Alma Mater Studiorum - Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea in
Metodologie di Progettazione Hardware - Software M

OpenMP Task Scheduling Strategies to Mitigate Hardware Variability in Tightly-Coupled Shared Memory Clusters

Candidato:

Daniele Cesarini

Relatore:

Chiar. mo Prof. Ing. **Luca Benini**

Correlatori:

Chiar. mo Prof. Ing. **Rajesh K. Gupta**

Dott. Ing. **Andrea Marongiu**

Dott. Ing. **Abbas Rahimi**

Anno Accademico 2013-2014
Sessione II

Abstract

Nell'ultimo decennio l'avvento delle architetture multi-processore su singolo chip (MPSoC) ha consentito ai progettisti di superare numerose barriere tecnologiche. La tendenza ad integrare un numero sempre maggiore di processori sullo stesso chip è tuttora predominante, sia in ambito general-purpose che embedded computing, e ci ha condotti all'era dei *many-cores*. Questa evoluzione ha portato a delle conseguenze non trascurabili nell'uso di questi sistemi. Lo sfruttamento efficiente del potenziale computazionale di queste architetture massivamente parallele ha introdotto una nuova serie di problemi a cui la comunità scientifica cerca di rispondere con l'utilizzo misto di tecniche hardware e software. Tra i principali problemi di scalabilità per queste architetture troviamo l'interconnessione tra unità di calcolo e la memoria. La gerarchizzazione della memoria e l'accoppiamento stretto tra processori è un problema ancora aperto su cui la comunità scientifica sta lavorando. Una tipica risposta a questi problemi è il raggruppamento in *cluster* di risorse computazionali localmente vicine per sfruttare il principio di località.

Se da un lato un sistema di interconnessione gerearchico consente la scalabilità, dall'altro, unito ad un parallelismo di centinaia di unità computazionali all'interno dello stesso chip, porta alla necessità di una profonda modifica dei modelli di programmazione. Sviluppare software che siano adatti a questo genere di architetture porta difficoltà nella programmazione e nella gestione dei flussi di esecuzione di ogni singolo processore. Avere molte unità computazionali porta anche a dei problemi di condivisione delle risorse, come ad esempio la memoria e le varie periferiche di input/output disponibile nella piattaforma. Per sopperire a questi problemi, i *modelli di programmazione*, e i *sistemi di runtime* associati, che gestiscono questi processori si sono evoluti a tal punto da diventare dei componenti essenziali al funzionamento di queste piattaforme.

Ad esacerbare la difficoltà di utilizzo dei manycores basati su *cluster* si aggiunge il problema della *variability*, che affligge i moderni sistemi prodotti con tecnologia nanometrica. I difetti, o in generale la variability, introduce eterogeneità nel sistema, perchè dei core nominalmente identici per funzionare nella stessa maniera hanno bisogno di operare con voltaggi e frequenze diversi. Per evitare di utilizzare dei margini troppo conservativi, che fanno perdere in performance ed energia, si possono usare circuiti di controllo d'errore, e tecniche di correzione degli stessi. Lo svantaggio è che comunque la gestione puramente hardware degli errori ha un costo. Questo costo si può ridurre se il sofware capisce quale particolare attivazione dei datapath fa scaturire questi errori (i.e., quale flusso di istruzioni) e ne minimizza la probabilità con la nostra tecnica.

Il primo contributo che porta questa tesi è stato l'estensione del modello di programmazione. OpenMP 3.0, che supporta la gestione di flussi di esecuzioni paralleli irregolari e dinamici (tasking). Un tipico esempio possono essere gli algoritmi ricorsivi o la ricerca in grafi come gli alberi. Abbiamo implementato un runtime ottimizzato per la gestione di questo modello di programmazione per acceleratori embedded con processori strettamente accoppiati in cluster e poi interconnessi attraverso una network on chip. Ci siamo focalizzati sulla loro scalabilità che è il requisito fondamentale richiesto in questo genere di acceleratori e sul supporto di task di granular-

ità fine, come è tipico nelle applicazioni embedded. Tramite i meccanismi proposti abbiamo raggiunto un buon livello di scalabilità. In particolare, sia nell'architettura single cluster che in quella multi cluster abbiamo raggiunto un speedup di 84% già con una dimensione di 5.000 operazioni ALU per task. Entrambe le soluzioni raggiungono questo speedup a questa granularità, questo significa che il runtime multi cluster ha la stessa scalabilità del single cluster avendo però x4 volte il numero dei processori e un livello in più di interconnessione (NoC).

Per cercare di minimizzare i problemi dati da fenomini di variability, il secondo contributo di questa tesi è stata proporre una estensione del runtime di OpenMP che cerca di prevedere la manifestazione di questi errori tramite una schedulazione efficiente del carico di lavoro. I core soggetti a questo fenomeno fisico formano quindi un sistema manycore eterogeneo. Ogni core è affetto da variability dipendentemente dal tipo di unità funzionali compromesse. All'interno del nostro ambiente di simulazione assumiamo un modello di errore che rappresenta gli effetti della variability come eventi che si manifestano con una certa probabilità sui vari path della logica (i.e., sul datapath). La metodologia di caratterizzazione del RTL di un core SPARC a diversi operating points da cui abbiamo derivato le probabilità di errore. Tramite il tasking model di OpenMP usiamo questo modello di errore per definire unità di lavoro rappresentate come dei task che poi vengono caratterizzati per ogni core. Lo scheduler che abbiamo integrato in OpenMP utilizzata questi metadati per una schedulazione efficiente (i.e., che causi meno errori) del carico di lavoro che l'acceleratore deve eseguire Infine tramite un esteso set di benchmark abbiamo poi valutato l'efficienza della nostra soluzione in cui abbiamo raggiunto in media una esecuzione il 22% più veloce e del risparmio nel consumo di energia del 35%.

Alla mia famiglia.

Contents

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Work Motivation 1

1.1.1 Heterogeneous Embedded Systems Equipped with Many-Core Cluster-Based Accelerators 2

1.1.2 Parallel Programming Models 2

1.1.3 Variability in Many-Core Cluster-Based Processors . 3

1.1.4 OpenMP Tasking Model and Variability 4

1.2 Overview of the Thesis 4

2 Target Architecture 7

2.1 Cluster 7

2.1.1 Processing Units 8

2.1.2 Tightly Coupled Data Memory - L1 Memory 8

2.1.3 Logarithmic interconnect 9

2.1.4 Cluster Components 10

2.2 Multi Cluster 11

2.3 Host-Accelerator Interface 11

3 Tasking support for embedded many cores 13

3.1 Introduction 13

3.2 Related works 15
3.3 OpenMP tasking model 16
3.4 Single cluster 17
3.4.1 Design and Implementation 17
3.4.2 Experimental Results 19
3.5 Multi Cluster 21
3.5.1 Experimental Results 22

4 Variability-aware OpenMP **25**
4.1 Introduction 25
4.2 Related Works 26
4.3 Architectural support for variation-affected processors . . . 27
4.4 Work-unit vulnerability and VOMP work-sharing 28
4.4.1 WUV corner cases 31
4.4.2 Run-time WUV Characterization 38
4.5 VOMP schedulers 39
4.5.1 Variation-Aware Task Scheduling (VATS) 39
4.5.2 Variation-Aware Section Scheduling (VASS) 43
4.6 Experimental results 44
4.6.1 Results for Tasking 44
4.6.2 Results for Sections 47

5 Conclusion **51**
5.1 Tasking Conclusions 51
5.2 Variability Conclusions 52

Bibliography **53**

List of Figures

2.1 Cluster 8
2.2 Mesh of trees 4x8 (banking factor of 2) 10

3.1 Single cluster scalability varying task granularities. 20
3.2 Multi cluster scalability varying task granularities. 23

4.1 EDS e VDD-Hopping. 28
4.2 Outlined WU types in a OpenMP program: task, section, for. 29
4.3 WU types each stressing a different class of instructions. . 31
4.4 Synthetic benchmark using OpenMP task. 32
4.5 Normalized WUV (NWUV) to temperature variations for task types. 33
4.6 Normalized WUV to voltage variations for task types. 33
4.7 Software pipelined synthetic benchmark using OpenMP sections. 35
4.8 Normalized WUV to temperature variations for sections types. 35
4.9 Normalized WUV to voltage variations for sections types. . 36
4.10 Pseudo-code for task-level WUV characterization. 38
4.11 Distributed queues for OpenMP tasking. 40
4.12 Execution time for VATS normalized to RRS under temperature variation. 45
4.13 Energy consumption for VATS normalized to RRS under temperature variation. 46
4.14 Execution time for VASS normalized to FCFS under temperature variation. 48

4.15 Energy consumption forVASS normalized to FCFS under temperature variation. 49

List of Tables

2.1 Latency time between core and different memories without concurrency 11

3.1 Architectural parameters for single cluster. 19
3.2 Architectural parameters for multi cluster. 22

4.1 Architectural parameters of the cluster. 44

Chapter 1

Introduction

1.1 Work Motivation

Before the advent of parallel architectures, processor development was based on increasing the clock frequency and the instruction throughput that a single core was able to perform. While the Moore's law continues to be valid, the power wall for which a transistor cannot increase his power and clock frequency have determinate a deeply change how system architectures are realized. Before reaching this limit, processor manufacturing was based on increasing these two factors to maintain the growth of performance. Afterwards, engineers had to focus on how to increase the number of instructions executed by employing multiple processors on the same time. This approach opened the era of parallel computing, where roday hundres to thousands of processing units are integrated in the same processor.

Moreover, the nanotechnology era in semiconductor circuits also brings side effects during the manufacturing of these chips. These side effects broadly go under the name of "variability" and the most common effect is violation of timing specification. Nowadays, these behaviors are very common and variability is an active field of research where approaches are being explored for mitigating timing errors through hardware/software solutions.

1.1.1 Heterogeneous Embedded Systems Equipped with Many-Core Cluster-Based Accelerators

Embedded systems have been revolutionized by the emergence of new parallel architectures with a very large number of processing units.

These embedded many-core accelerators have found a very profitable area where they could be used to increase performance and reduce energy consumption of computation intensive and highly-parallel code kernels. Heterogeneous systems which combine a general-pourpese “host” processor to an accelerator with a high parallelism compared to a traditional single powerful processor are nowadays common on high-end embedded. Conversely, they need a new programming model. A single-core programming model is based on a two simple concepts: i) having a huge private memory ii) and a single processing unit. Extensions required in embedded many-core programming stem from their most peculiar architectural features: the high number of parallel threads enabled and the use of an explicitly-managed memory hierarchy with non-uniform access (NUMA). The main difference between single core and many-core cluster-based accelerator is the high number of processing units that they have and a non-caching data hierarchy memory to minimize delays due to physical distance. On the downside, these systems require traditional programming models to be significantly revisited and extended.

1.1.2 Parallel Programming Models

Parallel architectures are complex and is not a trivial task to exploit them in a efficient way. Moreover, modern embedded applications often expose a high degree of parallelism.

Enclosing parallelism resources in a software abstraction that provides an easy and coherent runtime layer is a key aspect to achieving performance and platform usability. Consequently, several runtime systems have been proposed to support parallel programming models. Different runtime have been proposed for different scenario, but we can divide them in two

large families: based on shared memory and on message passing.

For embedded system, researchers have mostly focused on shared-memory, driven by the nature of their hardware architecture (mainly based on a multi-level explicitly managed (i.e. non cache-based) shared memory). Runtime layers have a wide impact on performance, thus minimizing overheads is a primary challenge. Embedded systems have always had constrained resources and typically runtime support implemented as lightweight middleware running on bare. A good runtime must have a deep knowledge of underlying hardware and should expose a powerful set of APIs to abstractive parallel resources. The main challenges in designing an efficient runtime for an embedded manycore are i) providing an interface that is rich enough to effectively use all the underlying hardware potential and ii) considering a streamòined implementation that minimizes the overheads.

1.1.3 Variability in Many-Core Cluster-Based Processors

While scaling of physical dimensions of CMOS in semiconductor circuit opens the way to many-core processor chips, variability problems within die grow as well. Variation is manifest from different physical sources and has static and dynamic components. Static variations manifest themselves as manufacturing problems, cores across the wafer have different working frequency. These induce a fixed mismatch of performance. Dynamic variation changes during run-time based on the environment where cores are used, typical examples are dynamic voltage drops and temperature fluctuations within die.The mainly manifestation of variability is violation of timing specification caused of circuit-level errors. For this reason an important aspect is having a robust system design and recovery mechanism to ensure error-free completion of the errant instructions.

Variability problems induce asymmetric behavior in fabricated chips that can bring unbalancing computational workloads if cores are considered as all equal. Moreover, having a balanced workload is not a trivial task because variation can change dynamically at run time, thus scheduling of

parallel tasks must take into account the type of workload.

1.1.4 OpenMP Tasking Model and Variability

OpenMP is a shared memory programming model consisting of a collection of compiler directives and library routines to allow C, C++, and Fortran developers to write multi-threaded applications. For this reason, OpenMP has been very successful on exploiting parallel architectures making them easily available. Until specification version 2.5, it has been focused on loop-level parallelism where each iteration can be independently performed.

