-incoherent processor (section 4.4.7). As explained above, it is completely unnecessary to do a cache-writeback-invalidate there.

To overcome the above-stated inefficiency, we provide the user with two new functions: PRE_FLUSH and POST_FLUSH. These functions do the above operations and also turn off the automatic cache coherence maintenance operations to support the efficient execution of such programs. Algorithm 4.12 shows how to use these functions.

Algorithm 4.12: Pseudocode showing usage of PRE_FLUSH and POST_FLUSH API's

```
1 PRE_FLUSH()
2 START_FINISH()
3 FORASYNC(MATMUL)
4 END_FINISH()
5 POST_FLUSH()
```

4.5 Experimental Evaluation

We first start by measuring the performance benefit obtained from U-RT as compared to HC-K2H for both nested `async–finish` and `for` loop based benchmarks. We then examine the speedup obtained over sequential ARM execution by using U-RT for all our benchmarks.

4.5.1 Speedup of U-RT over HC-K2H by using all ARM and DSP cores

In this section, we discuss the performance of U-RT relative to HC-K2H by using all ARM and DSP cores. We first start the discussion with the results for nested `async–finish` benchmarks followed by that of `for` loop benchmarks.

nested **`async–finish`** benchmarks

Table 4.2: Speedup of U-RT relative to HC-K2H by using all ARM and DSP cores for nested `async–finish` parallelism

Benchmark Name	Speedup Relative to HC-K2H
Mergesort	1.733
Jacobi	31.149
Matmul	1.075
Cilk Sort	2.082

Table 4.2 shows the results of this experiment for nested `async–finish` benchmarks. The performance improvement for Jacobi is massive. This was expected due to its memory-intensive nature which would greatly benefit from reduced pressure over memory. Both Mergesort and Cilksort also benefitted heavily from reduced cache write-backs and increased locality due to the use of private work-stealing deques. Matmul didn't give the same performance advantage compared to others. The reason likely is that unlike other benchmarks, in matmul, the number of writes are comparatively less compared to the number of reads. Thus, most of our cache is going to consist of read-only data, which will get invalidated but will not need to be written back in cache-writeback-invalidate operation as it is not dirty. Since the cost of a cacheline-invalidate

is much lower than that of cacheline-writeback, we observe a lower benefit in matmul.

`for` loop benchmarks

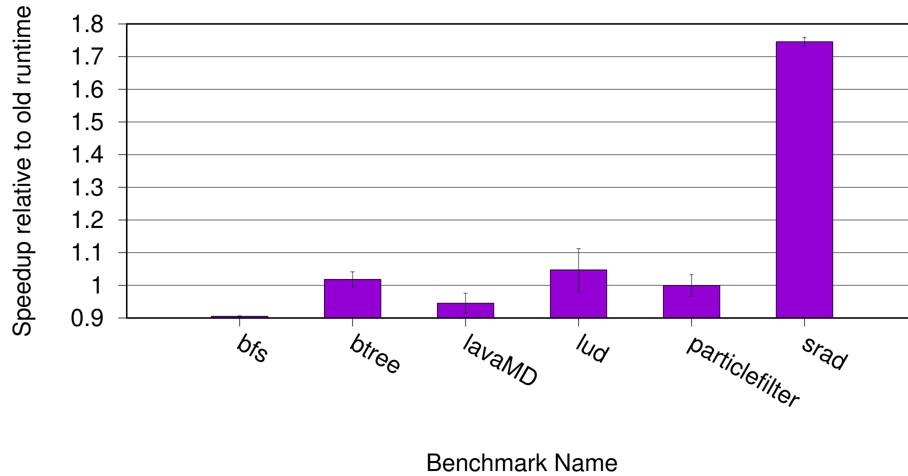
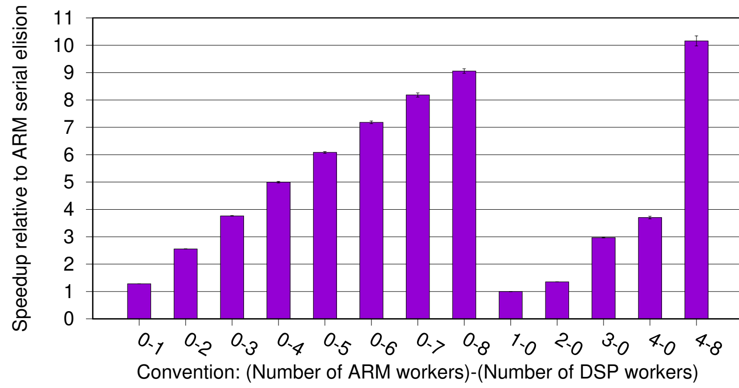


Figure 4.2: Speedup obtained in U-RT relative to HC-K2H when all available ARM and DSP workers are used

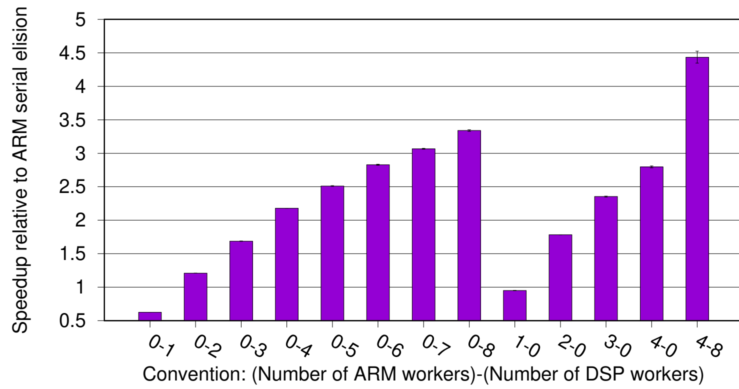
Figure 4.2 shows the results of this experiment for `for` loop benchmarks. We were unable to run Tiled Matmul and Hotspot over HC-K2H and hence their results have been dropped from figure 4.2. All benchmarks used PRE_FLUSH and POST_FLUSH API's as discussed in section 4.4.8. As we can observe, the performance of U-RT is comparable to that of HC-K2H. LavaMD and BFS performs slightly better in HC-K2H compared to U-RT as these benchmarks generate coarse granular tasks which have slightly poorer performance when using private deques. Reason for this is that the victim will not be able to quickly respond to steal requests when compared to the execution of fine granular tasks.

4.5.2 Speedup over sequential execution

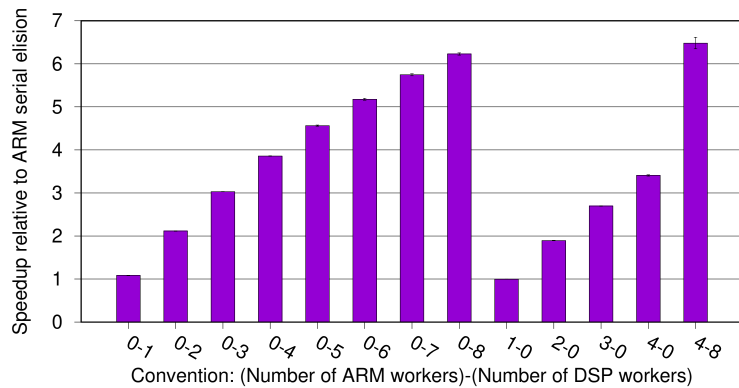
In this section, we discuss the speedup obtained in U-RT relative to the sequential ARM execution. We first start the discussion with the speedup results for nested `async–finish` benchmarks followed by that of `for` loop benchmarks.



