KTH Information and
Communication Technology

Improving Performance and Quality-of-Service through the Task-Parallel Model

Optimizations and Future Directions for OpenMP

ARTUR PODOBAS

Doctoral Thesis in Information and Communication technology
KTH Royal Institute of Technology
School of Information and Communication Technology

KTH
TRITA-ICT 2015:13
SE-100 44 Stockholm
ISBN 978-91-7595-711-1
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av doktorexamen 2015-11-10 i KTH Forum Kista.

Tryck: US-AB

Abstract

With the failure of Dennard's law, which stated that shrinking transistors will be more power-efficient, computer hardware has today become very divergent. Initially the change only concerned the number of processor on a chip (multicores), but has today further escalated into complex heterogeneous system with non-intuitive properties – properties that can improve performance and power consumption but also strain the programmer expected to develop on them.

Answering these challenges is the OpenMP task-parallel model – a programming model that simplifies writing parallel software. Our focus in the thesis has been to explore performance and quality-of-service directions of the OpenMP task-parallel model, particularly by taking architectural features into account.

The first question tackled is: what capabilities does existing state of the art runtime-systems have and how do they perform? We empirically evaluated the performance of several modern task-parallel runtime-systems. Performance and power-consumption was measured through the use of benchmarks and we show that the two primary causes for bottlenecks in modern runtime-systems lies in either the task management overheads or how tasks are being distributed across processors.

Next, we consider quality-of-service improvements in task-parallel runtime-systems. Striving to improve execution performance, current state of the art runtime-systems seldom take dynamic architectural features such as temperature into account when deciding how work should be distributed across the processors, which can lead to overheating. We developed and evaluated two strategies for thermal-awareness in task-parallel runtime-systems. The first improves performance when the computer system is constrained by temperature while the second strategy strives to reduce temperature while meeting soft real-time objectives.

We end the thesis by focusing on performance. Here we introduce our original contribution called BŁYSK – a prototype OpenMP framework created exclusively for performance research.

We found that overheads in current runtime-systems can be expensive, which often lead to performance degradation. We introduce a novel way of preserving task-graphs throughout application runs: task-graphs are recorded, identified and optimized the first time an OpenMP application is executed and are later re-used in following executions, removing unnecessary overheads. Our proposed solution can nearly double the performance compared with other state of the art runtime-systems.

Performance can also be improved through heterogeneity. Today, manufacturers are placing processors with different capabilities on the same chip. Because they are different, their power-consuming characteristics and performance differ. Heterogeneity adds another dimension to the multiprocessing problem: how should work be distributed across the heterogeneous processors? We evaluated the performance of existing, homogeneous scheduling algorithms and found them to be an ill-match for heterogeneous systems. We proposed a novel scheduling algorithm

that dynamically adjusts itself to the heterogeneous system in order to improve performance.

The thesis ends with a high-level synthesis approach to improve performance in task-parallel applications. Rather than limiting ourselves to off-the-shelf processors – which often contains a large amount of unused logic – our approach is to automatically generate the processors ourselves. Our method allows us to generate application-specific hardware from the OpenMP task-parallel source code. Evaluated using FPGAs, the performance of our System-on-Chips outperformed other soft-cores such as the NiosII processor and were also comparable in performance with modern state of the art processors such as the Xeon PHI and the AMD Opteron.

Sammanfattning

Ända sedan Dennard's förutsägelse om att mindre transistorer även förbrukar mindre ström avtog har processorer med flera kärnor blivit den främsta kandidaten att förbättra datorers prestande. Idag är i princip alla processorer så-kallade *multicores*, alltså processorer som i sig innehåller ett flertal mindre processor som delar utrymme och minne med varandra.

Detta paradigmskift medförde stora förändringar. Historiskt sett har man utvecklat mjukvara med seriel programkörning i åtanke. Att nu behöva tänka parallellt när man utvecklar program är svårt och tidskrävande, åtminstone med de flesta utvecklingsverktygen som finns.