Given the increasing complexity of applications and their irregular and dynamic structure, OpenMP has recently provided a set of new directives to improve flexibility to the previously loop-centric nature. This model is called ”tasking” and it has been embodied in OpenMP since specification 3.0. Different parallel programming models are based on tasking like CILK [37] and Intel Threading Building Blocks [45]. Tasking allows programmers to specify independent units of work that can be performed asynchronously, independently of the execution flow. When a thread is ready to perform a task, it asks the run-time library to get one. Explicit synchronization constructs to guarantee completion of tasks have been incorporated in the specification, plus some implicit synchronization points in the runtime environment.

Tasking is a good candidate to characterize different workloads, this approach allows easy integration of a new scheduling strategies to mitigate variability based on the status of cores by nclosing a set of lines of code in a pragma directive, is possible to define a new task and choose what is the best core where this workload must be performed.

1.2 Overview of the Thesis

This document describes a implementation of the OpenMP specification 3.0 for an embedded cluster-based many-core processor focusing on two

aspects: i) tasking for single and multi-cluster architectures and ii) how to mitigate variability problems using a software approach based on the OpenMP runtime.

In chapter 2, we analyze our main target architecture based on a many-core cluster-based processor, moreover we describe the different simulation environments that we used. Here, we describe the two architectures that we use to study variability and tasking model implementation problems.

Chapter 3 takes in exam support of tasking model for OpenMP in a embedded many-core processor. We describe the starting implementation for single-cluster architecture on which we based our work and how we extend the runtime to support many-cores. After that, we discuss how we integrate our tasking framework with *nested parallelism* to exploit the clustering of data. We talk about our experiments and how can vary the efficiency of our infrastructure having a nested-aware runtime.

Chapter 4 explores our OpenMP extensions to mitigate hardware variability in a single-cluster environment, using various parallel constructs, including `task`, `section` and `for` loop. We present here a complete software solution to handle variability issues hiding the heterogeneity caused by variability effects. We demonstrate the effectiveness of our approach with a large set of benchmarks and a different granularity of variation-affected cores.

The last part of the thesis concludes the work presenting final results.

Chapter 2

Target Architecture

In this chapter we describe our architectures used for experimental. We use a SystemC-based virtual platform, we take in exam two different architectures. First, it is a single cluster equipped with sixteen cores where we can simulate variation behavioral changing voltage clock and temperature degree independently for each core. We use this architecture to present our implementation for tasking model in the first part of Chapter 3, and even in the Chapter 3 to validate our methodology to mitigate variation effects.

Second, we have a multi-cluster architecture inspired by STMicroelectronics P2012 [13] that we use to submit our tasking support for multi-cluster processors and we deeply describe in the second part of Chapter 3.

2.1 Cluster

In this section we submit prevalent components of our target architecture considered in this this work. The main use that we do with single-cluster virtual platform is to exploit variability problems that afflict nanometric technology in modern processors.

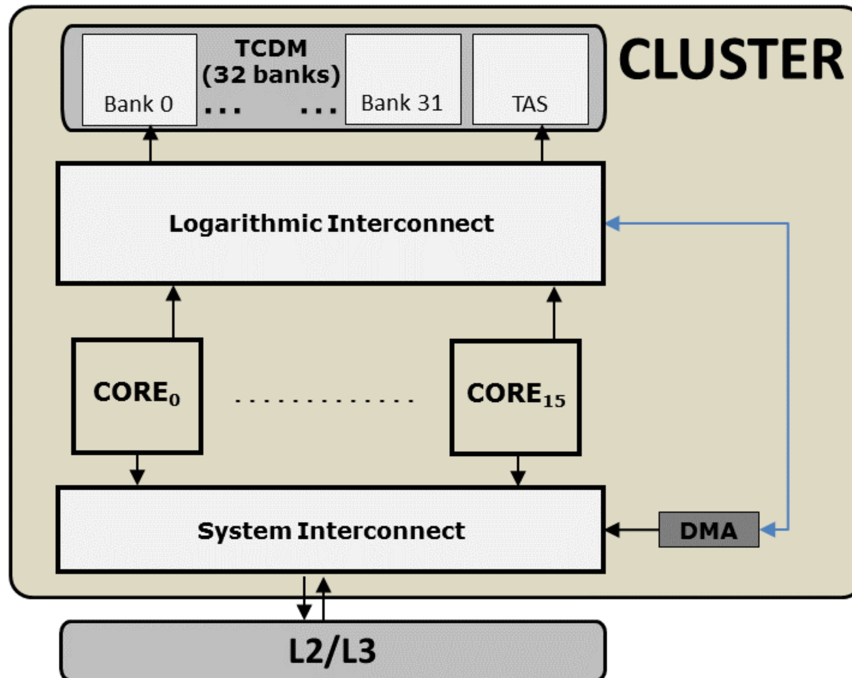

Figure 2.1: Cluster

2.1.1 Processing Units

Each single cluster contains up to sixteen 32-bit in-order ARMv6 processors. This processors are based on RISC LEON-3 [20] core with technology on 45-nm. All cores have Harvard architecture, so they have a private instruction cache with 16 KB of memory and no data cache. This is a set-associative cache and works in coupled with a large L3 memory where are stored all program instructions.

2.1.2 Tightly Coupled Data Memory - L1 Memory

Cores communicate with a multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM) through a logarithmic interconnection. Each bank

has a access port and we equip with K banks, where k is a multiple number of the cores. TCDM supports up to sixteen concurrency transaction, so all cores can access memory in the same clock cycle if memory addresses are in different banks. To minimize concurrence problems, banks have interleaved memories at the word-level to avoid memory transaction challenge in logically contiguous data structures. TCDM has only 256 KB size of memory, the rest of data and instructions are typically stored in larger L2 or L3 memory.

In this architecture, to guarantee synchronization among cores, has been equipped with a test-and-set memory on a dedicate bank. This memory return the content of the target memory location and updates the value in a single clock cycle (atomically). Hence, test-and-set memory can be accessed with standard read/write memory operations as any other memory address.

2.1.3 Logarithmic interconnect

Cores communicate with TCDM through a low-latency high-bandwidth logarithmic interconnection built as a parametric, fully combinational Mesh-of-Trees (MoT) design [14]. MoT allows 1-cycle L1 access, this behavior is compatible with pipeline depth for load/store for most processors.

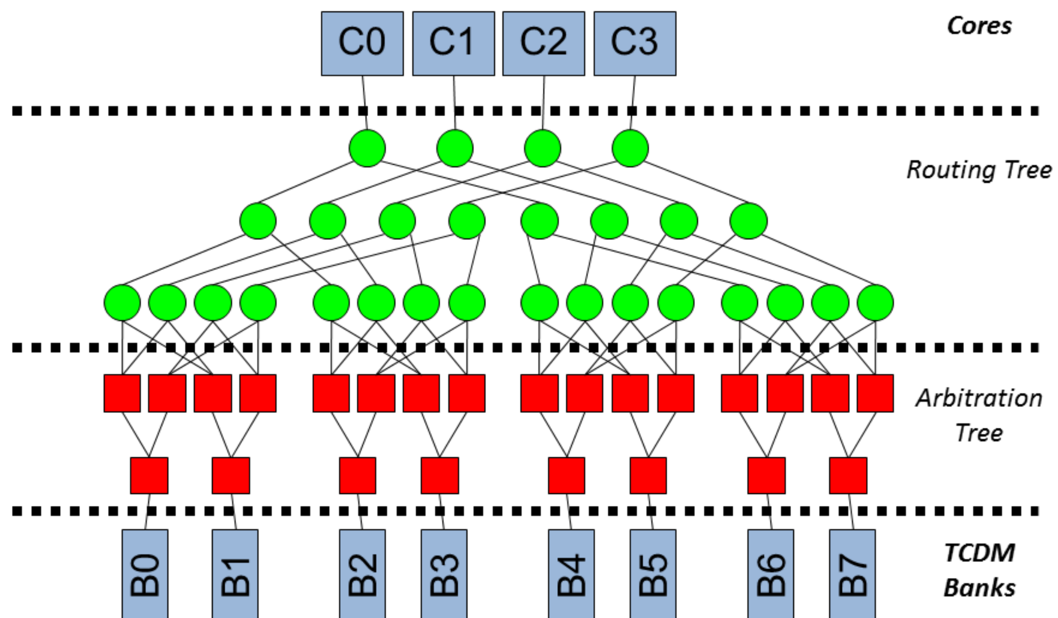

Figure 2.2: Mesh of trees 4x8 (banking factor of 2)

If multiple cores try to read the same address memory, MoT can service whole in 1-clock cycle through a broadcast read. But can happen that multiple cores try to access different addresses that are present in same bank, in this case requests are sequentialized on single bank port. This is also valid when multiple cores try to write in the same memory bank. Therefore, in the worst case, we can have at most N delayed memory transactions, where N is the number of cores within cluster.

2.1.4 Cluster Components

The cluster has a L2/L3 peripheral interconnect that through a demultiplex stage communicate with the outer world. In the embedded processor cluster energy saving is a key aspect to achieve, therefore to improve mem-

ory transfer, the cluster has a DMA engine. This is a simple DMA with one slave port on the peripheral interconnect where cores can directly program a transfer through memory mapped register and two master ports, one on the data interconnect and one on the system interconnect to move data.

2.2 Multi Cluster

The many-core platform is composed of a number of described above clusters interconnected via a NoC. This work take in exam a simple 2x2 mesh as show in Figure x. For each node of NoC is linked a cluster, plus there is a memory controller to the off-chip main memory. According with shared memory architecture, all core can explicitly access every memory segment because they are mapped in a global address space. Clearly, memory transaction outside of own TCDM are subject to NUMA effects: higher latency and smaller bandwidth. The NoC is implemented as asynchronous network, this allows the clock frequency tuning separately for each cluster. In the table below, we summarize latency effects to access different memory levels for a local core:

Local TCDM	1 cycle
Remote TCDM	10 cycles
L3 Memory - DRAM	50 cycles

Table 2.1: Latency time between core and different memories without concurrency

2.3 Host-Accelerator Interface

In real heterogeneous architectures, host processor and accelerator can execute in an asynchronous parallel fashion, and exchange data using non-blocking primitives. Usually the host processor, while running an application can offload asynchronously workloads to the accelerator to increment the efficiently of his work. Host processor can check the status of the workload

only when needed and synchronizing results between accelerator memory and host processors main memory.

In our virtual platform host processors and accelerator are simulate to start their execution in parallel and the first operation done from host processor is to load the job on the accelerator and starts its execution. The simulation ends when the jobs is completed and all statistic are available to the host processor.

Chapter 3

Tasking support for embedded many cores

Research work described in this Chapter is a part of my thesis that I conducted at Microelectronic Embedded Systems Laboratory (MESL). This work is focused on the starting implementation of the OpenMP tasking model suitable for embedded many-core cluster-based processors. We focus our work on the scalability of two runtime (single-, multi-cluster implementation) and in future works we will deeply validate these implementations with a full set of benchmarks.

3.1 Introduction

Since last decade, many-core processors has dominated the field of embedded and general-purpose architectures. This why the Moore's law predictions is even valid but the shrinking of transistor has reach a wall where powerful single core architecture has not much be able to achieves high efficiently in Gops/watt. This evolution brought to the many-core era where hundreds of single processing units (PU) are integrated in a single die. Architecture with high number of cores included in the same package, change drastically the scalability of many-core processors. To avoid these issues, engineers have focused on improving the interconnection systems

for such large amount of PUs. At instance, Plurality's HyperCore Architecture Line (HAL) processors [34], ST Microelectronics STHORM [35], or massively data-parallel architectures such as NVIDIA Fermi GP-GPUs [36]. All these architectures have a large number of PUs grouped in cluster, each cluster has a low-latency high-bandwidth interconnection memory to improve performance of data locality, all clusters are interconnected among them with a medium scalable network on chip (NoC).

While similar architecture can archives high efficient, the task to exploit all computational resources is a responsibility of programming models, compilers and runtime systems. OpenMP has been a very successful runtime library (RTL) in discharging these issues. In particularity one of the primary goal of OpenMP 3.0 was to define a standard dialect to express and to exploit unstructured parallelism efficiently. The tasking execution models represent a suitable candidate to handle irregular parallelism, because enables asynchronous dynamic creation of unit works in a simple matter. Nowadays, there are several implementation of this kind of paradigm, notable examples are Cilk [37], Apple Grand Central Dispatch [38], Intel Carbon [39], and OpenMP specification 3.1 [40].

In this Chapter we describe the design of our OpenMP tasking model implementation based on specification 3.0. We explore two cases suitable for embedded many-core cluster-based processors. In particular, we deeply explain our design and optimizations on the runtime to minimize major bottlenecks of the model. We validate our contribution on a cycle-accuracy virtual platform described in Chapter 2. Moreover, we expand the runtime to execute on a shared memory multi-cluster processors, we test our solution and its scalability in the presence on different granularity tasks.

The rest of the Chapter is structured as follows. We discuss related work in Section 2.2. In Section 2.3 we start to talk about the starting implementation and our contributors, validating our results with benchmarks. Section 2.4 is based on multi-cluster implementation, moreover we talk about our porting from the single cluster. Finally, we conclude the Chapter summarizing our main results.

3.2 Related works

There have been several task programming model to handle irregular parallelism in multi-, many-core architectures. We list a few here.