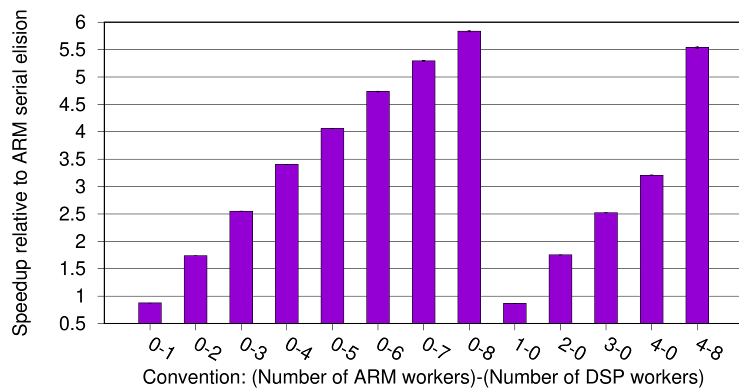
(a) Matmul



(b) Mergesort



(c) Cilksort



(d) Jacobi

Figure 4.3: Speedup obtained by using U-RT relative to sequential ARM execution

Nested `async–finish` benchmarks

Figure 4.3 shows the speedup obtained in nested `async–finish` benchmarks by executing them over U-RT relative to the sequential ARM execution. We can observe that for DSP-*only* execution, each of the four benchmarks is able to achieve good speedup over the sequential ARM execution by increasing the number of workers. Matmul when executed with a single DSP worker, performs slightly better than single ARM worker execution. This is the reason behind slight super-linear speedup in Matmul when scaling by using DSP-*only* execution. Cilksort and Jacobi perform nearly similar for DSP-*only* execution. However, Mergesort doesn't scale that well and even its single DSP worker execution takes nearly twice the time taken for single ARM worker execution. The likely reason may lie in the fallback algorithm used when merge size becomes too small. Mergesort falls back to insertion sort, whereas cilksort falls back to quicksort when the size of arrays to be merged is small.

For ARM-*only* execution, each of the four benchmarks is able to achieve almost linear speedup over the sequential ARM execution with the increasing number of workers.

For hybrid execution with 12 workers, i.e., by using all ARM and DSP cores, each of the four benchmarks are able to achieve better speedup than obtained with four ARM-*only* execution. Matmul is able to achieve the highest speedup of 10$\times$, Cilksort achieves 6.5$\times$, whereas Mergesort and Jacobi are in the same ballpark (4.5$\times$ and 5.5$\times$ respectively). Due to the sequential merge step inside Mergesort, tasks are slightly heavyweight as compared to Cilksort. Due to the presence of private deques, in case of Mergesort benchmark DSP workers process the steal requests slightly slower as compared to Cilksort, thereby resulting in slightly slower speedup than Cilksort. Jacobi achieves only a marginal improvement in speedup by using all ARM and DSP cores because of high-memory bandwidth requirement that gets even higher when cache coherency operations are being carried out. Compared to other benchmarks, Jacobi has an almost one-to-one mapping from output to input with little cache-utilization or computation in between, which leads to higher memory bandwidth requirement. The stolen task also have lower work present in them compared to other benchmarks, which is not enough to justify the cost of cache-writeback that we have to do. Another point to note is that all such write intensive benchmarks would also cause more cache-lines to become dirty, increasing the cost of cache-writeback-invalidate itself, since dirty cache

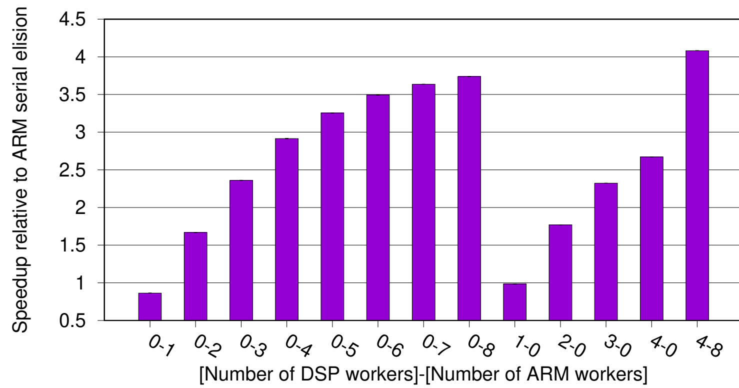
lines have to be written back to memory, unlike clean cache-lines which can simply be invalidated.

`for` loop benchmarks

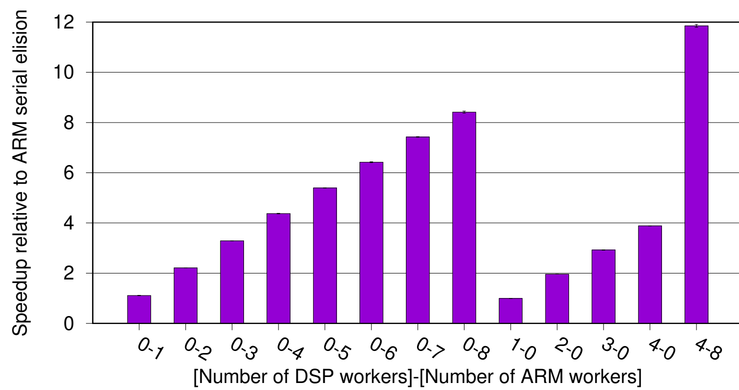
Figure 4.4 shows the speedup obtained in `for` loop benchmarks by executing them over U-RT relative to the sequential ARM execution. We can observe that for DSP-*only* execution, except Particle Filter and SRAD, all other benchmarks are able to achieve good speedup over the sequential ARM execution by increasing the number of workers. SRAD and Particle Filter perform poorly over DSP as they contain operation like division which cannot be handled efficiently by the DSP architecture. Tiled Matmul and LUD are very efficient on DSP, which shows the effectiveness of DSP architecture in handling matrix multiplication style kernels. The speedup in Tiled Matmul relative to ARM serial elision is super-linear and almost staggering. It is a good indication of the full potential of TI's C66x DSP.

For ARM-*only* execution, most of the benchmarks, except BFS, are able to achieve almost linear speedup over the sequential ARM execution with an increasing number of workers. BFS is likely not able to scale due to higher memory bandwidth requirements.

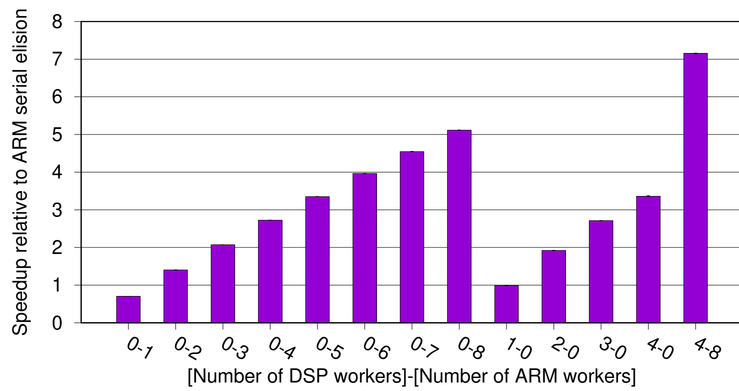
For hybrid execution with 12 workers, i.e., by using all ARM and DSP cores, each of the benchmarks (except Tiled Matmul) are able to achieve much better speedup than that obtained with four ARM-*only* or DSP-*only* execution. Moreover, this speedup is more than what nested `async–finish` benchmarks obtained in hybrid execution. This is expected because unlike in nested `async–finish` benchmarks, there is no cache-writeback-invalidate happening here during the loop execution.