Ett verktyg som faktiskt underlättar utveckling av parallella program är OpenMP. OpenMP kräver väldigt få ändringar i ursprungsprogrammet, har en enkel programmeringsmodel och är portabel, vilket har gjort den till en av de mer framgångsrika parallella programmeringsmodellerna. Framför allt så ger OpenMP tillgång till konceptet *tasks*. Konceptet har blivit en populär metod vid parallellisering av program eftersom det erbjuder asynkron och komposerbar parallellism.

I arbetet presenterat i denna avhandling behandlar vi framtida utökningar av OpenMP och liknande, *task-parallella*, programmeringsmodeller. Vi har undersökt hur dagens moderna task-parallella runtime-system presterar och vad som skiljer dem åt. Genom benchmarks och små specialskrivna program har vi visat att den främsta orsaken till dålig prestanda är schemaläggningen och omkostnader i att underhålla parallellismen i programmen.

Vidare har vi förbättrat prestandan genom att ta temperatur i beaktande vid schemaläggning av tasks på processorer. Genom att ge *runtime*-systemet – alltså den modul som bestämmer vilken processor som skall utföra vilket arbete – åtkomst till ändringar i processorns temperatur har vi visar hur schemaläggning kan undvika att processorn överhettas, vilket leder till en förhöjd prestanda. Vi har även undersökt hur man kan minska värmeutvecklingen vid real-tids mål genom att försätta processorer i viloläge om de inte behövs.

Avhandlingen avslutas med fokus på prestanda. Dagens och framför allt framtidens system kommer att bli med heterogena. Med heterogena menas att ett flertal olika processor kommer dela samma yta kisel. Ett exempel på heterogenitet i dagsläget är system där grafikkortet är inbyggda i processorn. Heterogena lösningar kan prestera väldigt bra givet att man vet deras begränsningar. Vi har undersökt hur man schemalägger arbete på ett system som innehåller tre distinkt olika processorer. Genom att dynamiskt mäta deras prestanda under programkörning tar vi reda på var och när arbete skall schemaläggas för bästa resultat.

Avhandlingen avslutas med frågan: kan man använda OpenMP för att generera hårdvara som accelererar det parallela programmet? Vårat arbete har visat hur OpenMP kan fungera som model för att skapa hårdvara med hög potential att utnyttja parallellism. De systemen som våran metod genererar har liknande eller bättre prestanda än dagens moderna Intel eller AMD processorer.

Acknowledgments

I would like to start by thanking my main supervisor, Mats Brorsson, who has guided me, given me positive feedback and encouraged me through the entire PhD-studies. I would also like to thank my secondary supervisor Vladimir Vlassov, for the help and support he has offered me. Tremendous thanks to all my co-authors – you know who you are (if you forgot, then you can find it on the next page)!

Next is Ananya Muddukrishna, my closest colleague. Together we have shared moments of despair, calamity, joy and happiness throughout these five years – you are like no other person I know of.

Thanks to Georgios Varisteas and Ahsan Javed Awan – both colleagues of mine – who have helped make every day seem a tad bit darker. Thanks!

I would like to thank all colleagues at my KTH department and at SICS – it is a privilege to have worked with and among such great minds.

My family, including Daniel, Teresa and Jerzy Podobas, to whom I still cannot explain what exactly I am researching. Needless to say, they have been there not only throughout my studies, but throughout my life. Particular thanks goes to my grandmother, Chryszanta Kasperska, whom I hope is watching me finish this from afar.

Finally, my love Linda Schenk, who has been there for me, listening, giving me advice's, cheered me on and helped me. The 51554 words of this thesis are insufficient to describe my appreciation.

Thank you!

Funding

This thesis and its work have been funded by the European Communities Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement nr. 248647 and the Artemis PaPP projected nr. 295440.

Contents

List of papers included in the thesis

Paper I: A comparative performance study of common and popular taskcentric programming frameworks

Published: *Concurrency and Computation: Practice and Experience 27 (Journal, Wiley)*
Authors: Artur Podobas, Mats Brorsson, Karl-Filip Faxén
Author contribution: Designed and performed experiments, analysis and wrote the paper.