The Cilk programming language [37] is an extension of C multithreading based on dynamic generation tasks. Cilk scheduler is made on the work-first principle but also adopt an intrusive scheduling for work-stealing technique. However, Cilks is only focused on tasking model and lacks loop constructions that make OpenMP much complete and flexible for solving heterogeneous parallel computational problems.

The Intel work-queuing model [41] is a proprietary extension to OpenMP before the advent of tasking model in the specification 3.0. Intel created a lexical extension of OpenMP for tasking model called taskq construct. These model can have hierarchical generation of tasks and synchronization is achieved of implicit barriers at the end of constructs. However, the implementation shows to exhibit some performance lacks [42], [43] unsuitable for embedded processors.

The Nanos groupd at UPC proposed an extension of OpenMP sections to generate tasks [44]. The mainly problems of these model is the recursive creation for tasks. Direct nesting of section blocks is allowed, but hierarchical synchronization must be accomplished through nesting parallel regions.

Intel Threading Building Blocks (TBB) [45] is a C++ RTL that allows users to program in terms of tasks. Each task is represented as a instance of a particular class called task. Intel TBB runtime has the entire responsibility to schedule tasks to achieve locality and load balancing. Intel TBB has also a loop construct that is built on top of the task scheduler and is responsible for creation tasks. This proposal is similar to OpenMP but is not based on the incremental parallelization and sequential consistency principles that are the base to the success of OpenMP philosophy. Moreover, OpenMP is not targeted to a specific language but works for different OpenMP target languages (C, C++ and Fortran). For this reason, Intel TBB is unportable for embedded processors often programmed through a low-level language

as C.

Our tasking proposal aims to make OpenMP more suitable for embedded many-core cluster-based processors, improving the performance, and focusing on the shared data locality in multiple clusters. We extend OpenMP implementation to be aware of clusters and tightly-coupled processing units.

3.3 OpenMP tasking model

Tasking model has been implemented in OpenMP since the specification 3.0. Nowadays, irregular parallelism are common patterns where complex program can expose in their execution flows. A typical example can be all recursive algorithms such as a tree data structure traversal, multiblock grid solvers, adaptive mesh refinement [47] and dense linear algebra [48], [50], [**?**].

An OpenMP program start his execution with a single thread[1] until not encounters a parallel construct. At this point, execution switch to the runtime that creates a new team of threads composed from itself and N-1 additional threads, where N is the all available cores or a number specified with the num_threads clause. At the end of the team creation, the execution come back to the program and until the end on parallel construction the execution flow will be executed in parallel from multiple threads. When a tread encounters a task construct, a new task region is generated with the code contained within task. Programmer can specify data-sharing clauses (shared, private, firstprivate) associated to the data environment. By default, the execution of the task is asynchronous, when a thread encounter a task scheduling point (TSP) can start to execute tasks. A task can be immediately performed specifying a if clause. Within if clause there is a expression that must be evaluated on the fly, if the expression is false, the thread must suspend the current task region and its execution cannot be resumed until the newly generated task is completed. When on a task construct is specify a untied cluase, the task is not tied to any thread and

[1] In the rest of the document, the term “thread” and the term “core” are completely interchangeable because a thread has a fixed bind with a core

if the task is suspended it can later resumed by any thread in the team. By default, all task are tied. All thread created in a parallel region are guaranteed to be completed at the implicit barrier at the end of the parallel region. As well as at any other explicit barrier constructs. Programmer can force a termination of all created task with a construct called taskwait. When a thread encounter a taskwait construct must wait all its first-level descendant tasks to complete before proceeding. Tasks can be nested, thus a task can contain another tasks. In this case, taskwait construct guarantees the termination only within first level of task and not whole his children.

3.4 Single cluster

In this section take in exam the scalability of our implementation for single cluster. Here, we don't want to make a deeply examination of the runtime (we will it in future works) but we want to test its scalability in tightly-coupled shared memory cluster.

3.4.1 Design and Implementation

We developed our extension of OpenMP tasking model runtime for single cluster from scratch following the specification 3.0. By default, our design relies on a centralized queue stored within team descriptor. Tasks are marked through descriptors which identify their associated task regions and which are stored in the work queue. This queue is built on top of a double linked list and synchronization is achieved with a dedicate lock. There are three basic operations to access on the queue: push, pop, and remove. Respectively, push insert a task in the tail of work-queue, pop try to remove a task from the head and if there are not present task return false. Differently from pop, remove operation remove a task in whatever position is placed. Every time that a task is inserted o removed from work queue a counter is updated to maintain the task number stored in the queue.

Each task is always a child of another task (except the implicit tasks). Implicit tasks are created during the team instantiation, each thread has always an implicit task which lives for all thread duration (until the end of parallel construct). Implicit tasks represent the current execution task of parallel construct. Every time that a thread encounter a task construct, this construct has three effects: i) instance a new task descriptor, ii) push the task within work queue, and iii) push the task within parent queue. For undeferred task (if clause false) the task must be performed on the fly. Task parent contain the current execution task, if the current execution task is at first level, the task parent is the implicit task. Nested tasks form a tree which can have different level (at least one with root the implicit task) related of the number of nested tasks. For example, recursive algorithms can have several nested task levels. Therefore, a task descriptor belongs to two queues in the same moment, the work queue and the parent queue. For the access on the parent queue we implement other three basic operation. Likewise for the work queue, we create push, pop, and remove tasks. They exactly work as work queue operations but they have effects on different memory pointers and counters to maintain the updated status of the parent task. We must maintain also a parent queue because all threads in taskwait can perform only tasks contained in the parent queue of the current execution task [51].

When a thread encounter a task scheduling point start to execute tasks. First of all, thread try to pop a task from the work queue, if this operation fails, thread put himself in sleep mode. Every time that a thread push a task in the work queue, wake up another thread which pop the task and start its execution. When pop operation on the work queue is performed, the thread must even remove the task from parent queue and vice versa. In this way, we guarantees that a task cannot be consume from two threads. Both pop and remove operations must be atomically executed (under lock synchronization).

3.4.2 Experimental Results

To validate our design we performed a synthetic experiment using a SystemC-based virtual platform modeling the tightly-coupled cluster described in Chapter 2. Table **??** summarizes the main architectural parameters.

Table 3.1: Architectural parameters for single cluster.

ARM v6 core	16	TCDM banks	16
I$ size	16KB per core	TCDM latency	2 cycles
I$ line	4 words	TCDM size	256KB
Latency hit	1 cycle	L2 latency	60 cycles
Latency miss	59 cycles	L2 size	256MB

We implement the tasking support on top of a OpenMP runtime optimized for the target platform. We focus our attention on a scalability experiment to handle the speedup with different task granularity. In Fig. 3.1 shows how different task granularity affect speedup of the target platform. In this case, we consider a synthetic benchmark consisting of a loop with a fixed number of iterations (600), each iteration creates a task. The task body contains a loop with a parameterizable number of ALU (ADD) instructions, hence this workload is not affected from critical memory access outside of cluster. Thanks to workload we can distinguish the runtime limitations due to high latency accesses. The creation task loop is performed from the master thread while the remaining 15 threads can immediately start the execution (the producer thread can also join task execution after creating all tasks). We perform experiments for task granularities varying in the range between 1 and 20K. The time of parallel execution is divide by the time of a sequential experimental performed with the same workload to find out the speedup.

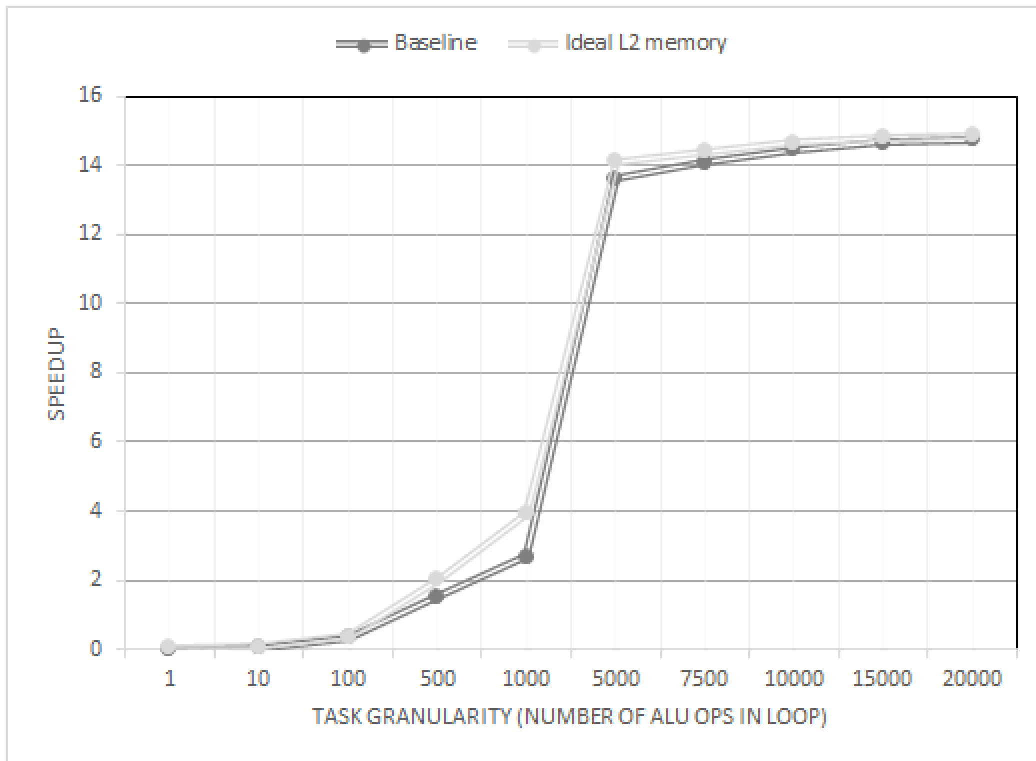

Figure 3.1: Single cluster scalability varying task granularities.

The figure 3.1 shows that high speedup is reached with 5.000 ALU operations for each task, because with this granularity the producer thread is enough fast to maintain all threads busy. Before of this granularity there is always at least one thread that try to pop a task but fails in his attempt because tasks are not large enough to keep busy all threads. When producer thread start to accumulate task in the work queue means that processor can reach an high speedup. Having a optimized implementation of task construct increase the progress for smaller task granularity, moreover having a large task granularity greatly increase the speedup of processor because it make tiny the runtime cost. These two parameters make the difference in an efficiently implementation.

Fig. fig:single_cluster shows two similar progress, the baseline progress and Ideal L2 memory. The different between these two benchmarks are the architectural parameters of the platform. The baseline progress represent the platform parameters in the table **??** without changes. In the Ideal L2

memory we decrease the time access for the L2 memory with a 1 clock cycle. This experiment demonstrates that our runtime execute always within TCDM without accessing outside of cluster avoiding NUMA effects. This is validate by a good progress overlapping.

3.5 Multi Cluster

In this section we port the single cluster implementation for multi-cluster architectures. We first introduce the basic nested parallelism support for OpenMP runtime in tightly-coupled shared memory clusters, then we demonstrate our approach with a synthetic benchmarks focusing on the scalability of our implementation.

Nested Parallelism and Design Implementation

Nowadays many-core architectures have hundreds of simple processing units (PU) integrated on a single chip. Several embedded accelerators grouped into tightly-coupled processor clusters such a large amount of PUs to overcome scalability bottlenecks. These clusters sharing high-performance local interconnection and memory, they usually are interconnected with a scalable medium like a NoC. These system often are based as a shared memory model, where each PU can access to a different levels of memory (L1, L2, L3 memories). However, hierarchies architecture for their nature are subject to non-uniform accesses (NUMA) depending on the physical path.

Nested parallelism represents a powerful programming model for these architectures, it is particularly suitable for accelerations with a large number of processors and a NUMA memory hierarchy. Nested parallelism is typically used to split workload among all processing units avoiding NUMA effects through data locality but can be implemented in different ways [52], [53], [55], [**?**].

We use nested parallelism described in [56] to avoid NUMA effects. A central design choice in this nested parallelism implementation, is the

adoption of a fixed thread pool approach. In straightforward manner, nested parallelism in the runtime is enabled with a API called before the first parallel construct. In this parallel construct, all team threads will be chosen from different clusters. By default in the nested parallel, threads are chosen sequentially (starting from the master thread ID). In this way, in the first parallel we can fit the exactly number of threads related to the number of clusters. This team is composed to threads from different clusters. When these thread encounter the nested parallelism construct, they form a team for each cluster. Every cluster team is only composed from threads that belong to the cluster.

The implementation of tasking model for multi cluster is exactly equal to the single cluster but thanks to nested parallelism each task is confined only in the local team avoiding NUMA effects. This design allows to partition the workload with the usual nested parallelism and supports tasking model in a scalability manner.

3.5.1 Experimental Results

In this subsection we validate our support for tasking model with nested parallelism using a synthetic benchmark to exploit the scalability of our implementation and we compare the results with single cluster runtime. We performed the synthetic experiment with a extensive set of parameters and we use a SystemC-based virtual platform described in Chapter 2 to model an embedded many-core multi-cluster accelerator. We summarize the main architectural parameters in the table below.

Table 3.2: Architectural parameters for multi cluster.