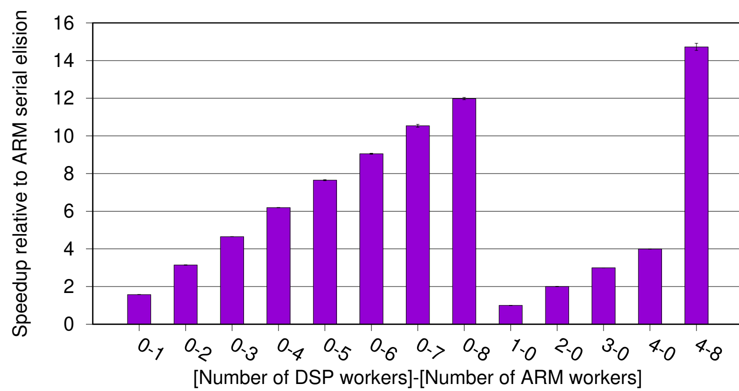
(a) BFS



(b) B+tree

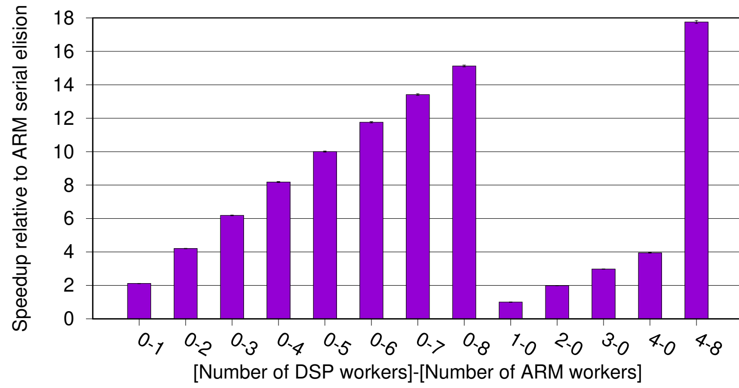


(c) Hotspot

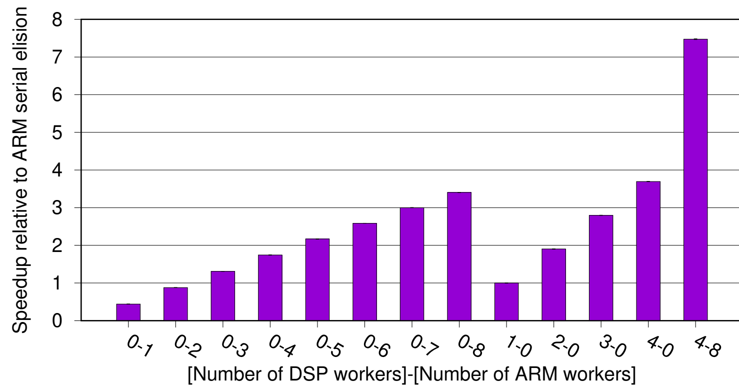


(d) LavaMD

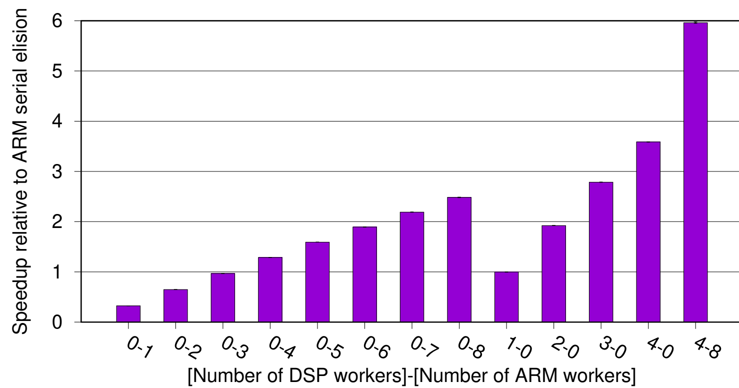
Figure 4.4: (Cont.) Speedup obtained by using U-RT relative to sequential ARM execution



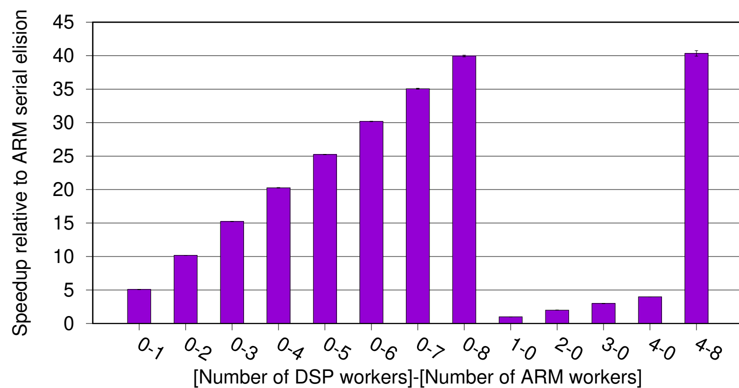
(e) LUD



(f) Particle Filter



(g) SRAD



(h) Tiled Matmul

Figure 4.4: Speedup obtained by using U-RT relative to sequential ARM execution

4.6 Summary

This chapter first showed the extent of overheads due to repeated cache-writebacks in HC-K2H for nested `async-finish` benchmarks. These frequent cache flushes were the side-effect of using hardware queues at DSPs. We propose the design and implementation of a novel hybrid work-stealing runtime for Keystone-II Hawking MPSoC, U-RT, that uses private software deques instead of hardware queues at DSPs. This design significantly reduced the total cache write back and invalidation operations, thereby improving the performance of both nested `async-finish` and `for` loop based parallel programs.

CHAPTER 5

PRODUCTIVELY DETERMINING ENERGY-OPTIMAL PLACE OF EXECUTION

Recall from section 2.5 that optimizing for power and optimizing for energy over MPSoC are two different goals. However, it is a daunting task for the programmer to determine the energy-optimal work-partitioning for MPSoC. Using a dynamic load balancing runtime, the problem of finding the energy-optimal partition is simplified to finding the energy-optimal place of execution. This chapter addresses this issue by introducing a novel approach for automatically determining the energy-optimal place of execution (henceforth, referred to as EOPE in this chapter) over Keystone-II Hawking MPSoC, where place maybe ARM, DSP or ARM+DSP.

The rest of this chapter is structured as follows. Section 5.2 provides the motivation for having a runtime to automatically find the EOPE. Section 5.3 describes the insights and the basic approach used for determining the EOPE. Section 5.4.1 describes the design and implementation of a runtime based approach for determining the EOPE. Section 5.5 describes the experimental evaluation of our novel approach using several well known benchmarks, and finally, section 5.6 concludes this chapter.

5.1 Introduction

This chapter uses the energy model proposed by Mitra *et al.* (2016) and develops an automated approach to determine the EOPE on Keystone-II Hawking MPSoC by using the U-RT runtime. We develop this automated approach for `for` loop based parallelism comprising of loop iterations with a similar amount of work in each iteration i.e., regular `for` loops. By using this approach, programmers can execute on the EOPE on Keystone-II Hawking MPSoC directly during the actual execution without the need for profiling multiple trial runs.

The principal contributions in this chapter are as follows: a) a new API FORASYNC_ENERGY that can automatically determine the EOPE in Keystone-II

Hawking MPSoC during actual execution; b) design and implementation of this API over U-RT runtime; and c) a detailed experimental evaluation of the proposed approach using several well known benchmarks.

5.2 Motivation

Energy has become an important optimization goal along with performance especially in the embedded and high performance communities. Along with optimizations to hardware, optimizations in the software are also required to make use of the hardware in the most energy-optimal way, i.e., giving more emphasis to *energy-to-solution* rather than the *time-to-solution*. Nowadays, software for embedded systems is often made in a way such that it is fine if the performance degrades without any noticeable effect on the user if the consumption of energy is reduced. The goal, however, remains how to provide an automatic mapping of tasks on to heterogeneous processing elements that could provide energy-optimal partition. Without this automatic support, programmers have to perform trial runs of manually created work-partitions to discover the energy-optimal partition. Mitra *et al.* (2016) provided an energy model to predict whether it would be worth splitting the iterations of a parallel `for` loop across heterogeneous processors or not. However, for using this energy-model a programmer is still left to manually partition the iterations of the parallel `for` loop.

5.3 Insights

The insight we use is that when sufficient parallelism is available in parallel `for` loop, our hybrid work-stealing runtime U-RT can automatically create multiple work-partitions for Keystone-II Hawking MPSoC and execute it at different processing elements (*ARM*, *DSP*, or *COMBINED*) to determine the EOPE. To implement this insight, we propose a new API FORASYNC_ENERGY that can automatically determine the EOPE for a parallel `for` loop.

Difference between FORASYNC_ENERGY and FORASYNC (Section 2.4) is not in syntax but rather in hinting the runtime for an energy-to-solution execution via FORASYNC_ENERGY or for time-to-solution execution via FORASYNC.