Paper II: Architecture-aware Task-scheduling: A thermal approach

Presented: *First Workshop of Future Architecture Support for Parallel Programming, FASPP'11. Held in conjunction with ISCA'11*
Published: *In proceedings of the First Workshop of Future Architecture Support for Parallel Programming, FASPP'11*
Authors: *Artur Podobas, Mats Brorsson*
Author contribution: Formed the hypothesis, designed and conducted the experiments, analyzed results and wrote the paper.

Paper III: Considering Quality-of-Service for Resource Reduction using OpenMP

Presented: *Programmability Issues for Heterogeneous Multicores 2014, MULTIPROG'14. Held in conjunction with HiPEAC'14*
Authors: *Artur Podobas, Mats Brorsson, Vladimir Vlassov, C.C. Chi, Ben Juurlink*
Author contribution: Formed the hypothesis, designed and conducted the experiments, analyzed results and wrote the paper.

Paper IV: Exploring heterogeneous scheduling using the task-centric programming model

Presented: *Eleventh International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, HeteroPAR'12. Held in conjunction with EuroPAR'12*
Published: *Euro-Par 2012: Parallel Processing Workshops (Springer)*
Authors: *Artur Podobas, Mats Brorsson, Vladimir Vlassov*
Author contribution: Formed the hypothesis, designed and conducted the experiments, analyzed results and wrote the paper.

Paper V: TurboBŁYSK: Scheduling for Improved Data-Driven Task Performance with Fast Dependency Resolution

Presented: *10th International Workshop on OpenMP, IWOMP'14*
Published: *Using and Improving OpenMP for Devices, Tasks, and More (2014, Springer)*
Authors: *Artur Podobas, Mats Brorsson, Vladimir Vlassov*
Author contribution: Formed the hypothesis, designed and conducted the experiments, analyzed results and wrote the paper.

Paper VI: Accelerating Parallel Computations with OpenMP-Driven System-on-Chip Generation for FPGAs

Presented: *8th International Symposium on Embedded Multicore/Manycore SoCs, MCSoC'14*
Published: *In proceedings of the 8th International Symposium on Embedded Multicore/Manycore SoCs, MCSoC'14 (IEEE)*
Authors: *Artur Podobas*
Author contribution: Wrote the paper by myself.

Paper VII: From software to parallel hardware through the OpenMP programming model

Submitted to: *22nd IEEE Symposium on High Performance Computer Architecture, HPCA'16*
Authors: *Artur Podobas, Mats Brorsson*
Author contribution: Formed the hypothesis, designed and conducted the experiments, analyzed results and wrote the paper.

Other relevant but not included publications

Investigating the Potential of Energy-savings Using a Fine-grained Task Based Programming Model on Multi-cores

Presented: *2nd Workshop on Applications for Multi and Many Core Processors, A4MMC'11*
Authors: *Alexandru Iordan, Artur Podobas, Lasse Natvig, Mats Brorsson*

Task scheduling on manycore processors with home caches

Presented: *First workshop on On-chip memory hierarchies and interconnects: organization, management and implementation, OMHI'12*
Published: *Euro-Par 2012: Parallel Processing Workshops (Springer)*
Authors: *Ananya Muddukrishna, Artur Podobas, Mats Brorsson, Vladimir Vlassov*

Using Transactional Memory to Avoid Blocking in OpenMP Synchronization Directives

Presented: *11th International Workshop on OpenMP Workshop, IWOMP'15*
Published: *To appear in proceedings of IWOMP'15 (Spriger)*
Authors: *Lars Bonnischen and Artur Podobas*

Chapter 1

Introduction

The performance of a processor is dependent on the clock frequency that drives it. Because nearly all processors are synchronous (driven by a clock), the width of the clock pulse determines the minimum time for the computer to perform an operation – the higher the frequency, the faster does the processor perform instructions.

Twentieth century processors could continuously increase the clock-frequency for each new generation. Improvements in manufacturing technology led to a doubling of transistors roughly every second year. Improved computer architectures allowed pipelining which divided heavy operations in the processor to span several clock cycles, which allowed an increased clock frequency.

The continuous increase in frequency was beneficial for hardware and software developers alike. Programmers could continue to use legacy code while benefiting from better performance for each new processor generation. Hardware manufacturers could follow the transistor technology and reuse the same designs with smaller transistors to gain performance.