CLUSTER	MULTI-CLUSTER		
ARM v6 core	16	Clusters	4
I$ size	16KB per core	NoC topology	2x2 mesh
TCDM size	256KB	L3 size	256 MB
TCDM access time	1 cycles	L3 access time	50 cycles

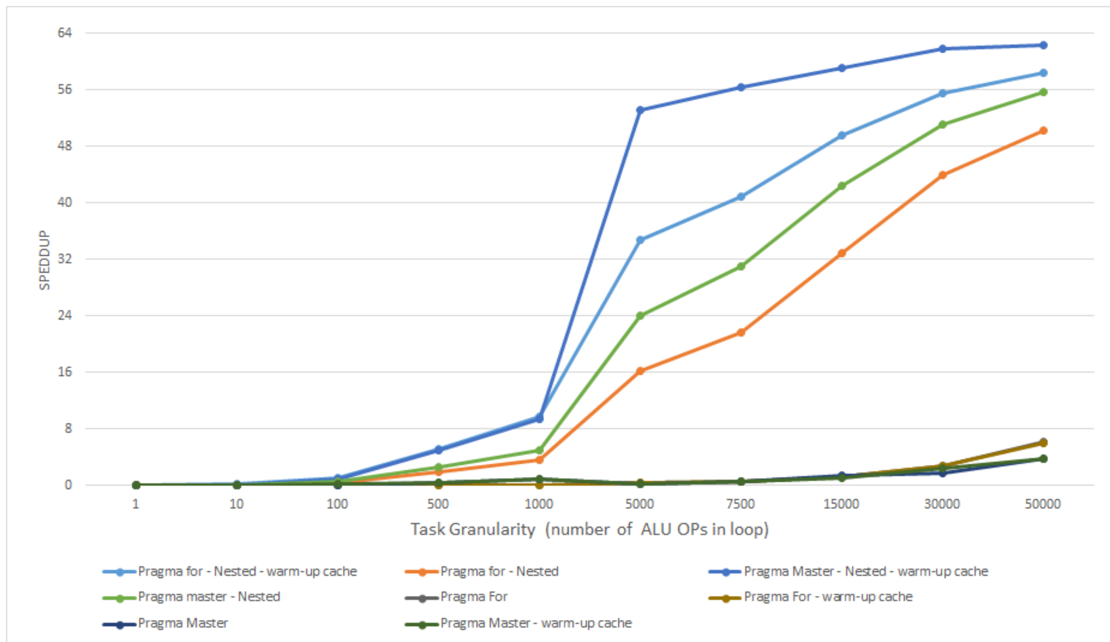

Figure 3.2: Multi cluster scalability varying task granularities.

Fig. 3.2 shows our experiments for multi-cluster architecture. Synthetic benchmarks that we use to validate our architecture are structured as single-cluster benchmarks, they are similarly compose in task parametrization and for the type of workload. The workload is not affected from high latency memory accesses because the workload is composed only from ALU operations leaving memory transactions only for the runtime.

The two main trends show an important factor that extremely penal the runtime, the absence of nested parallelism. We can see that without nested parallelism benchmark speedup collapse, arriving in the best case around 6x for huge coarse-grain tasks. This result is very important of our point of view because the runtime penalty without nested parallelism has a huge scalability bottleneck independent on the running applications.

Figure shows that until the benchmarks not reach the 1.000 ALU operations for task their speedup trends are similar, after this threshold progress change with wide differences. Another valid result to underline is the different between pragma master and for. The latter has a major penalty due synchronization locks that have a large impact on the performance such to

completely discard the benift brought to parallel task pushing, this penalty can reach 30% of difference in the worst case.

The last import different is between the warm-up cache and non-warm-up cache. We can see in Figure that a non-warm-up cache can reach up to 45% speedup different between the two benchmarks. Non-warm-up cache can be fixed with a pre-fetch instruction strategy mitigating the difference.

Chapter 4

Variability-aware OpenMP

Research work describe in this Chapter, was conducted as part of joint group between the Microelectronic Embedded Systems Laboratory (MESL) of University of California, San Diego and Microelectronics Research Laboratory (MICREL) of University of Bologna. I developed this part of the thesis during my visiting trip at MESL and it has been presented to the research community as a special issue on "Emerging and Selected Topics in Circuits and Systems", IEEE Journal [21].

4.1 Introduction

In the nanotechnology era circuit level components are increasingly affected on variability that induce high timing errors. To correct these effects, are used recovery mechanisms in the circuit level with significant energy penalty. The energy cost of timing errors can be reduce by bringing this knowledge on the software task where workload schedulers can take in exam processor heterogeneity within schedulers decreasing errors and overhead. In this chapter we present a variability-aware OpenMP (VOMP) runtime suitable for embedded shared memory processor clusters. The runtime system monitors software workload units that are executed on various cores and their impacts on timing errors, transparently associates different descriptive metadata for each type of workload to every cores. By

utilizing this characterized information, we realize smart schedulers that prevent high timing errors taking countermeasures against bad affiliations between tasks and cores. VOMP is a extension of OpenMP v3.0 and it is strongly based on tasking model to handle variability effects, but we also covers others parallel constructions including sections and for loop.

We define work-unit vulnerability (WUV) that we use to capture metadata on timing errors caused by circuit-level variability bringing these informations on high-level software stack. WUV is a metadata to classify variability for a given core, each workload has a different WUV that is updated every time which is performed on the selected core. The hardware provide support to allows access WUV metadata within core.

4.2 Related Works

Nowadays have been proposed various solutions to mitigate hardware variability.

Circuit level techniques [1], [2], [3] monitor the pach delay variation and when find out timing errors raise up a warning signal. The integer resilient core [2] use a EDS [1] circuit in the pipeline stages when there are critical paths. When EDS detect a timing error, raises the warning signal to prevent conclusion of the errant instruction. Error control unit (ECU) receive the warning signal propagated from EDS, for this operation is requested an extra recovery penalty. ECU flushes the entire pipeline stages, and then replays the errant instruction multiple times (N+1). The first N instructions are just replica instructions, while the $N+1^{th}$ instruction is a valid instruction that changes the state of the machine. Moreover, this mechanism has a lower impact on the occupied area in the die and a scalable timing error recovery without changing the clock frequency. If the pipeline stages of the processor are a high number this mechanism impose an huge timing penalty and power consumption [4], [4].

Various software level approaches are presented to expose variability avoiding expensive hardware mechanism to guarantees correct program

execution. From fine-grained abstraction to capture characterize vulnerability of instructions set [12], to the coarser-grained abstraction focus on threads [6], procedure [7], and tasks [8], [9], [10], [26], [11]. However, these mechanism have the following lacks: i) limited applicability in different processors and environment because are tied for a determinate technology [6], [7], [10]. ii) no support for runtime characterization and dynamic mapping [7], [10]. iii) high penalty in recovering and scheduling [8], [9], [26], [11], [6].

4.3 Architectural support for variation-affected processors

We demonstrate our approach on a SystemC-based virtual platform described in Chapter 2, but to emulate variation effects we integrate variation models at level instruction using a characterization methodology presented in [19]. We re-synthesized the processors described in Chapter 2 - Section I, with the 45nm TSMC technology library, general-purpose process. The front-end flow with normal V_{TH} cells has been performed using Synopsys DesignCompiler and for the back-end we have been used Synopsys IC Compiler, the core is optimized for performance. We analyze effects of dynamic and temperature variation analyzing the power and delay variabilities on the individual instructions. Moreover, we use the features of Synopsys PrimeTime to scale voltage and temperature. We use the same flow to extract the power models for other cluster components. We apply these models on the virtual platform where it dynamically maps on instructions for the corresponding instruction variability models.

To detect and correct variability effects induced from the models, we equipped all cores with two circuit-level resiliency techniques. First, each cores is based on EDS [16] circuit sensors that allows to detect timing errors as consequent of dynamic delay variation. This mechanism can recover errant instructions without changing the clock frequency because the core employs multiple-issue instruction replay mechanism [17] in its ECU, or error recovery unit.

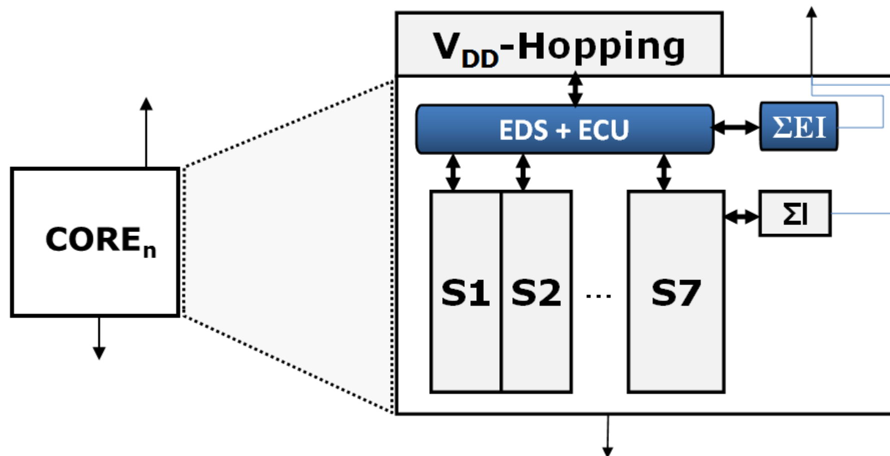

Figure 4.1: EDS e VDD-Hopping.

All cores within cluster is essentials that they have all the same clock frequency to avoid latency of synchronization [15], so the second technique is v_{DD}-hopping [18], this mechanism change the voltage of slow cores that are affected by static process variation. This enables all cores to work on the same frequency avoiding inter-core synchronization that increases TCDM latecy.

4.4 Work-unit vulnerability and VOMP work-sharing

OpenMP [40] specification is based on a particularity construct called *#pragma omp parallel*. A parallel directive has the effect of lunching multiple instance on the enclosed code on all processors (or a specified number of them). Within the parallel directive, we can find a shared data structure called work share (WS) that is the base of the OpenMP specification. WS is used to deploy jobs to all thread that compose a parallel, all thread involved in the parallel pragma compose a *Team*. Each WS contains within one or more jobs called work-unit (WU), each WU is deployed on a thread and executed.

There are different OpenMP costructs that use WS to deploy their jobs: *#pragma omp for, #pragma omp sections, #pragma omp task, #pragma omp master, #pragma omp single*. We focus our work on tree directives, respectively sections, task and for because others constructs are based on a WS with only one WU. Single directive is performed from the first thread that encounters directive, instead master directive can be only performed to the master thread (team creator).

```
#pragma omp parallel
{
   #pragma omp for
   for (i=0; i<N; i++)          WU type 1
      loop_A();

   #pragma omp sections
   {
      #pragma omp section
      section_A();              WU type 2
      #pragma omp section
      section_B();              WU type 3
   }

   for (i=0; i<N; i++)
      #pragma omp task
      loop_B();                 WU type 4
}
```

Figure 4.2: Outlined WU types in a OpenMP program: task, section, for.

Therefore, this work take in exam for, sections and task directive because they have different WUs. In particular, sections directives contains multiple section blocks that are executed only one time from the first available thread. For instance, it easy to describe software pipeline parallelism with sections, by just adding point of synchronizations among parallel directives to maintain dependency between states. Task directive is the only construct not based on the WS, each task can be considerate a independently WU with a asynchrony execution and deployed from a dedicate

infrastructural built on the top of a queue data structure. Task has been made to exploit irregular parallelism in the presence of complicated control structures [**?**]. However, task implies significant overheads and a efficient implementation is a key aspect. Intuitively, for directive execute always the same code associated to the loop nest and distributes loop iterations over available cores.

Enclosing portions of code using above directives allows us to have different WU types in the program, as a direct consequence our runtime can characterized them. Therefore, we create a new metric: parallel work-unit vulnerability (WUV). WUV is used to estimate execution time of each WU type for each core under variability. This metric is brought on the software stack and used from VOMP runtime in the scheduler. WU types can be done statically (i.e., at compile time), moreover WUV characterization has to be done online due two main reasons. First, dynamic instances of the same WU can have different effects in different cores, because WU may exercise the processor pipeline in a non-identical manner due to the class of instructions that compose the WU. Second, WU can have different behavior due to data-dependent control flow. WUV is defined as follows:

$$WUV_{(i,j)} = \sum I + \sum RI \mid \forall core_i, \forall WUtype_j \tag{4.1}$$

where ΣI is the number of error-free executed instructions; ΣRI is the number of replayed instructions[1] during execution of WU type *j* on core *i*, as reported by the ECU. Intuitively, for a given WU type if all the instructions run without any timing error, the corresponding WUV is equal to ΣI. In the event of timing errors, WUV also accounts for the additional replica instructions. If WUV is low means there are few (or nothing) recovery cycles, consequently instruction count is low and we have a higher throughput and energy efficiency. WUV dynamically characterizes both vulnerability and execution time of WU types.

[1] proportional to the number of errant instructions

4.4.1 WUV corner cases

WUV is a representative metadata of the total number of instructions executed to a core, this total number is the sum of the assembler instruction and all replay instruction caused by timing errors. WUV is not uniform across different cores, which can cause different timing errors related on the type of variability. To demonstrate how WUV change in present of different level of variability and how these metadata reaches the software level, we build four stress test, which iterate several times over an identical instruction, as show in Fig. 4.3.