5.4 Implementation

5.4.1 Energy Model

Let the power draw of the system during ARM execution be P_c, during DSP execution be P_d and during simultaneous execution by both ARM and DSP be P_{cd}. Let E_c be the energy consumed when the entire computation is executed on ARM side, E_d be the energy consumed when the entire computation is executed on DSP side and E_{cd} be the energy consumed when both ARM and DSP execute the computation simultaneously. Let N be the total computational cost and let the rate of computation for ARM be R_c, the rate of computation for DSP be R_d and the rate of computation for simultaneous execution be R_{cd}. We use the knowledge that dynamic load-balancers like work-stealing scheduler do not need to split the computation as they can automatically schedule the computation efficiently with minimum idle time for each worker, given that it has sufficient parallelism (Acar *et al.*, 2013). Thus, we are only interested in finding out which of these subsets of processors (in our case $\{ARM\}$, $\{DSP\}$ and $\{ARM, DSP\}$) will lead to minimum total system energy consumption. So, we modify equations 2.2, 2.3 and 2.4, and re-write the equations for energy consumption of system at each place as

$$E_c = \frac{N}{R_c} P_c \tag{5.1}$$

$$E_d = \frac{N}{R_d} P_d \tag{5.2}$$

$$E_{cd} = \frac{N}{R_{cd}} P_{cd} \tag{5.3}$$

From the above equations, we can derive which one out of ARM, DSP or combined execution will lead to least energy consumption.

ARM will be energy optimal if $E_c < E_d$ and $E_c < E_{cd}$. Using equations 5.1, 5.2 and 5.3, we can derive the optimality criteria for ARM as

$$P_c < P_d \frac{R_c}{R_d} \quad and \quad P_c < P_{cd} \frac{R_c}{R_{cd}} \tag{5.4}$$

Similarly, execution at DSP will be optimal if

$$P_d < P_c \frac{R_d}{R_c} \quad and \quad P_d < P_{cd} \frac{R_d}{R_{cd}} \tag{5.5}$$

and execution at both ARM and DSP will be optimal if

$$P_{cd} < P_c \frac{R_{cd}}{R_c} \quad and \quad P_{cd} < P_d \frac{R_{cd}}{R_d} \tag{5.6}$$

By using the above equations, the problem of executing at the EOPE reduces to finding the power draw values P_c, P_d, P_{cd} and the ratio of rates of computation $\frac{R_c}{R_d}$, $\frac{R_{cd}}{R_d}$ and $\frac{R_{cd}}{R_c}$. The technique used in this thesis for finding out these values is discussed in the next section.

5.4.2 Determining energy-optimal place of execution by using U-RT

The problem of finding the values of P_c, P_d, P_{cd} and values of $\frac{R_c}{R_d}$, $\frac{R_{cd}}{R_d}$ and $\frac{R_{cd}}{R_c}$ can be divided into four stages. Three stages for finding the values of P_c, P_d, P_{cd} each and one stage for finding the values of $\frac{R_c}{R_d}$, $\frac{R_{cd}}{R_d}$ and $\frac{R_{cd}}{R_c}$. Every stage is independent of the other stages and can be profiled separately and even across different invocations of FORASYNC_ENERGY for the same loop kernel, but the stage for calculating rates of computation must complete within the same invocation of the loop. The reason for this is that we do not assume that different invocations of a parallel loop will have the same amount of work in each iteration. However, we do assume that within an invocation of the loop, each iteration will have the same amount of work, limiting our approach to regular parallel `for` loops. We also assume that the power and performance characteristics of the parallel `for` loop do not change with change in input during different invocations, and thus store the energy optimal place once a loop has been completely profiled and re-use it later during the same program run. This reduces the overhead of profiling the loop again and again. Time is profiled for each core individually and the number of tasks executed by that core are also stored along with the time to find out the approximate computation time of each task. Energy consumption can only be profiled for the entire system. Note that we do not assume that $\frac{R_{cd}}{R_c} = R_c + R_d$ and thus profile

each place separately even for the rate of computation measurements.

Algorithm 5.1: Scheduling Task on Energy Optimal Place

```
1  struct loop_info
2    dimension
3    start[dimension]
4    end[dimension]
5    stride[dimension]
6    tile[dimension]
7    kernel
8
9  procedure FORASYNC_ENERGY(loop_info)
10   profile_info ← RETREIVE_CURRENT_PROFILE_INFO(loop_info.
        kernel)
11   if LOOP_PROFILING_COMPLETE(loop_info) then
12     EXECUTE_ENERGY_OPTIMALLY(loop_info, profile_info)
13     return
14   time_estimates ← ESTIMATE_TASK_EXECUTION_TIMES(loop_info)
15   if POWER_PROFILING_COMPLETE(ARM, profile_info) = false then
16     tasks ← APPROXIMATE_TASKS(ARM, time_estimates)
17     PROFILE_POWER_DRAW(ARM, loop_info, profile_info, tasks)
18   if POWER_PROFILING_COMPLETE(DSP, profile_info) = false then
19     tasks ← APPROXIMATE_TASKS(DSP, time_estimates)
20     PROFILE_POWER_DRAW(DSP, loop_info, profile_info, tasks)
21   if POWER_PROFILING_COMPLETE(COMBINED, profile_info) = false
        then
22     tasks ← APPROXIMATE_TASKS(COMBINED, time_estimates)
23     PROFILE_POWER_DRAW(COMBINED, loop_info, profile_info, tasks)
24   if RATES_OF_COMPUTATION_PROFILING_COMPLETE(profile_info) =
        false then
25     PROFILE_RATES_OF_COMPUTATION(loop_info, profile_info)
26   EXECUTE_ENERGY_OPTIMALLY(loop_info, profile_info)
```

Algorithm 5.1 states the basic algorithm for finding the EOPE, the components of which have been described below.

5.4.3 Basic Helper Routines

This section details some basic helper functions which have been used in Algorithm 5.1.

RETREIVE_CURRENT_PROFILE_INFO

Whenever we profile a loop, we store its info for this run of the program so that we could use it later without needing to profile the loop again. This saves us from sub-

stantial overheads which are spent when tuning for only one processor and also allows us to tune in parts if a single run of the loop is not enough to gain the entire profile information. As of now, the data we save is temporal and is lost when the program is executed again. The reason for doing this is to not to show an unfair advantage in the benchmarks. But in the real world, there is nothing which would stop someone from storing the data in a file and then retrieving it later, negating any overhead that one incurs due to profiling. If there is no previous data of profiling for this loop, then RETREIVE_CURRENT_PROFILE_INFO returns a newly created *profile_info* structure.

LOOP_PROFILING_COMPLETE

This function simply tells whether all the necessary profiling information for this loop has been gathered or not.

EXECUTE_ENERGY_OPTIMALLY

This function uses the information in *profile_info* structure and the model presented in 5.4.1 to execute the loop at its EOPE. The processors which are idle are sent to sleep, using the implementation mentioned in 5.4.5.

5.4.4 Pre-tuning: Estimating Execution Times

The pre-tuning stage is used for estimating the execution times for each task in a parallel `for` loop. A task in a parallel `for` loop is simply the number of iterations in a parallel `for` loop divided by the tile size of the loop. This stage is important mainly because of two reasons, first being that it later allows us to profile power for sufficient time to gain accurate power profiling information and second being that it allows us to profile for the minimum amount of time necessary to gain accurate profiling information, which reduces the overheads due to profiling.

ESTIMATE_TASK_EXECUTION_TIMES

This function is responsible for estimating the time taken on both sides for executing a task. To achieve this, a timer is associated with each processor which begins just before

task execution and ends just after it. The time calculated is then stored (possibly added) in a worker private cell along with the number of tasks that it has executed. Any issues arising due to context switching have been ignored in the current implementation. In our current implementation, 10% of the loop is reserved for this stage, but this can be changed through an environment variable. This function must also run every time a loop runs before profiling is complete because the size of the computation in the loop may have changed from the previous run. We found the best performance using work-sharing in this stage since the computation is small and the private link between ARM and DSP steals as mentioned in section 4.4.6 was not giving the expected speedup because of insufficient parallelism.

APPROXIMATE_TASKS

This function uses the time estimates gathered by ESTIMATE_TASK_EXECUTION_TIMES to approximate the time it takes for each processor to execute a task. From these estimates, the number of tasks required for keeping the worker busy for enough time to get accurate profiling information can be calculated. Currently, we use the following heuristic to give tasks to each of these stages: Give around 5ms of time for profiling the rates of computation for ARM and DSP side and around 5ms of time after load-balancing the tasks for profiling the rate of computation during simultaneous execution by both processors. For power measurements, we use the same technique except that around 20ms of computation is given, though it may reduce to 10ms if the loop is not big enough. The reason for choosing these values is that our energy measurement framework has been tuned to send a reading every 1ms (section 2.5) and we want to get a certain amount of readings to get accurate results while minimizing the overheads of profiling.