Continuing to increase the clock frequency was stopped in the twentieth century. Moore's law [1] was still followed, with transistor doubling every 18 months. So why is processor frequency no longer increased? Power consumption is the answer why.

The power consumption of any piece of digital circuitry consists can be briefly summarized with the following equation:

$$P = \alpha C V^2 * f + V I_{leak} \text{ [2, 3].}$$

The dynamic part of the power consumption is related to how often the transistors switch from logical zero to logical one (α,when the transistor draws current to charge a capacitive node up), how many transistors we have (the area or capacitance C), the frequency f and the square of the operating voltage (V^2).

Because the power consumption is directly proportional to the frequency, doubling the frequency means doubling the dynamic power consumption. As previously

stated, one way to increase the clock frequency from an architecture perspective is to split heavy operations so that they overlap several clock cycles. This is called pipelining. The higher the clock-frequency, the more pipeline-stages are required to reduce the critical-path of the processor, where the time to propagate through the critical-path is equal to the clock pulse-width. However, more pipeline stages makes it expensive to restart computations currently in execution. For example, a wrongly predicted branch on a processor with 40 pipeline stages wastes a lot of time to flush everything out of the pipeline.

Another way to increase the clock-frequency is to increase the operating voltage. Because the time to charge up a capacitance is proportional to the voltage (the current drawn from the voltage source), the higher the voltage the faster transistor switch. But increasing the voltage squares the dynamic power consumption (because the power is proportional to V^2).

The driving discovery that ended single-core processors was that Dennard scaling [4] – which states that each new generation of transistors will operate on a lower voltage – has failed. Since Moore's law was still being followed – with a doubling of transistors every 18 months – the power that was consumed had to be dissipated on a continuously shrinking chip die area. This lead to a temperature problem, which is also why today's processor have so much more cooling attached to them compared to decades ago.

Given the power and thermal problems, the only way to progress was to reduce the frequency, reduce the voltage while increasing the performance – the "free lunch is over" [5] for software and hardware developers alike. Answering the challenges, the multicore processor was born.

A multicore processor is composed of several processing elements on the same chip-die. Rather than spending silicon on expensive branch-predictors, deep pipelines, larger caches or complex speculative out-of-order schemes, the silicon is spent on replicating several simple cores. Because they are simple, the processor can work with lower clock frequency and thus a reduced voltage yielding a smaller power budget. The silicon is better spent on adding many such simple cores rather that adding features to one large core, as it yields higher performance and is less power-hungry– assuming that we can execute programs in parallel.

The weaker processor cores can be used to improve program execution time. Amdahl's law [6] describes how the execution time of a program is improved when an application becomes parallel:

$$Speedup = \frac{1}{f_{serial} + \frac{f_{parallel}}{N}}$$

Here, f_{serial} is the fraction of the application running sequentially and $f_{parallel}$ is the fraction of the application capable of executing in parallel divided over the N number of processing elements. Amdahl's law implies that by parallelizing our serial application as much as possible, we can exploit multiple cores to reduce the execution time.

Today the multicore processors has fully replaced the single-core processor in general purpose processor market by enabling higher performance and a reduced power-budget – and new challenges.

Programming for parallel, shared-memory processor has historically been performed through the thread-parallel model. Programmers were exposed to threads and had to work closely with the architecture to define and orchestrate parallel execution. Abstraction was needed, as portability and performance of manually managing parallelism was difficult to attain and maintain.

Programming multicore processors is today assisted by programming models – language extensions and libraries that simplify the programmer's job to create portable and efficient parallel code. The task-parallel model is a popular programming model that has transcended the limitations of the thread-parallel model. Offering asynchronous parallelism in the shape of fork/join and data-flow parallelism, the task-parallel model copes with many parallel computation patterns and is well-prepared for future challenges such as exploiting heterogeneous systems.

One of the challenges concerns heterogeneity. Aware of the end of Dennard scaling and the power-consuming threats of dark silicon [7] – all transistors on the chip cannot be activate at the same time – chip designers have proposed use of accelerators such as general-purpose graphics cards (GPUs) [8] or the Intel Xeon Phi [9] to deal with specific parallel patterns. But how do we distribute the computations across such heterogeneous systems? How do we extend the task-parallel runtime-system to support heterogeneity?