These synthetic benchmarks exploit variation among the cores for the same WU thus we can exploit behaviors at the software level. Fig. 4.4 illustrates the synthetic benchmarks with #pragma omp task construct, while in Fig. 4.7 we have the synthetic benchmark with #pragma omp sections construct. We organize the presentation of these experiment in following three consecutive subsections, one per each OpenMP construct.

```
#define OP_MUL   1
#define OP_ADD   2
#define OP_DIV   3
#define OP_SHIFT 4
int A[][][], B[][][], C[][][];
void WU_run (int z, int OP)
{
  for (int y = 0; y < N; y++)
    for (int x = 0; x < N; x++)
    {
      switch(OP)
      {
        case OP_MUL:   C[x][y][z] =
            A[x][y][z] * B[x][y][z];
            break;

        case OP_ADD:   C[x][y][z] =
            A[x][y][z] + B[x][y][z];
            break;

        case OP_DIV:   C[x][y][z] =
            A[x][y][z] / B[x][y][z];
            break;

        case OP_SHIFT: C[x][y][z] =
            A[x][y][z] ≫ B[x][y][z];
            break;
      }
    }
}
```

Figure 4.3: WU types each stressing a different class of instructions.

1) task-Level WUV

We measure WUV for different WU type (here, task) when executing on fixed and variable operating corners (current voltage and temperature). We change temperature in a range of 0°C - 140°C and a voltage range of 0.88 - 1.1V. In this section we illustrate a normalized WUV (NWUV) dividing WUV value to its ΣI. Therefor this normalized value will have a value equal or grater to 1. For example, if NWUV has a value of 1 means that there is no replica instructions (ΣRI = 0).

```
#pragma omp parallel
{
  #pragma omp master
  {
    for (int z = 0; z < N; z++)
      #pragma omp task
        WU_run (z, OP_MUL);

    for (int z = 0; z < N; z++)
      #pragma omp task
        WU_run (z, OP_ADD);

    for (int z = 0; z < N; z++)
      #pragma omp task
        WU_run (z, OP_DIV);

    for (int z = 0; z < N; z++)
      #pragma omp task
        WU_run (z, OP_SHIFT);
  }
}
```

Figure 4.4: Synthetic benchmark using OpenMP task.

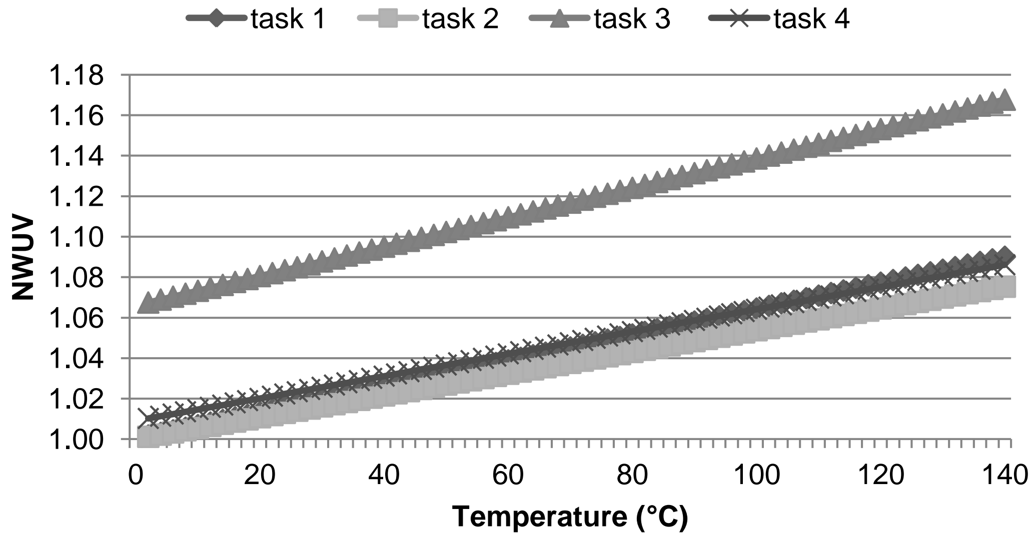

Figure 4.5: Normalized WUV (NWUV) to temperature variations for task types.

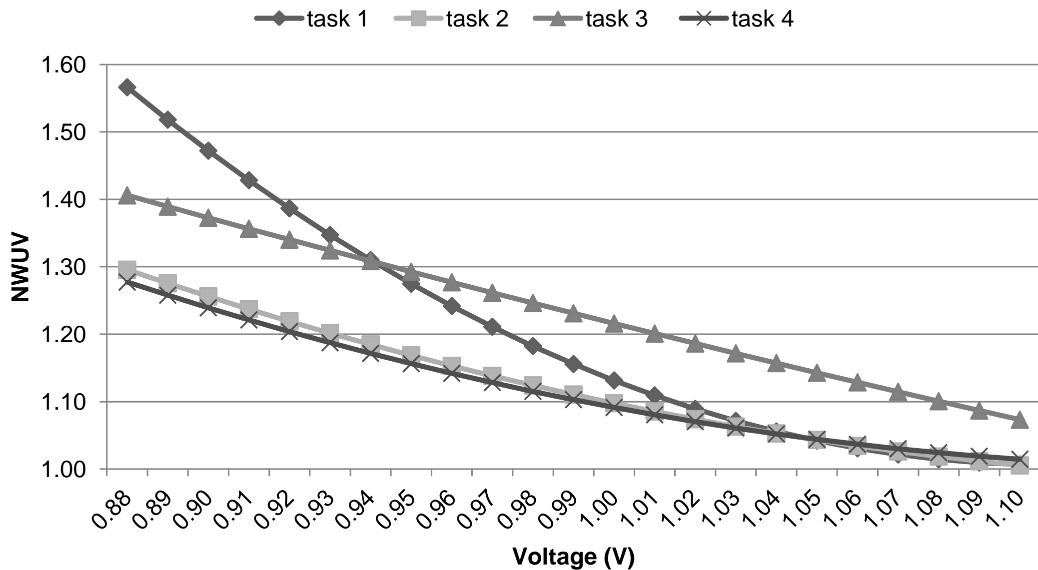

Figure 4.6: Normalized WUV to voltage variations for task types.

Fig. 4.5 shows the task-level WUV for a core with a fixed voltage of 1.1V while chip temperature is dynamically varied. We can see that task-level vulnerability change related in function of temperature, for example for the $task_1$ at temperature of 0°C result in a NWUV value of 1.0017, increasing the temperature to 140°C we have a NWUV of 1.09 with a different of 9%

between the states. With these corner cases we can prove that variability is a direct manifestation of temperature variation. In the fig. 4.5 at the fixed temperature of 0°C, we can see that there is a considerable variation across task types. WUV for each task type is different even with in absence of environmental variation, different class of instructions have different behaviors within the same operating conditions.

Fig. 4.6 shows the task-level WUV for a core with a fixed temperature of 10°C while chip voltage frequency is dynamically varied. As show in the graph, higher voltage reduce timing errors because NWUV is tends to reach ideal value (1). Likewise in the Fig. 4.6 different class of instructions have different NWUV penalty changing operating voltage. This behaviors show us that vulnerability of instructions is not uniform [23] in different level of vulnerability for task types.

2) sections-Level WUV

Fig. 4.7 shows the synthetic benchmark used for sections profiling. This benchmarks represent a software pipeline widely used in processor virtual platforms or image processing kernels where a set of filter is applied in sequence on the image blocks. Each WU identify as a section is performed on a different core. In the beginning and at the end of sections there are synchronization points to maintain coherence among stage pipelines that we implement on top of test-and-set semaphores. In this way, we can guarantees that once computation of previous pipeline is finished can start the next one. Inside of each block section there is a loop that simulate a computational cost for a class of instruction. Note however that dependency is only between a section and its subsequent, this allows paralleling execution of sections.

```
#pragma omp parallel
{
  for (int z = 0; z < N; z++)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
      {
        WU_run(z, OP_MUL);
        synch();
      }
      #pragma omp section {
        synch();
        WU_run(z, OP_ADD);
        synch();
      }
      #pragma omp section {
        synch();
        WU_run(z, OP_DIV);
        synch();
      }
      #pragma omp section {
        synch();
        WU_run(z, OP_SHIFT);
      }
    }
  }
}
```

Figure 4.7: Software pipelined synthetic benchmark using OpenMP sections.

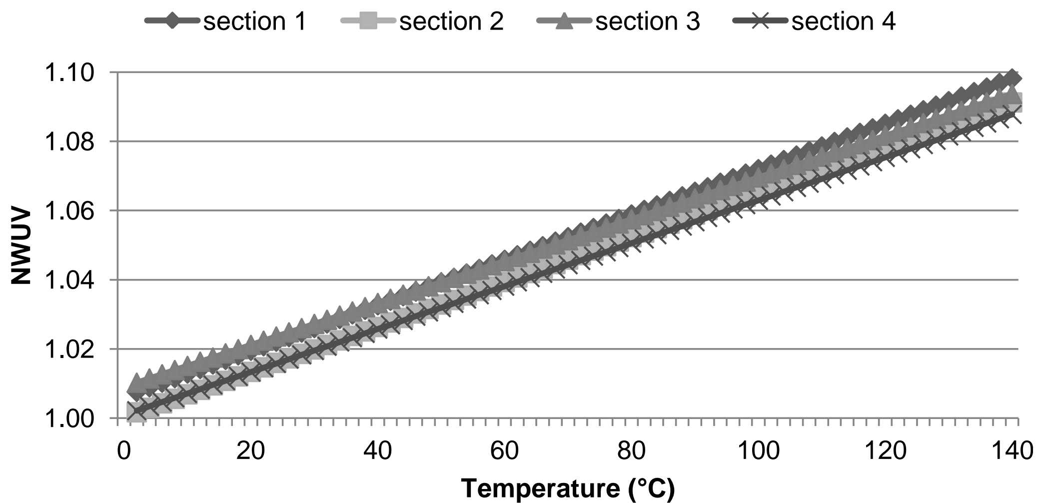

Figure 4.8: Normalized WUV to temperature variations for sections types.

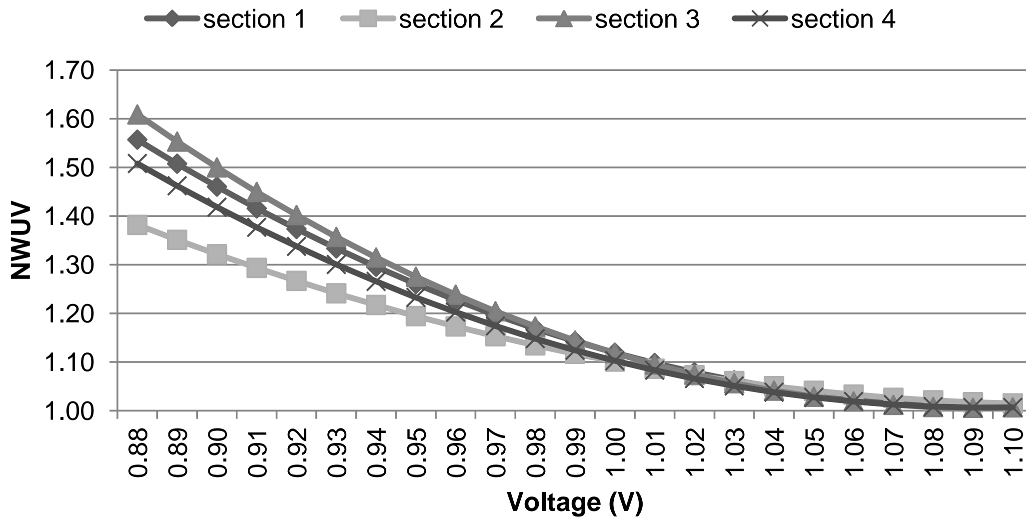

Figure 4.9: Normalized WUV to voltage variations for sections types.

In this benchmark, there are N_{sec} stages, such that $N_{sec} < N_{core}$, where N_{core} is the number of available cores[2]. Normally, at the end on each sections there is a wait construct for synchronization, to avoid this constraint, we specify nowait clause and allow idle cores to start execution of the next section.

Fig. 4.8 shows NWUV values for a core with a fix supply voltage of 1,1 V and variable temperature range of 0°C–140°C, while Fig. 4.9 shows NWUV values for a fixed temperature of 10°C with a supply voltage variation range of 0.22V. Likewise for task-Level WUV, this plot describes behaviors increasing function of temperature in a range of 0°C - 140°C and decreasing function of voltage in a range of 0.88 V - 1.10 V respectively. We can observe that at temperature of 140°C we have an WUV average penalty around 9%, in the plot of voltage variation penalty increase up to 50% when we have 0.22V. Among the section types we have a maximum different of 16% observed at (10°C,1.09V).

[2] 16 cores in our platform

3) for-Level WUV

Parallel applications accelerated trough many-core platforms are often focused on splitting work on each available core. Afterwards, synchronization is typically done with barriers for waiting the termination of all WUs. Usual mechanisms are used on parallel loops, whose all iterations are spread and assigned on different cores to be processed. Equally, all loop iterations can be spread on different clusters and afterwards scheduled to the available cores to exploit distribute workload in tingly-couple cluster-base processors. OpenMP provides a well-defined directive to distribute loop iterations among the cores, #pragma omp for. This directive create a number of WUs related on the number of iterations that compose the for loop. Each WU is assigned on a core and performed. In our case, every WU is dynamically instantiated but it uniquely identifies as the same type from our characterization point of view. The program code that compose all loop iterations is always the same for all WUs created. In other words, for directive build a homogeneous work-load across all cores. In this case, our VOMP runtime has a limit capability of scheduling due of this behavior, therefore for-Level WUV will not be handle for variation effects.