Profiling the power draw in execution across ARM and DSP workers is tricky since we can only guarantee that around 20ms of computation will be given to each, but not exactly how much. If tasks are coarse granular and have a lot of work in them, then the time may end up being well above or below 20ms. Moreover, it may end up being highly variable depending on the rate of execution of different processors. We cannot get accurate readings if the split is not well balanced as a decent amount of readings would end up being for ARM only or DSP only computation. To resolve this issue we

balance the expected computation time given to each processor till it is within 10% of the other processor, before giving the computation to them. The percentage value has been chosen such that the profile time does not increase by a huge amount (will be at most 10x the time taken at ARM or DSP side, whichever is higher) and yet the readings we get are accurate, i.e, 90% of the readings are for simultaneous execution. In most of the practical cases, this will not lead to a huge increase in profile time if the tasks are finely granular.

5.4.5 Tuning Power Draw

POWER_PROFILING_COMPLETE

This function simply tells us whether the power draw of the system has been profiled for the side given to it.

PROFILE_POWER_DRAW

This function is used for profiling the power draw of the system for the processor given as an argument to it. The power draw profiled is then stored in the *profile_info* structure. When profiling power, one issue is to decide whether to include inter-architecture work-stealing during simultaneous execution in the profiling phase or not. We turn off inter-architecture stealing because a steal at the wrong moment near the end of the execution, especially by the slower processor may completely ruin the efforts we did in section 5.4.4 to balance the execution time at each worker. Moreover, this stage first statically partitions the work to each processor as we do not want to include any imbalances which may arise due to dynamic-load balancing in our results. Profiling execution at any single side is not as tricky as profiling simultaneous execution. To reduce the contribution of ARM power draw during DSP profiling, we temporarily reduce the number of workers on ARM side to 1(the master thread) using condition variables. To reduce the power draw of DSP side during ARM profiling, we simply make the DSPs execute NOPs. To further reduce energy during DSP only execution, the master ARM thread goes to sleep for around 0.1ms when polling for task completion by DSP. This contrasts with FORASYNC, where the master ARM thread keeps on polling continuously.

5.4.6 Tuning Rate of Computations

RATES_OF_COMPUTATION_PROFILING_COMPLETE

This function simply tells us whether all the ratios of rate of computation have been calculated or not.

PROFILE_RATES_OF_COMPUTATION

This function is responsible for calculating the ratio of rates of computation for this loop. For this, it simply keeps a track of the number of tasks executed and the time spent in them. Using this information the ratio of rates of computation are calculated. Just like in ESTIMATE_TASK_EXECUTION_TIMES, the timer starts before task execution and ends just after it. The reason for doing so is that we do not want to take into account any imbalances due to dynamic load-balancing runtime as while the overhead of it may be high for the small number of tasks given to this stage, it will end up amortizing eventually when the entire loop is executed.

5.5 Experimental Evaluation

For the sake of clarity and comprehensibility, the values for only the most time consuming kernel (loop) for each benchmark have been shown in the tables below even if there were multiple parallel loops in the benchmark. These were also the most energy consuming loops in all benchmarks and accounted for more than 90% of the energy consumption in majority of the benchmarks. One should note that this is not a limitation of the runtime as FORASYNC_ENERGY can be used with all parallel `for` loops.

The energy values shown are always the approximate energy consumption values since the energy measurement equipment sends a reading at every 1ms as discussed in section 5.4.4. Thus, it is not possible to accurately measure energy consumption value when the time taken is 6.5 or 6.7 etc. Also, for showing the times, the values of time required for flushing the cache and overheads of the energy measurement framework have been removed for the sake of analysis.

Table 5.1 compares the time taken when executing at different places using

FORASYNC, with scheduling limited to the given processor(s). In our case, heterogeneous execution, i.e., ARM + DSP execution always performs the best.

Table 5.1: Time Taken during ARM, DSP and ARM+DSP execution

Benchmark Name	Time Taken for ARM execution	Time Taken for DSP execution	Time Taken for ARM+DSP execution
BFS	1.801	1.282	1.148
B+Tree	0.281	0.087	0.067
Hotspot	1.310	0.816	0.553
LavaMD	16.981	5.622	4.549
LUD	19.595	4.313	3.610
Particle Filter	62.127	58.031	30.143
SRAD	1.520	2.573	0.964
Tiled Matmul	60.590	6.049	5.990

Table 5.2 compares the energy consumed when executing at different places using FORASYNC, with scheduling limited to the given processor(s). Here we see that while heterogeneous execution is always the fastest, it may not always be the most energy efficient. In our case, tiled matmul consumes lower energy when executing only at DSP.

Table 5.2: Energy consumed during ARM, DSP, ARM+DSP execution

Benchmark Name	Energy Consumed for ARM execution	Energy Consumed for DSP execution	Energy Consumed for ARM+DSP execution
BFS	39.137	25.483	24.226
B+Tree	6.027	1.761	1.540
Hotspot	28.322	16.475	12.436
LavaMD	360.634	106.986	98.540
LUD	442.255	90.881	88.846
Particle Filter	1305.200	1156.550	675.229
SRAD	31.867	50.598	20.862
Tiled Matmul	1302.54	126.081	140.587

Table 5.3 verifies whether the approximated values of P_c, P_d, P_{cd}, $\frac{R_c}{R_d}$, $\frac{R_{cd}}{R_d}$ and $\frac{R_{cd}}{R_c}$ as measured using FORASYNC_ENERGY match with those calculated using FORASYNC. The approximations of ratios of rate of computation seem to be slightly less accurate for Particle Filter, LavaMD, Tiled Matmul and BFS. For BFS, we observed that increasing the time for which the loop is profiled led to more accurate

readings advocating that an adaptive approach may be better. For the others, the difference is due to the differences in performance between static partitioning used during profiling and work-stealing using private deques used during normal execution. We also notice that calculated P_c when using FORASYNC is higher compared to P_c approximated when using FORASYNC_ENERGY. This difference is there because when using FORASYNC_ENERGY the DSPs execute multiple NOPs when they are not required, reducing the power usage (Section 5.4.5). Also, P_d calculated when using FORASYNC is always higher then the P_d calculated when using FORASYNC_ENERGY because as explained in section 5.4.5, the master ARM core keeps on polling continuously when using FORASYNC but follows a sleep-poll-sleep-poll strategy when using FORASYNC_ENERGY. We observe no such trend during simultaneous (ARM+DSP) execution and the differences are mainly due to inaccuracies and differences in the performance of work-stealing runtime used during normal execution and static partitioning done during profiling.

Table 5.4 shows the overheads incurred due to profiling and the accuracy in the judgment of FORASYNC_ENERGY. Only benchmarks incurring decently high execution time during a single loop invocation are shown as their energy readings will be the most accurate. BFS is shown just to show the energy overheads, as there was an error of around 30% in its reading. Except for LUD, the most energy efficient place of execution was always chosen for all benchmarks. The likely reason for the error in LUD may be due to the fact that the percentage difference in energy consumption between executing at DSP and ARM+DSP is not that high. We observe that overheads of *forasync_energy* are below 10% in most cases and sometimes negligible. For Tiled Matmul, which is more efficient at DSP, we even achieve energy savings of around 5% compared to executing using FORASYNC, even after incurring the cost due to profiling. If the kernel were to run again then we won't incur any more overhead and would save around 10% energy.