One direction of heterogeneity that has been explored in the thesis is High-Level Synthesis [10]. Emerging and existing processors such as Xilinx Zynq [11] or Intel-Altera HARP [12] that contains both general purpose processors and Field-programmable gate-arrays call for a change in programming models – a change from a fully software approach to a hybrid software/hardware approach. Today, the task-parallel model is exclusively used for writing software. Our work have extended it to also drive hardware generation, which is needed to exploit future heterogeneous architectures.

Another challenge is the thermal issue [13]. Temperature is a first-class constraint in computer systems. Processors today use various frequency and voltage scaling schemes to manage temperature – when a processor is overloaded and reaches a certain temperature, its performance is crippled until it cools down. However, current parallel models are oblivious of the temperature of the underlying hardware. Because most task-parallel models are targeting performance, they can quickly heat the processor up, which leads to performance loss. How could a parallel runtime-system utilize knowledge about the temperature to improve performance or other objectives?

This thesis deals exclusively with the task-parallel programming model and its materialization in the OpenMP framework– a directive-driven framework originally proposed to handle parallelism in shared-memory nodes in high-performance computers.

1.1 Contributions

The contributions of this thesis are:

- An evaluation of modern task-parallel runtime-systems with respect to their performance and power consumption properties in order to reason about their performance trade-offs
- A thermal-aware task-parallel scheduler that balances the temperatures of processor cores in order to improve execution time in a thermally stressed environment
- A task-parallel scheduler that provides soft real-time guarantees in OpenMP tasks and attempts to minimize the overall temperature of underlying processors
- An evaluation of commonly used homogeneous scheduling techniques on a highly divergent heterogeneous system and a proposed scheduling algorithm that improves execution time performance under such a heterogeneous system.
- An novel OpenMP-based task-parallel runtime-system, which allows recording of tasks' dependency patterns to to improve OpenMP data-flow performance
- Methods for using OpenMP to drive hardware generation through High-Level Synthesis, allowing software programmers to automatically generated highly-parallel computer hardware that improves performance of task-parallel applications

1.2 Layout

Chapter 2 presents a background on multiprocessor programming and dives deeper into the task-parallel programming model used in OpenMP. Chapter 3 summarizes the publications of the thesis, briefly explaining why, what and how the studies were conducted. We conclude and discuss future work in Chapter 4. The remainder of the thesis includes the peer-reviewed original contributions.

Chapter 2

Background

2.1 Flynn's Taxonomy and Parallelism today

Today there are multiple ways of utilizing parallelism in computer architectures, ranging from programmer-transparent automatic schemes inside processors to explicit task-based parallelism that require interaction from the programmer. Most of these parallelization strategies can be describes by three out of the four concurrent classes described by Flynn [14], and are summarized below:

Single Instruction Single Data (SISD, Figure 2.1:a)

Even before the end of the single-core processor, architects developed techniques to exploit instruction-level parallelism inside a processor. Out-of-Order (OO) instruction scheduling, a technique for automatic tracking dependencies between instructions to detect possible parallelism without the intervention of the programmer is a common technique used in high-performance processors today. Dating back to the 1960's [15], out of order instruction scheduling can today be found in nearly all non-embedded general purpose processors such as Intel x86/x64, SPARC-v9 and IBM Power architectures.

Embedded processors are also seeing instruction-level parallelism today. Here, the trend is to push the work of instruction scheduling over to the compiler and adding very-long instruction-word support (VLIW) for processors [16]. In a VLIW architecture, several instructions are compressed and executed in parallel on different resources in the processor. Unlike Out-of-Order execution, which is relatively expensive (area-wise) to implement, VLIW is much cheaper and relies quite heavily on the compiler to construct tight program code. VLIW today exists in for example Tilera TILE64/TILEPro64 [17], Hexagon [18] and Intel Itanium [19].