Conclusion for WUV

Exploiting all corner cases in the presented experiments we can conclude that WUV varies significantly: 1) among WU types; and 2) among the operating conditions. We have conduced all these corner cases to exploit how much variability affects different class of instructions and how they varies in the presence of temperature and voltage range. This confirms the observation that executing different stream of instructions my result in different error rates [24]. We observe that in any operating condition the WUV os simple arithmetic instructions (e. g., addition/shift) is always lower than WUV of complex arithmetic operations (e. g., MUL/DIV). A deeply study of variability for class of instructions in different operating conditions in voltage and temperature are provided in [25]. Moreover, even the behavior of identical instructions on different cores is not equal because voltage and

temperature can change dynamically at run time. This is much evident for the #pragma omp for construct, which always distribute the same WU among the cores. Yet, WUV can have significantly variation caused by the different variability that each core can be affected. All these motivations are collected to describe how much is important having WUV characterization for different WU types and for different cores.

4.4.2 Run-time WUV Characterization

At the end on a WU execution, VOMP can have access to the metadata related on the WU performed. To quantify WUV, VOMP collects ΣI and ΣRI through a set of available counters in the ECU. Each core is responsible for executing and monitoring his characterization related at performed task.

```
When task_j is scheduled on core_i:
begin
  EXTRACT_TASK (task_j)
  WUV_old = LUT_rd (task_j, core_i)
  reset_WUV (core_i)
  EXECUTE_TASK (task_j)
  WUV_new = read_WUV (core_i)
  WUV_write = (WUV_new-(WUV_new≫3))+(WUV_old≫3)
  LUT_wr (task_j, core_i, WUV_write)
end
```

Figure 4.10: Pseudo-code for task-level WUV characterization.

After the execution, core read WUV represented as a a two-dimensional lookup table (LUT) where each row contain a core and each column contain a WU type. This lookup table is distribute on the TCDM for having a fast and parallel access from every core. Each entry of the LUT contain a 32-bit integer data with with the sum of ΣI and ΣRI (WUV). Hence, LUT has a memory footprint of $N_{WU} \times 4 \times N_{core}$ Bytes, where N_{WU} being the number of total task types that can be indexed in VOMP and N_{core} is the number of the cores in the cluster. We provide two simple APIs to access the LUT, respectively for reading and writing:

```
int LUT_rd (int WUtype, int coreID);
void  LUT_wr (int WUtype, int coreID, int WUV);
```

They have two parameter to indicate WU type (row), core ID (column). The write function even have a thirst parameter that indicates the new value of WUV. In addition, we implement two functions to access and reset metadata within core.

```
int read_WUV (int coreID);
void reset_WUV (int coreID);
```

In Fig. 4.10 we demonstrate how scheduling has been modified to support runtime WUV characterization (our additions in **bold font**). These instance shows the case for task scheduler, we modify the sections scheduler in an equivalent manner.

In the startup phase of the program, the LUT has all value initialized to zero and in principle we need to characterize a couple $<$*WUtype*, *coreID*$>$ only once. Afterwards, we have a value in the LUT that shows metadata for a WU related on a core thus we can use in the scheduler. However, we rather keep the characterization active for all the time of program, for each task performed WUV is updated and re-written in the LUT. Updating procedure not replace entire metadata value but the scheduler calculate the average between old and new value with a factor K applied to one of the value to balance the imprinting of the parameter (usual K is 0.5 that means both parameter have the same impact on the average). This is used to better captures the effects of dynamic variations on the cores and having a better scheduling. On the other hand, we have a fixed negligible overhead given to the computational cost of the characterization APIs.

4.5 VOMP schedulers

4.5.1 Variation-Aware Task Scheduling (VATS)

In this subsection we present our specific of variation-aware scheduling policy and how we implement it following the standard OpenMP specification. OpenMP tasking has been implemented from the specification 3.0 and

has been considered a convenient programming model to handle irregular parallelism in multi- and many core systems [26], [27], [28], [29]. In the OpenMP specification there is not defined a implemented strategy to realize the task scheduler, but usually it is build on a centralized queue contenting all task descriptors. This queue is realized as a FIFO data structure where all cores can access to insert and remove tasks if they are producers or consumers respectively. A centralize data structure has a very simple and efficient implementation, which can reduce relevant computational overhead and decries energy - and resource constrain system. This design choice work very well for homogeneous shared memory processors with uniform memory access (UMA) but places limitations in variability-affect processors because have important consequences on the task management accomplished by schedulers.

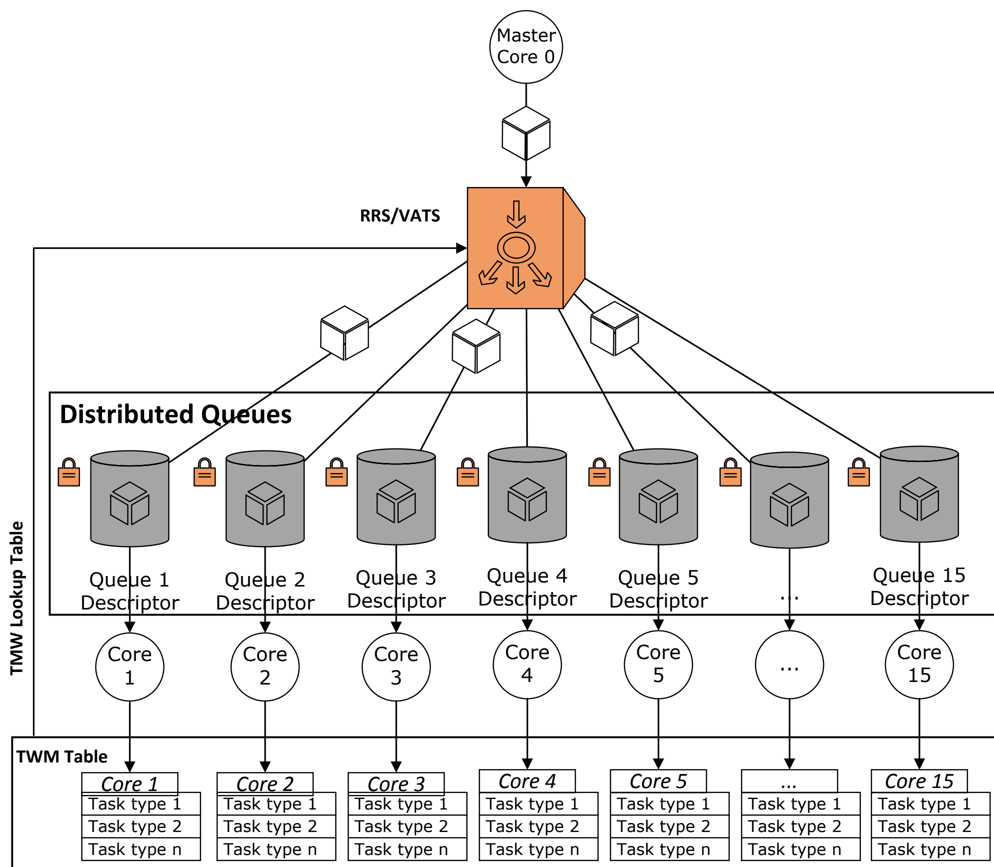

Figure 4.11: Distributed queues for OpenMP tasking.

Our OpenMP implementation is based on a set of distribute queues that are private per each core and where all thread can push and pop task descriptors. Fig. 4.11 shows a schema of our OpenMP tasking implementation based on multiple private queues. Every core has a dedicate queue where are stored all assigned task for the core. The queues have two basic operations: insert and extract, which are synchronized using multiple locks (one lock per each queue). All task descriptors and the queue data structures are allocate in TCDM for minimal access time. The queue data structure are instantiated during the startup phase of the runtime to avoid the time overhead for dynamic memory management. All cores presenting in our platform can transit in low-power IDLE mode. When a producer core inserts a new task in a private queue if the queue core is in IDLE mode, producer have the responsibility to wake up the consumer. Producer core can exploit the state of a core inspecting an additional flag placed in the queue descriptor[3]. Each queue has a own lock to synchronize all access on data structure, every time that a core try to insert o remove a task descriptor from a queue the first operation is to take the ownership of the lock associated on the queue. As long as a thread holds a lock no one can access on the queue, every lock are implemented through a busy waiting on the test-and-set memory address associate to the queue descriptor. Extraction operation is only possible from the head of the queue, while insertion occurs from the head and the tail. Insertion from head are widely used to prioritize the execution of non-characterized tasks, in this way the characterization try to be as fast as possible reducing wrong scheduling in the early stages of execution. Stealing tasks as equal to a normal pop operation occurs from the head of the queue.

As a baseline policy we implement a simple round-robin scheduler (RRS). This policy deploy all tasks in equal manner to all queues, it start to distribute tasks from the first queue to the last queue starting again when it finish the round until the tasks are not all assigned. Runtime has a minimal overhead when use this scheduler due to a very lightweight implementation. To avoid unbalanced issues, RRS is enhanced with a steal scheduling

[3] it can have only two value: executing and sleeping

policy. Steal policy work in a round-robin fashion among all queues.

Here, we propose a policy for variability-aware task scheduling (VATS) shown in Algorithm 4.5.1. This algorithm allows to use WUV metadata to assign tasks to cores for minimizing both number of instruction replays and unbalanced loads. VATS try to avoid that tasks are executed on unreliable cores but this is impossible in the startup-phase of algorithm even when there are no WUV metadata available. In this particular case the scheduler operates in round-robin mode. Every time that scheduler find out a new task type not initialized in the LUT, push directly the task in the head of the core queue (out-of-order task characterization). This will give higher priority to the non-characterized task types speeding up the "system warm-up".

Algorithm 4.5.1: VATS $(task_j)$

---

for $i \leftarrow 1$ **to** N_{core}

do $\begin{cases} load_i \leftarrow loadQueue_i + WUV(core_i, task_j) \\ min \leftarrow findMinimum(load_i) \end{cases}$

$Queue_{min} \leftarrow insert(task_j)$

return (min)

VATS scheduler take into account also the load on each queue, to avoid unbalanced situation. The load of the queue is formed from the sum of all WUV task metadata contained. The queue with the lower load plus the metadata value of the current task is the chosen queue. This metric is better than counting only the number of tasks present within queue because different task may have various computational weight.

To avoid imbalance effects due to non-homogeneous task duration and other system-level issues, VATS also has a most loaded queue-first stealing algorithm. This policy checks the load of all queues and choose the queue with higher load. In the meantime of execution of stealing policy the scheduler checks if some tasks have been inserted in the local queue. In this case the core stop the execution of stealing policy and start to execute the own tasks, otherwise it continues executing the stealing algorithm until

all task in the system are executed.

4.5.2 Variation-Aware Section Scheduling (VASS)

The default of OpenMP runtime is to allocate a section to the first available thread, in a first-come fist-server policy (FCFS). Usually sections have not dependency among them and there is not a distribute algorithm. However, when sections are used to model a parallel pipeline software we have constraint among the sections. Variability effects have a great impact on the performance, because timing errors for a section causing bottlenecks in the entire pipeline execution. This effect dominates the overall pipeline duration. Variability effects have a double impact on these type of software caused by standard variability issues and low performance due to the variation-affected synchronization constraints.

For these cases, we propose a variation-aware section scheduling (VASS) policy shown in Algorithm 4.5.2. Similarly to VATS, VASS has a warm-up phase where assign different section to different cores until all cores performs all section types at least once. After the execution of each section, the characterization process update the WUV metadata in the LUT using the same mechanism described for tasks in the previous Section. After the warm-up phase, the scheduler can start to assign each section to a set of suitable cores avoiding the major problems given by variability.

Algorithm 4.5.2: VASS ($sec_0 : sec_{N_{sec}}$)

$sortedSecList \leftarrow SortSectionsWUV(sec_0 : sec_{N_{sec}})$
while $sortedSecList \neq EMPTY$
do $\begin{cases} secID \leftarrow extractTopList(sortedSecList) \\ \{coreIDs\} \leftarrow findBestSetCores(secID) \\ tag[\{coreIDs\}] \leftarrow tag[\{coreIDs\}] \bigcup secID \end{cases}$
return $(tag[core_0 : core_{N_{core}}])$

The first operation of VASS is to sort all sections based on their average WUV. This means that first section after the sorting has the higher average

WUV metadata and the last has the lower. Hence, a section can be executed only on a set of suitable cores that display fewer error rate during its execution. For this reason, each core has a vector tag to maintain a list of sections that can execute. This constraint avoid that worse cores execute long or high vulnerable types of sections. In other word, a good core can execute all types of sections and an worse core can execute only few type of sections. Worse core can execute only short sections and with minor degree of variation effects. VASS performs a one-to-many mapping between the section types (i.e., stages) and the core such that the overall execution time is reduced.

After the execution of the scheduler, each core has a fulled vector that use to choose whose section can execute. Every time that a core encounter a section can independently discern if the current section can be executed from him.