Table 5.3: Table comparing values computed using FORASYNC with values approximated by FORASYNC_ENERGY during profiling

Benchmark Name	FORASYNC $\frac{R_c}{R_d}$	Approximated $\frac{R_c}{R_d}$	FORASYNC $\frac{R_{cd}}{R_c}$	Approximated $\frac{R_{cd}}{R_c}$
BFS	0.711	0.692	1.569	2.011
B+Tree	0.309	0.32	4.194	4.080
Hotspot	0.623	0.649	2.369	2.350
LavaMD	0.331	0.349	3.733	4.041
LUD	0.220	0.237	5.428	5.207
Particle Filter	0.934	0.916	2.061	1.827
SRAD	1.693	1.725	1.577	1.574
Tiled Matmul	0.100	0.101	10.11	10.912
Benchmark Name	FORASYNC $\frac{R_{cd}}{R_d}$	Approximated $\frac{R_{cd}}{R_d}$	FORASYNC P_c	Approximated P_c
BFS	1.117	1.392	21.73	21.384
B+Tree	1.299	1.306	21.488	20.919
Hotspot	1.531	1.524	21.627	21.237
LavaMD	1.236	1.410	21.237	20.838
LUD	1.195	1.233	22.570	22.074
Particle Filter	1.925	1.674	21.009	20.544
SRAD	2.669	2.716	20.965	20.337
Tiled Matmul	1.010	1.103	21.498	21.140
Benchmark Name	FORASYNC P_d	Approximated P_d	FORASYNC P_{cd}	Approximated P_{cd}
BFS	19.880	19.446	21.097	21.221
B+Tree	20.297	19.636	22.764	23.266
Hotspot	20.201	19.572	22.493	22.746
LavaMD	19.030	18.602	21.663	21.935
LUD	21.068	20.237	24.605	24.618
Particle Filter	19.930	18.598	22.401	21.590
SRAD	19.661	18.750	21.644	21.665
Tiled Matmul	20.848	20.247	23.470	24.065

Table 5.4: Accuracy and Overhead Comparison of FORASYNC_ENERGY

Benchmark Name	Energy Optimal Place	Energy Optimal place as determined by FORASYNC ENERGY	Energy Consumed when executing directly at Optimal Place	Energy Consumed using FORASYNC ENERGY
BFS	ARM+DSP	ARM+DSP	24.226	25.437
LavaMD	ARM+DSP	ARM+DSP	98.540	104.183
LUD	ARM+DSP	ARM+DSP 6 times, DSP 1 time	88.846	88.247
Particle Filter	ARM+DSP	ARM+DSP	675.229	686.164
SRAD	ARM+DSP	ARM+DSP	20.862	20.691
Tiled Matmul	DSP	DSP	126.081	133.091

5.6 Summary

This chapter analyzes the impact of executing the application on different cores and provides a way for automatically selecting the most energy-efficient processors to execute the application on. An API geared towards optimizing for energy was developed along with a technique for automatically determining the most energy optimal place. The technique presented is general and would work with a wide variety of runtimes and heterogeneous systems. The technique also has minimum overheads and was able to save around 5% energy during the profiling run itself.

CHAPTER 6

RELATED WORK

6.1 Parallel Programming Models for Heterogeneous Processors

Heterogeneous multi-core architectures are used to meet the demanding needs (Igual *et al.*, 2012) of data and signal processing intensive applications such as those found in high performance computing areas (Blinka, 2014). Prior work has looked into providing programming models for architectures with heterogeneous cores e.g. CUDA and OpenCL. There have been pragma based approaches for expressing data parallel computations such as the StarS for targeting the IBM cell Processor (Eichenberger *et al.*, 2006) and GPUs (Ayguadé *et al.*, 2009). Nanos++ is a runtime for OmpSs parallel programming model, which is an extension to OpenMP and was developed at the Barcelona Supercomputing Center (Duran *et al.*, 2011). The extensions supported by them includes task reduction, priorities, and SIMD extension. Nanos provides support for coherence across different address-spaces. It provides support for CUDA on Nvidia GPUs, ARM and MALI GPUS. Mitra *et al.* (2014) presented an implementation of the accelerator model which transforms OpenMP directives to OpenCL code. This work was further improved by Aguilar *et al.* (2016) in which they presented a framework which automatically addressed the parallelizing of code, annotating the code with OpenMP 4.0 directives. Capotondi and Marongiu (2016) proposed two constructs for OpenMP 4.0 to expose the clustered organization of many-core architectures, featuring efficient resource utilization for nested parallelism. Chapman *et al.* (2009) explores the goals of an OpenMP based model for heterogeneous systems taking into account priorities, power, etc, with implementation of the work sharing constructs for a DSP based MPSoC. Vargas *et al.* (2016) present an implementation of the OpenMP 4.0 runtime for embedded processors which has a low memory footprint. OpenACC is an alternative programming model for accelerators. The standardization of the of directive based GPU programming was done by OpenACC. Luk *et al.* (2009) presented Qilin which

featured parallelism on heterogeneous Multiprocessor with adaptive Mapping. They got a speedup of 9.3x on an average, over the best serial implementation by judicious distribution of work over CPU and GPU.

6.2 Energy Efficiency on Heterogeneous Processors

The doctoral thesis by Chandramohan develops a framework for execution of OpenMP for MPSoCs. It also featured a power constrained codegen which given a power-budget would partition the code among the different heterogeneous elements (Chandramohan, 2016). Panyala *et al.* (2017) looked at performance energy trade-offs of applications with irregular data accesses. Cabrera *et al.* (2015) implemented EML (energy measurement library) to measure energy consumption in HPC systems. LaKomski *et al.* (2015) showed that energy and performance in HPC applications can be conflicting goals and a trade-off between them varies with the characteristics of the runtime implementation and the applications executing. Balaprakash *et al.* (2014) give a formalism for multi objective optimization for power, time and energy. According to their work in some cases the objectives are strictly correlated and there is a single ideal decision point; in others, significant trade-offs exist. In Chandramohan and O'Boyle (2014) a framework for partitioning in heterogeneous parallel programs is presented, along with a method for accurately measuring runtime and energy usage.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

MPSoC has started a new computing era but brought a twofold challenge in building software: how to fully exploit the performance of different processing elements and how to achieve an execution with lower energy footprint. Furthermore, building such a software without hurting the programmer's productivity is another daunting task.

This thesis explores the challenge of achieving high performance and energy-optimal execution by using a work-stealing runtime for MPSoC that can abstract away all the hardware complexities and provide an easy-to-use parallel programming environment.

We present a quantitative analysis of the overheads associated with a hybrid work-stealing runtime for Keystone-II Hawking MPSoC. We identify that taking a lazy approach to gather results of partial computations running at different processing elements in MPSoC can significantly boost the parallel performance. We used this insight to develop a novel hybrid work-stealing runtime for Keystone-II Hawking MPSoC that lazily flushes the caches of cache-coherent ARM and cache-incoherent DSP cores to gather the results of computations running at different cores without incurring any significant overheads. We demonstrate that this runtime design can boost the performance of both recursive divide-and-conquer and `for` loop based parallelism.

We further tackle the issue of automatically determining the most energy-optimal place to execute the application on MPSoCs by introducing an automated approach that relies on our novel hybrid work-stealing runtime. We show that by using this low overhead approach it is possible to find the energy optimal place of execution directly during the actual execution, without the need for doing multiple trial runs.

In combination, these contributions demonstrate that parallel programming over MPSoCs can be made significantly easier and provide further hope for runtimes aimed

to improve programmer's productivity, where the hard-work of optimization of resource utilization is left to the runtime by the programmer.

7.2 Future Work

The current model for reducing cache flushes does not take any advantage of timing information. One could say that if a task has been generated by a processor before the start of a cache write-back-invalidate and if that cache write-back-invalidate has completed, then all data pertaining to that task has already been made available to the memory. Now, if that task has to be given to some other cache-incoherent processor then no write-back-invalidate operation needs to be carried out. This could further increase the max attainable speedup, as the tasks are sent from the top of the queue and are likely to have been flushed already. However, this would also require more synchronization during task creation and cache write-backs.

Another issue is that due to the focus on the ease of use of the model, it cannot determine whether a cache-writeback-invalidate before sending a task is actually required or not. Some codes, like mergesort, can write-back the data before the start of the first mergesort call and do not need to do a write-back before transferring a task. They only need to invalidate the pre-fetch during END_FINISH. However, our current model cannot take advantage of this feature. Combining both timing and structural properties to minimize cache-write-backs would result in significantly lower cache flushes and higher speedups.

One other way of reducing cache-writebacks would be to determine exactly what tasks should be given to DSPs to keep them busy for as long as possible to reduce the number of steals. Since the rate of execution of both the processors is different, the benchmarks end up possessing some characteristics of irregular benchmarks and the number of steals is significantly higher compared to work-stealing in homogeneous environments. A multi-steal strategy based on the rate of computation of different processors could help here. Combining the above optimizations to implement the tasking model in OpenMP on the Keystone-II Hawking MPSoC is also an interesting direction for future work due to the ease of use, portability and the popularity offered by OpenMP.

In our case, gaining energy efficiency from executing only at only one of the pro-

cessors was turning out to be difficult, as it would require one of them to significantly out-perform the other or for the computation to get bounded in speed due to memory bandwidth. The main reason for this anomaly is the high energy consumption of other system components compared to that of processors. Thus, it will be more suitable to evaluate the energy-model on systems where power consumption of processors is the dominating factor. It would also be interesting to see if the model can be extended for nested `async–finish` parallelism to create a more feature complete runtime.

The current work also had its limitations. There was no support for optimizing irregular loops. Several techniques can be devised to handle this if we are able to identify the irregularity of the loop. The simplest would be to execute the same iteration of the loop on both the processors, but one of the processors should write the result to dummy location which would later be discarded. Also, energy optimization functions other than processor selection were not considered. The processor selection itself was not done on the basis of optimization of other functions but only on the raw characteristics of the application. Investigation of other optimization functions could lead to a more energy-optimal approach. Further, repeated profiling or learning the characteristics of the application for improving the predictions was also not considered, as the goal was to keep the profiling overheads at a minimum during the execution of parallel `for` loop. Depending upon the requirements from the run-time, this could be an interesting area for further research.

APPENDIX A

SOFTWARE INSTALLATION ON TI KEYSTONE-II

MCSDK provides readily configured tools, softwares, libraries and linux operating system for Keystone-II Hawking MPSoC to allow rapid deployment and evaluation of applications. We used MCSDK-HPC 03_00_01_12 for development and evaluation of the work in this thesis. The steps followed for installation of MCSDK-HPC using a 64-bit Ubuntu host are given below. More detailed version of these steps are also present in MCSDK-HPC SETUP

A.1 Install MCSDK

1. The version of MCSDK given in the MCSDK-HPC download page should be used
2. sudo apt-get install ia32-libs
3. chmod +x mcsdk_native<version>_native_setuplinux.bin
4. ./mcsdk_native<version>_native_setuplinux.bin

A.2 Install MCSDK-HPC

1. chmod +x mcsdk-hpc_<version>_setuplinux.bin
2. ./mcsdk-hpc_<version>_setuplinux.bin

A.3 Install TFTP server

1. sudo apt-get install xinetd tftpd tftp
2. Create /etc/xinetd.d/tftp
3. Append the following to the above file
   service tftp
   {
   protocol = udp

```
port = 69
socket_type = dgram
wait = yes
user = nobody
server = /usr/sbin/in.tftpd
server_args = /tftpboot
disable = no
}
```

4. mkdir /tftpboot
5. sudo mkdir /tftpboot
6. sudo chmod -R 777 /tftpboot
7. sudo chown -R nobody /tftpboot
8. sudo /etc/init.d/xinetd restart
9. sudo service xinetd restart

A.4 Install NFS Server

1. sudo apt-get install nfs-kernel-server
2. sudo service nfs-kernel-server restart

A.5 EVM Setup

A.5.1 Basic setup of EVM

1. Make sure ethernet cable is connected to ENET0 port
2. Set the boot mode to SPI Little endian Boot mode, i.e., only switch 3 on for rev 1.0 EVM.
3. Attach the serial port cable to the SoC UART port
4. Connect the power cable

A.5.2 Install Terminal Software on PC for Serial Port Connection

1. Obtain EVM serial device by using dmesg | grep tty. The higher number is BMC (usually /dev/ttyUSB1) and the lower number is the SoC (usually /dev/ttyUSB0).
2. sudo apt-get install picocom

3. picocom -b 115200 /dev/ttyUSB0
4. Power up the EVM

A.5.3 Bring up Linux on EVM using an NFS filesystem

1. Program or Upgrade Uboot image
   - Copy Uboot image (u-boot-spi-keystone-evm.gph) in mcsdk/images directory to /tftpboot
   - Use putty to connect to SoC and type the below steps in Uboot prompt
   - env default -f -a
   - setenv serverip 192.168.1.10
   - setenv tftp_root <tftp root directory>
   - setenv name_uboot <name of u-boot gph image under the tftp root>
   - run get_uboot_net
   - run burn_uboot
2. Copy Required Linux Kernel images
   - The below images are present in the images folder in MCSDK directory
   - cp skern-keystone-evm.bin /tftpboot
   - cp uImage-k2hk-evm.dtb /tftpboot or cp k2hk-evm.dtb. If uImage-k2hk-evm.dtb is not present then k2hk-evm.dtb can be used instead.
   - cp uImage-keystone-evm.bin /tftpboot
3. Create a folder for EVM in host machine (/evmk2h_nfs for this appendix). Avoid making this folder in root folder of host desktop.
4. sudo tar xvf tisdk-rootfs.tar.gz -C /evmk2h_nfs
5. Edit /etc/exports to add /evmk2h_nfs *(rw,subtree_check,no_root_squash,no_all_squash,sync)
6. sudo service nfs-kernel-server restart
7. Set the Environment variables in Uboot of the EVM
   - Once again logon to SoC using picocom and type the below commands in Uboot prompt
   - env default -f -a
   - setenv boot net
   - setenv mem_reserve 1536M
   - setenv gatewayip [gateway address on which host PC and EVM are present]
   - setenv serverip [IP address of the host]
   - setenv tftp_root /tftpboot
   - setenv name_fdt uImage-k2hk-evm.dtb
   - setenv name_kern uImage-keystone-evm.bin

- setenv nfs_root /evmk2h_nfs
- setenv nfs_serverip [IP address of the host containing nfs root]
- saveenv

8. Power cycle the EVM

A.6 Software Installation on File System of the EVM

Install the following packages after the above steps are complete.

A.6.1 Install HPC IPKs and CGTools on to the EVM

- cd /evmk2h_nfs/home/root
- sudo mkdir install
- cd install
- sudo cp [mcsdk-hpc install dir]/mcsdk_hpc_<version>/images/*.ipk .
- sudo cp [mcsdk-hpc install dir]/mcsdk_hpc_<version>/images/*.sh .
- Logon to SoC using putty or ssh
- cd /home/root
- chmod +x install_hpc_packages_evm.sh
- ./install_hpc_packages_evm.sh
- cd ..
- rm -rf install

A.6.2 Install cross compilation tools

- sudo apt-get install libelf1:i386
- Download and install Linaro toolchain
- sudo apt-get install qemu qemu-user-static binfmt-support mesa-common-dev binutils-dev flex u-boot-tools

A.6.3 Setup HPC environment

- Modify variables in <mcsdk-hpc_install_dir>/scripts/setup_hpc_env.sh as needed to point to install location of MCSDK 3.0.x, MCSDK-HPC, TI CGTools, and etc on the Desktop Linux PC. source ./setup_hpc_env.sh
- ./install_devkit.sh
- sudo -E ./install_devfs.sh

A.6.4 Adding newer CMEM API

By default the CMEM API in the installed devkit may not have the entire cache writeback-invalidate function. It may only have range based functions which could take huge amounts of time for large arrays. To get functions which writeback-invalidate the entire cache a newer CMEM API must be used. The methods below provide ways for using the newer API:

- Compile the code on EVM itself
- Copy the CMEM libraries from EVM to devkit folder on host to get the new API and to use cross compilation
- Install the HPC devkit on the host also

REFERENCES

1. **Acar, U. A.**, **A. Chargueraud**, and **M. Rainey** (2013). Scheduling parallel programs by work stealing with private deques. *SIGPLAN Not.*, **48**(8), 219–228. ISSN 0362-1340. URL http://doi.acm.org/10.1145/2517327.2442538.

2. **Aguilar, M. A.**, **R. Leupers**, **G. Ascheid**, and **L. G. Murillo**, Automatic parallelization and accelerator offloading for embedded applications on heterogeneous mpsocs. *In Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016.

3. ARM Cortex A15 MPCore Processor Technical Reference Manual (). Arm cortex a15 mpcore processor technical reference manual. URL https://developer.arm.com/docs/ddi0438/latest/preface.

4. ARM Floating Point (). Floating point operations in arm. URL https://developer.arm.com/technologies/floating-point.

5. **Ayguadé, E.**, **R. M. Badia**, **F. D. Igual**, **J. Labarta**, **R. Mayo**, and **E. S. Quintana-Ortí**, An extension of the starss programming model for platforms with multiple gpus. *In Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09. Springer-Verlag, 2009.