4.6 Experimental results

We demonstrate our approach on an virtual platform describes in the Chapter 2 with updates describes in the section "Architectural support for variation-effected processors" on this Chapter. We use a configuration listed in the table below.

Table 4.1: Architectural parameters of the cluster.

ARM v6 core	16	TCDM banks	16
I$ size	16KB per core	TCDM latency	2 cycles
I$ line	4 words	TCDM size	256KB
Latency hit	1 cycle	L2 latency	60 cycles
Latency miss	59 cycles	L2 size	256MB

4.6.1 Results for Tasking

We demonstrate our VOMP task implementation with a set of largely adopted kernels used in the field of image processing, cryptography and mathematical matrix operations. These kernels include RGB-to-HSV and

XYZ-to-RGB for colormap conversions, Integral image and Sobel for filter operations, FAST for corner detection, Color Tracking , Strassen matrix multiplication, and Blowfish for encryption/decryption. Each kernel has on task type and there is not task dependency during the execution. We collect information regard two fundamental parameters for embedded systems, total execution time and energy consumption. All values that we will show are normalized values. We compare VATS policy normalized on the baseline RRS policy to demonstrate the effects of our solution.

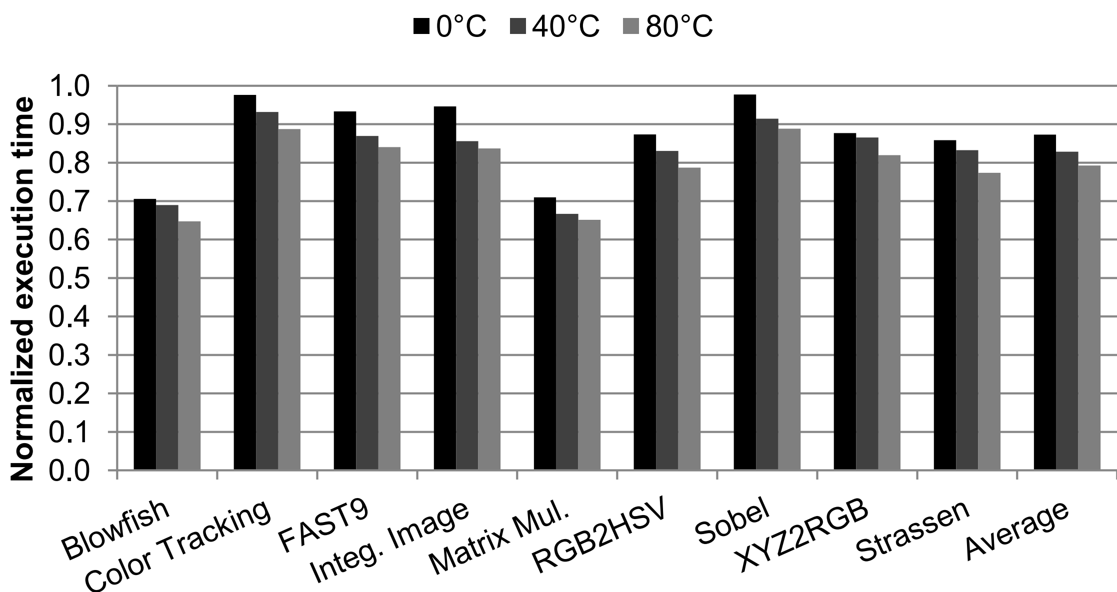

Figure 4.12: Execution time for VATS normalized to RRS under temperature variation.

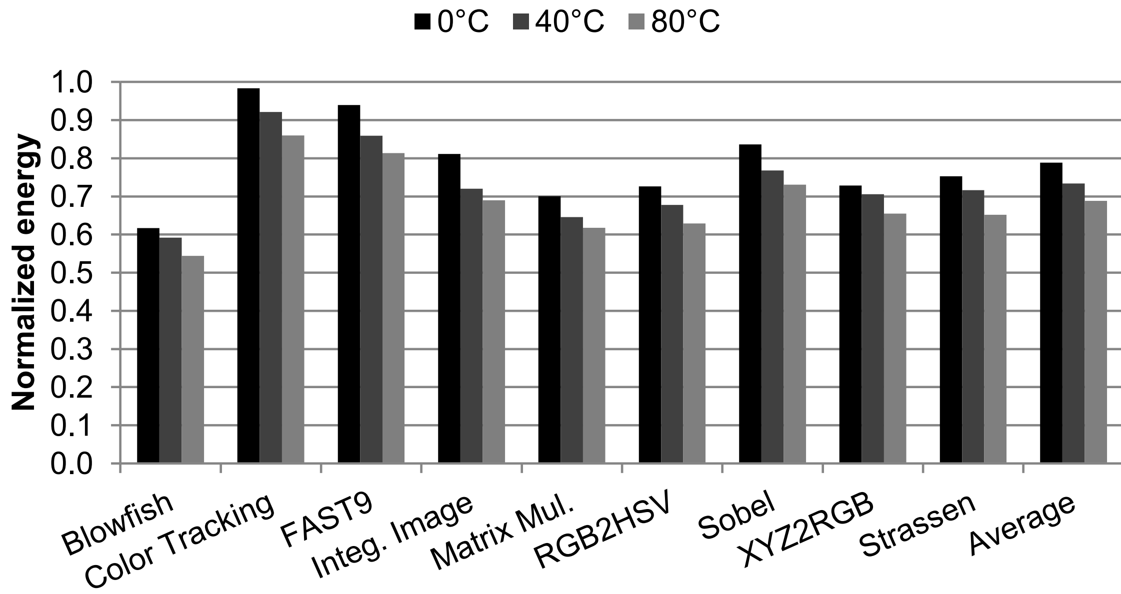

Figure 4.13: Energy consumption for VATS normalized to RRS under temperature variation.

Fig. 4.12 shows the execution time for all the kernels for three operating corners with temperature of 0°C, 40°C, and 80°C. As shows, at an operating temperature of 0°C, VATS achieves up to 30% better performance than RRS, and 13% on average. Entire our scheduler and characterization APIs are paid off from the gain of avoiding timing errors. Moreover, VATS displays a robust behavior across temperature variations thanks to the reflection by the always-on characterizations. At higher temperature, VATS reaches better results because in RRS policy increase replayed instructions, conversely VATS can save more instructions from timing errors. VATS achieves average performance gain of 17% (at 40°C) and 21% (80°C).

Fig. 4.13 shows the energy consumption of the kernel. Likewise on execution time, VATS achieves on average 21% and up to 38% better energy efficiency than RRS at temperature of 0°C. Further, VATS increases his gap from RRS in higher temperature penalty saving 31% at the 80°C.

We also compares the TLV technique with the centralized queue proposed in [30]. TLC has a variation-agnostic task insertion operations with a penalty of 75% respect of RRS. Moreover, TLV has 100% penalty in energy consumption from RRS. This huge gap between RRS and TLV is because TLV characterization does not consider the overall system workload. TLV is based on a single tasking queue which has limits of potentials for task scheduling policies: a core can pop a task from the head of the queue and choose if to proceed to the execution or leave it in the queue for other cores. Every time that a core execute this algorithm all available cores must wait the completion of operation. Can happen that a core try to pop always the same task if also other core decide to leave it, in this case the current core can discard the task at most three time. All these constraints describe the lack of efficient utilization of resources under variability.

4.6.2 Results for Sections

We demonstrate our VOMP section implementation with a set of pipeline software typically used from a different field of research. We have Pitch extractor algorithm (PEA), and FFT with covariance matrix factorization

(DFT-COV) are embedded signal processing kernels extracted from [31], [32]. Sobel and Prewitt are filter operations useful in the edge detection algorithms. N-body is a simulation of a large number of particles under the influence of physical forces. Mersenne twister is a pseudorandom number generator. Synthetic is a microkernel implementing a fourstage parallel pipeline (see Fig. 4.7), representative of streaming applications [33]. We evaluate the effectiveness and robustness of our approach with the same corner cases used in tasking benchmarks.

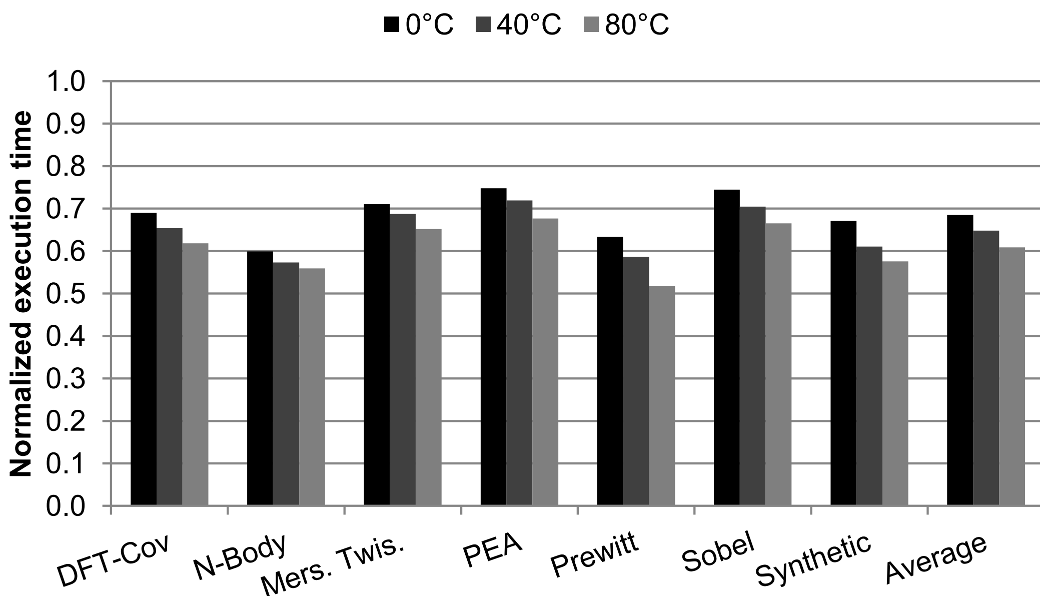

Figure 4.14: Execution time for VASS normalized to FCFS under temperature variation.

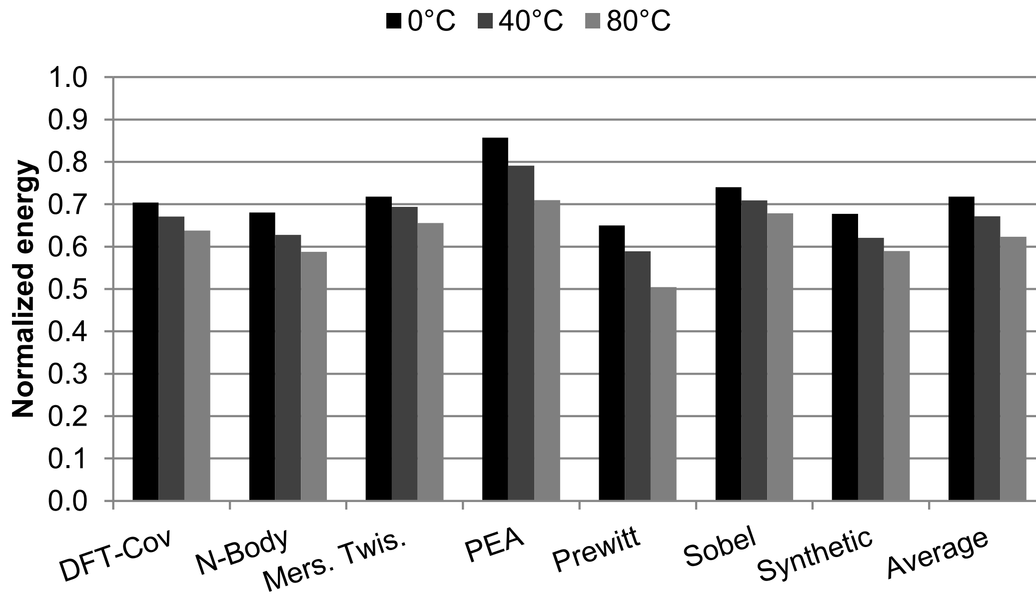

Figure 4.15: Energy consumption forVASS normalized to FCFS under temperature variation.

Fig. 4.14 shows the normalized execution time of VASS to FCFS for the same operating conditions of task. At the temperature 0°C, VASS achieves an average gain of 31% reach up to 40% in the best case. This is accomplished avoiding the worst core from the long and variation-effected section type that leads to the highest WUV. At the operating temperature of 80°C, VASS increase on average 39% performance, thanks to runtime WUV metadata characterization which reflects the latest temperature variations.

Further, as show in Fig. 4.15 VASS reduce the energy consumption for an average of 28% (up to 35%) at the operating condition of 0°C. A similar pattern for energy saving is observed under the other conditions. VASS reduce in the highest gain an average by 37% at 80°C degree of temperature.

Chapter 5

Conclusion

5.1 Tasking Conclusions

Scalability in the recent embedded many-core cluster-based accelerators is considered an important goal to achieve. Future architecture will host thousand of processing units, for this reason clustering and scalable interconnecting systems will become key aspects. Runtime system for these architectures are extremely fundamental to abstract all available resources in a scalability manner. We presented two software approaches to handle scalability in many-core cluster-based processors. In particular, we proposed two runtime to implement the tasking model suitable for both single and multi cluster architectures. We implemented our tasking model from scratch following the specification of OpenMP 3.0.