6. **Balaprakash, P.**, **A. Tiwari**, and **S. M. Wild**, *Multi Objective Optimization of HPC Kernels for Performance, Power, and Energy*. Springer International Publishing, Cham, 2014. ISBN 978-3-319-10214-6, 239–260. URL http://dx.doi.org/10.1007/978-3-319-10214-6_12.

7. **Blinka, E.**, Programming TI KeyStone II-based ARM + DSP devices using industry standard tools. *In 2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 2014.

8. **Blumofe, R. D.** and **C. E. Leiserson** (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, **46**(5), 720–748. ISSN 0004-5411. URL http://doi.acm.org/10.1145/324133.324234.

9. **Cabrera, A.**, **F. Almeida**, **J. Arteaga**, and **V. Blanco** (2015). Measuring energy consumption using eml (energy measurement library). *Computer Science - Research and Development*, **30**(2), 135–143. ISSN 1865-2042. URL http://dx.doi.org/10.1007/s00450-014-0269-5.

10. **Capotondi, A.** and **A. Marongiu**, On the effectiveness of openmp teams for cluster-based many-core accelerators. *In High Performance Computing & Simulation (HPCS), 2016 International Conference on*. IEEE, 2016.

11. **Cavé, V.**, **J. Zhao**, **J. Shirako**, and **V. Sarkar**, Habanero-java: The new adventures of old x10. *In Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0935-6. URL http://doi.acm.org/10.1145/2093157.2093165.

12. **Chandramohan, K.** (2016). *Mapping parallelism to heterogeneous processors*. Ph.D. thesis.

13. **Chandramohan, K.** and **M. F. O'Boyle**, Partitioning data-parallel programs for heterogeneous mpsocs: Time and energy design space exploration. *In Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2877-7. URL `http://doi.acm.org/10.1145/2597809.2597822`.

14. **Chapman, B.**, **L. Huang**, **E. Biscondi**, **E. Stotzer**, **A. Shrivastava**, and **A. Gatherer**, Implementing openmp on a high performance embedded multicore mpsoc. *In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009.

15. **Charles, P.** *et al.*, X10: An object-oriented approach to non-uniform cluster computing. *In OOPSLA*. 2005. ISBN 1-59593-031-0.

16. **Chase, D.** and **Y. Lev**, Dynamic circular work-stealing deque. *In Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05. ACM, New York, NY, USA, 2005. ISBN 1-58113-986-1. URL `http://doi.acm.org/10.1145/1073970.1073974`.

17. **Che, S.**, **M. Boyer**, **J. Meng**, **D. Tarjan**, **J. W. Sheaffer**, **S. H. Lee**, and **K. Skadron**, Rodinia: A benchmark suite for heterogeneous computing. *In 2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009.

18. **Duran, A.**, **E. Ayguadé**, **R. M. Badia**, **J. Labarta**, **L. Martinell**, **X. Martorell**, and **J. Planas** (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, **21**(02), 173–193.

19. **Eichenberger, A. E.**, **J. K. O'Brien**, **K. M. O'Brien**, **P. Wu**, **T. Chen**, **P. H. Oden**, **D. A. Prener**, **J. C. Shepherd**, **B. So**, **Z. Sura**, **A. Wang**, **T. Zhang**, **P. Zhao**, **M. K. Gschwind**, **R. Archambault**, **Y. Gao**, and **R. Koo** (2006). Using advanced compiler technology to exploit the performance of the cell broadband engine x2122; architecture. *IBM Systems Journal*, **45**(1), 59–84.

20. **Frigo, M.**, **C. E. Leiserson**, and **K. H. Randall**, The implementation of the Cilk-5 multithreaded language. *In PLDI*. 1998. ISBN 0-89791-987-4.

21. **Guo, Y.**, **R. Barik**, **R. Raman**, and **V. Sarkar**, Work-first and help-first scheduling policies for async-finish task parallelism. *In IPDPS '09*. 2009. ISBN 978-1-4244-3751-1.

22. HClib (). Hclib. URL `http://habanero-rice.github.io/hclib/`.

23. **Igual, F. D.**, **M. Ali**, **A. Friedmann**, **E. Stotzer**, **T. Wentz**, and **R. A. van de Geijn**, Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC. *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society Press, 2012.

24. **Imam, S.** and **V. Sarkar**, Habanero-java library: A Java 8 framework for multicore programming. *In PPPJ*. 2014. ISBN 978-1-4503-2926-2.

25. Intel Software Developer Zone (). Avoiding and identifying false sharing among threads. URL `https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads`.

26. **Kumar, V.**, **D. Frampton**, **S. M. Blackburn**, **D. Grove**, and **O. Tardieu**, Work-stealing without the baggage. *In OOPSLA*. 2012. ISBN 978-1-4503-1561-6.

27. **Kumar, V.**, **A. Sbirlea**, **A. Jayaraj**, **Z. Budimlic**, **D. Majeti**, and **V. Sarkar**, Heterogeneous work-stealing across cpu and dsp cores. *In 2015 IEEE High Performance Extreme Computing Conference (HPEC)*. 2015.

28. **Kumar, V.**, **Y. Zheng**, **V. Cavé**, **Z. Budimlić**, and **V. Sarkar**, HabaneroUPC++: A compiler-free PGAS library. *In PGAS '14*. 2014. ISBN 978-1-4503-3247-7.

29. **LaKomski, D.**, **Z. Zong**, **T. Jin**, and **R. Ge**, Optimal balance between energy and performance in hybrid computing applications. *In Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*. IEEE, 2015.

30. **Lea, D.**, A Java Fork/Join framework. *In JAVA*. 2000. ISBN 1-58113-288-3.

31. **Luk, C.-K.**, **S. Hong**, and **H. Kim**, Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. *In Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42. 2009.

32. MCSDK-HPC (). Mcsdk-hpc download page. URL `http://software-dl.ti.com/sdoemb/sdoemb_public_sw/mcsdk_hpc/latest/index_FDS.html`.

33. MCSDK-HPC SETUP (). Hpc (high performance computing) development tools for mcsdk. URL `http://processors.wiki.ti.com/index.php/MCSDK_HPC_3.x_Getting_Started_Guide`.

34. **Mitra, G.**, **A. Haigh**, **A. Varghese**, **L. Angove**, and **A. P. Rendell**, Split wisely: When work partitioning is energy-optimal on heterogeneous hardware. *In 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2016.

35. **Mitra, G.**, **E. Stotzer**, **A. Jayaraj**, and **A. P. Rendell**, Implementation and optimization of the openmp accelerator model for the ti keystone ii architecture. *In* **L. DeRose**, **B. R. de Supinski**, **S. L. Olivier**, **B. M. Chapman**, and **M. S. Müller** (eds.), *Using and Improving OpenMP for Devices, Tasks, and More*. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11454-5.

36. **nCore HPC** (2015). ncore browndwarf supercomputer. "http://ncorehpc.com/pdf/ncorehpc_browndwarf_y-class.pdf".

37. **Packard, H.** (2014). Hp moonshot system. "https://www.hpe.com/us/en/product-catalog/servers/proliant-servers.hits-12.html".

38. **Panyala, A.**, **D. Chavarria-Miranda**, **J. B. Manzano**, **A. Tumeo**, and **M. Halappanavar** (2017). Exploring performance and energy tradeoffs for irregular applications: A case study on the tilera many-core architecture. *Journal of Parallel and Distributed Computing*, **104**, 234 – 251. ISSN 0743-7315.

39. TMS320C66x DSP Cache User Guide (). Tms320c66x dsp cache user guide. URL http://www.ti.com/lit/ug/sprugy8/sprugy8.pdf.

40. TMS320C66x DSP CorePac User Guide (). Tms320c66x dsp corepac user guide. URL http://www.ti.com/lit/ug/sprugw0c/sprugw0c.pdf.

41. **top500.org** (2017). https://www.top500.org/news/china-will-deploy-exascale-prototype-this-year/.

42. **Vargas, R. E.**, **S. Royuela**, **M. A. Serrano**, **X. Martorell**, and **E. Quinones**, A lightweight openmp4 run-time for embedded systems. *In Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*. IEEE, 2016.