To valid the scalability of our runtime, we tested both implementations on two different architectures. In both architecture we reached a good scalability with a granularity with a workload of 5.000 ALU operations for task. We demonstrated that locality of data is maintained within cluster thus we can avoid side effects typically of NUMA architecture. In multi cluster architecture result that applying nested parallelism to the many-core cluster-based processors (in our case 64 PUs) have the same scalability of single cluster processors (in our case 16 PUs). Moreover, we demonstrated that NUMA effects on multi cluster architectures can have a huge impact on the platform bringing the speedup even for coarse-grain task.

5.2 Variability Conclusions

Timing errors caused by circuit failures are considered an important issues where all nanotechnologies architecture will be called to confront in the present and future works. In this thesis we presented our software runtime to mitigate hardware variability in many-core processor cluster. Our runtime archived good results using metadata informations (WUV) to capture vulnerability in work-unit via the software stack. We used WUV to realize a variation-aware OpenMP run-time (VOMP) adding variability-tolerant schedulers to minimize timing errors. In particular, we proposed scheduler algorithms for tasks and sections constructs that use WUV metadata to reduce timing errors. WUV metada is characterized at runtime for each individual core, and is used from the scheduler to distribute new instances of work-unit types for having efficiently scheduler. We implemented our approach extending default OpenMP runtime enables to execute efficiently in variability-affected environment. Our scheduling algorithms took in exam two fundamental constructs to determinate work-unit instances, tasks and sections that use WUV metadata to take countermeasures against timing errors. Algorithms match characteristics of different variability-affected cores and the different work-unit types in the program, minimizing the total execution cost affiliating best WUV type instances to the suitable cores even taking into account the balancing of the total workload for a determinate processor.

Afterwards, we tested our approaches on different operative conditions on fluctuation of voltage and temperature. VOMP results reduce timing errors recovery in the 16-core cluster in a wide operating temperature of 80°C, resulting in average 17% and 36% faster execution for task and sections, respectively. Moreover, VOMP reaches an average of 27% for task and 33% for sections energy saving.

Bibliography

[1] K.A. Bowman, et al., *"Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance"*, IEEE JSSC, 2009.

[2] K.A. Bowman, et al., *"A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance"*, IEEE JSSC, pp.194-208, Jan. 2011.

[3] S. Das, et al., *"Razor II: In situ error detection and correction for PVT and SER tolerance"*, IEEE JSSC, vol. 44, no. 1, pp. 32-48, Jan. 2009.

[4] M.R. Kakoee, et al., *"Variation-Tolerant Architecture for Ultra Low Power Shared-L1 Processor Clusters"*, IEEE Trans. on Circuits and Systems II, vol.59, no.12, pp.927-931, Dec. 2012.

[5] D. Jeon, et al., *"Design Methodology for Voltage-Overscaled Ultra-Low-Power Systems"*, IEEE Trans. on Circuits and Systems II, vol.59, no.12, pp.952-956, Dec. 2012.

[6] S. Dighe, et al., *"Within-die variation-aware dynamic-voltage-frequencyscaling with optimal core allocation and thread hopping for the 80-core teraflops processor"*, IEEE JSSC, 46(1): 184-193, Jan. 2011.

[7] A. Rahimi, et al., *"Procedure hopping: A low overhead solution to mitigate variability in shared-L1 processor clusters"*, Proc. ACM/IEEE ISLPED, 2012, pp. 415-420.

[8] O. Tahan, M. Shawky, *"Using dynamic task level redundancy for OpenMP fault tolerance"*, Proc. ARCS, pp. 25-36, 2012.

[9] C. Bolchini, et al., *"An adaptive approach for online fault management in many-core architectures"*, Proc. IEEE/ACM DATE, pp.1429-1432, 2012.

[10] F. Chaix, et al. *"Variability-aware task mapping strategies for many-cores processor chips"*, Proc. IEEE IOLTS, pp.55-60, 2011.

[11] H. Cho, et al., *"ERSA: Error Resilient System Architecture for Probabilistic Applications"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.31, no.4, pp.546-558, April 2012.

[12] A. Rahimi, et al. Analysis of Instruction-level Vulnerability to Dynamic Voltage and Temperature Variations, Proc. IEEE/ACM DATE, 2012.

[13] L. Benini, E. Flamand, D. Fuin, D. Melpignano, *"P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator"*, Proc. ACM/IEEE DATE, 2012, pp. 983-987.

[14] A. Rahimi, I. Loi, M. R. Kakoee, and Luca Benini, *"A fullysynthesizable single-cycle interconnection network for Shared-L1 processor clusters"*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011.

[15] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, *"Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications"*, in Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, June 2012, pp. 1137–1142.

[16] K. Bowman, J. Tschanz, N. S. Kim, J. Lee, C. Wilkerson, S. Lu, T. Karnik, and V. De, *"Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance"*, Solid-State Circuits, IEEE Journal of, vol. 44, no. 1, pp. 49–63, Jan 2009.

[17] K. Bowman, J. Tschanz, N. S. Kim, J. Lee, C. Wilkerson, S. Lu, T. Karnik, and V. De, *"Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance"*, Solid-State Circuits, IEEE Journal of, vol. 44, no. 1, pp. 49–63, Jan 2009.

[18] S. Miermont, P. Vivet, and M. Renaudin, *"A power supply selector for energy- and area-efficient local dynamic voltage scaling"*, in Proceedings of the 17th International Workshop on Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation, ser. PATMOS 07́. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 556565. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74442-9-54

[19] A. Rahimi, et al. *"Analysis of Instruction-level Vulnerability to Dynamic Voltage and Temperature Variations"*, Proc. IEEE/ACM DATE, 2012.

[20] Leon3. [Online]. Available: http://www.gaisler.com/cms

[21] A. Rahimi, D. Cesarini, A. Marongiu, R. J. Gupta and Luca Benini *"Improving Resilience to Timing Errors by Exposing Variability Effects to Software in Tightly-Coupled Processor Clusters"*, Emerging and Selected Topics in Circuits and Systems, IEEE Journal of, vol. 4, no. 2, pp.216-229, Jun 2014.

[22] TheGNUproject,GOMP—An openMP implementation for GCC [Online]. Available: http://gcc.gnu.org/projects/gomp

[23] A. Rahimi, L. Benini, and R. Gupta, *"Analysis of instruction-level vulnerability to dynamic voltage and temperature variations"*, in Design, Automat. Test Eur. Conf. Exhibit., Mar. 2012, pp. 1102–1105.

[24] G. Hoang, R. B. Findler, and R. Joseph, *"Exploring circuit timingaware language and compilation"*, in Proc. 16th Int. Conf. Archit. Support Program. Lang. Operat. Syst., NY, 2011, pp. 345–356.

[25] A. Rahimi, L. Benini, and R. Gupta, *"Application-adaptive guardbanding to mitigate static and dynamic variability"*, IEEE Trans.

[26] A. Rahimi et al., *"Variation-tolerant openmp tasking on tightly-coupled processor clusters"*, in Design, Automat. Test Eur. Conf. Exhibit., Mar. 2013, pp. 541–546. Comput., 2013.

[27] O. Tahan and M. Shawky, *"Using dynamic task level redundancy for openmp fault tolerance"*, in Proc. 25th Int. Conf. Archit. Comput. Syst., 2012, pp. 25–36.

[28] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini, *"Enabling fine-grained openMP tasking on tightly-coupled shared memory clusters"* in Design, Automat. Test Eur. Conf. Exhibit., Mar. 2013, pp. 1504–1509.

[29] S. Agathos, V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, *"Deploying openMP on an embedded multicore accelerator*, in Proc. Int. Conf. Embed. Comput. Syst.: Architect., Model., Simulat., Jul. 2013, pp. 180–187.

[30] A. Rahimi et al., *"Variation-tolerant openmp tasking on tightly-coupled processor clusters"*, in Design, Automat. Test Eur. Conf. Exhibit., Mar. 2013, pp. 541–546.

[31] P. Hoang and J. Rabaey, *"Scheduling of DSP programs onto multiprocessors for maximum throughput"*, IEEE Trans. Signal Process., vol. 41, no. 6, pp. 2225–2235, Jun. 1993.

[32] V. K. P. M. Lee and W. Liu, *"A mapping methodology for designing software task pipelines for embedded signal processing"*, Parallel Distribut. Process., pp. 937–944, 1998.

[33] A. Moreno et al., "Load balancing in homogeneous pipeline based applications," Parallel Comput., vol. 38, no. 3, pp. 125–139, 2012 [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001566.

[34] *"Plurality Ltd. The HyperCore Processor Whitepaper"*. http://www.warthman.com/projects-Plurality Architecture Shared-Memory Synchronizer-Scheduler Load-Balancing Task-Oriented-Programming Parallel-Cores.htm. April 2010.

[35] D. Melpignano et al. *“Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications”*, Design Automation Conference, 2012, pp.1137-1142.

[36] *“NVIDIA. FERMI Series Whitepaper”*. http://www.nvidia.com/content/PDF/fermi white papers/NVIDIA Fermi Compute Architecture Whitepaper.pdf. 2009.

[37] R. D. Blumofe et al. *“Cilk: An efficient multithreaded runtime system”*, in Journal of Parallel and Distributed Computing, pages 207–216, 1995.

[38] Apple, Inc. *“Grand Central Dispatch”*, https://developer.apple.com/library/mac/#documentation/Performance/Reference/GCD libdispatch Ref/Reference/reference.html. 2010.

[39] S. Kumar et al. *“Carbon: architectural support for fine-grained parallelism on chip multiprocessors”*, SIGARCH Comput. Archit. News, 35:162–173, June 2007.

[40] *“OpenMP Application Program Interface v.3.1”*, http://www.openmp.org/mp-documents/OpenMP3.1.pdf. July 2011.

[41] S. Shah, G. Haab, P. Petersen, and J. Throop, *“Flexible Control Structures for Parallelism in OpenMP”*, Proc. First European Workshop OpenMP (EWOMP ’99), Sept. 1999.

[42] F. Massaioli, F. Castiglione, and M. Bernaschi, *“OpenMP Parallelization of Agent-Based Models”*, Parallel Computing, vol. 31, nos. 10-12, pp. 1066-1081, 2005.

[43] F.G.V. Zee, P. Bientinesi, T.M. Low, and R.A. van de Geijn, *“Scalable Parallelization of FLAME Code via the Workqueuing Model”*, ACM Trans. Math. Software, submitted, 2006.

[44] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, and J. Labarta, *“Nanos Mercurium: A Research Compiler for OpenMP”*, Proc. Sixth European Workshop OpenMP (EWOMP ’04), pp. 103-109, Sept. 2004.

[45] J. Reinders, *“Intel Threading Building Blocks”*, O’Reilly Media Inc., 2007.

[46] D. Leijen and J. Hall, *“Optimize Managed Code for Multi-Core Machines”*, MSDN Magazine, pp. 1098-1116, Oct. 2007.

[47] R. Blikberg and T. Sørevik, “Load Balancing and OpenMP Implementation of Nested Parallelism”, Parallel Computing, vol. 31, nos. 10-12, pp. 984-998, 2005.

[48] S. Salvini, *“Unlocking the Power of OpenMP”*, Proc. Fifth European Workshop OpenMP (EWOMP ’03), invited lecture, Sept. 2003.

[49] F.G.V. Zee, P. Bientinesi, T.M. Low, and R.A. van de Geijn, *“Scalable Parallelization of FLAME Code via the Workqueuing Model”*, ACM Trans. Math. Software, submitted, 2006.

[50] J. Kurzak and J. Dongarra, *“Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead”*, Dept. Computer Science, Univ. of Tennessee, LAPACK Working Note 178, Sept. 2006.

[51] E. Ayguad et al. The Design of OpenMP Tasks. IEEE Trans. Parallel Distrib. Syst., 20(3):404–418, Mar. 2009.

[52] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro, “Exploiting multiple levels of parallelism in openmp: a case study,” in Parallel Processing, 1999. Proceedings. 1999 International Conference on, 1999, pp. 172 –180.

[53] S. Karlsson, “A portable and efficient thread library for openmp,” in In Proc. 6th European Workshop on OpenMP, KTH Royal Institute of Technology. John Wiley, 2004, pp. 43–47.

[54] X. Martorell, E. Ayguade, N. Navarro, J. Corbaln, M. Gonzlez, and J. Labarta, “Thread fork/join techniques for multilevel parallelism exploitation in numa multiprocessors,” in inNUMA Multiprocessors. In 13th Int. Conference on Supercomput- ing ICS’99, Rhodes, 1999, pp. 294–301.

[55] P. E. Hadjidoukas and V. V. Dimakopoulos, “Nested parallelism in the ompi openmp/c compiler,” in Proceedings of the 13th international Euro-Par conference on Parallel Processing, ser. Euro-Par’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 662–671. [Online]. Available: http://dl.acm.org/citation.cfm?id=2391541.2391620

[56] A. Marongiu, A. Capotondi, G. Tagliavini, and L. Benini, “Improving the programmability of STHORM-based heterogeneous systems with offloadenabled OpenMP,” in Proceedings of the First International Workshop on Many-core Embedded Systems - MES ’13. New York, New York, USA: ACM Press, 2013, pp. 1–8.