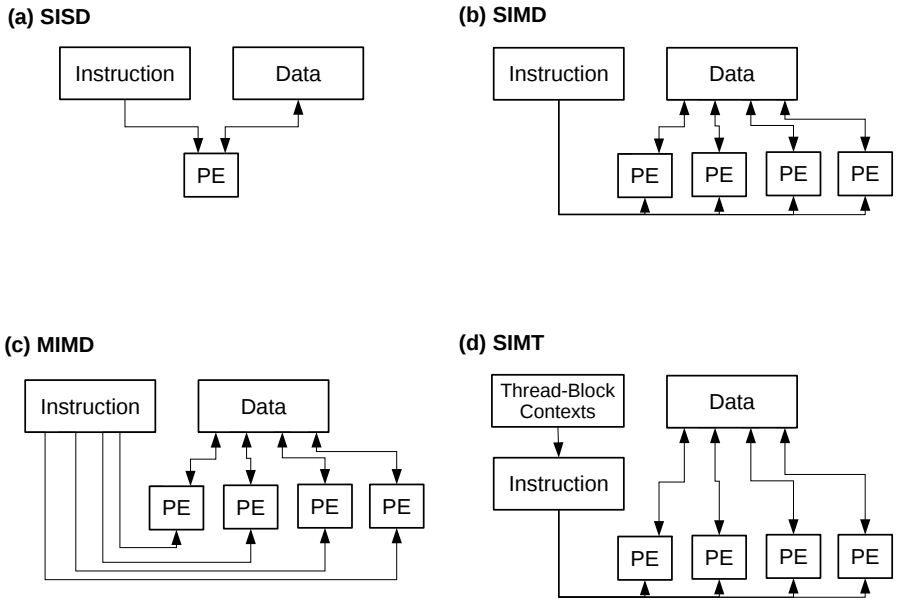


Figure 2.1: Various types of parallelism in today's computer architectures

Single Instruction Multiple Data (SIMD, Figure 2.1:b)

SISD-architectures such as the general purpose processor today often implement some form of Single Instruction Multiple Data functionality (SIMD), also known as vector-processing. SIMD differs from SISD in that the very same instruction(s) are applied simultaneously to different data. The motivation to include SIMD functionality in processor design is intuitive: algorithms often perform the same operation on large arrays of data. Rather than serially performing the computation one data-item at a time, it is better to delegate computation of different data points to different but identical resources in the processor.

SIMD can speed calculations up. However, unlike VLIW or OO parallelism, they still rely on explicit programmer help because compiler support for automatic vectorization is limited. SIMD instructions occurred as early as the vector-processors in the 1970's such as the Cray-1 [20]. Today, most general purpose processors contain SIMD functionality: Intel have MMX/AVX instructions, MIPS have MSA and ARM use the NEON co-processor.

Recently a derivate of the SIMD parallel class has emerged called Single Instruc-

tion Multiple Threads (SIMT) [21]. SIMT (Figure 2.1:d) applies to the massively parallel General Purpose Graphics Processor Units (GPGPUs) that have become popular to use in accelerating very specific parallel patterns. The difference between SIMT and SIMD is how the vector-units are exposed: in SIMD they are exposed to the software (the programmer must explicitly state where and how SIMD operates) while in SIMT they are not [22] (all threads will execute the kernel). Additionally, SIMT allows thread-level parallelism since each thread have its own control flow.

A GPU usually has a large number of threads (often in the order of thousands). These threads are grouped into thread-blocks [23], and each thread-blocks executes the very same instruction. Resources are shared between thread-blocks, whose execution is often interleaved. GPUs focus on hiding latency and to increase throughput, which is the opposite to general purpose processors that tend to minimize latency instead. Programming GPUs is often done using either nVidia's CUDA [8] or OpenCL [24]. Programs are written in Single Program Multiple Data fashion (SPMD) [25]– all threads in the GPUs will execute the very same program on different data-sets.

Multiple Instruction Multiple Data parallelism (Figure 2.1:c)

The final class of concurrency is the Multiple Instructions Multiple Data (MIMD) class. Multicore/multiprocessor systems belong to the MIMD class. Each processing element has its own execution content (their own program counter and memory stack) from which instruction and data are fetched, asynchronous to what other processing elements are doing.

There are many programming models that can exploit MIMD architectures, but we will focus on explaining the two most commonly used for shared-memory architecture, in particular the task-parallel model which is the primary focus of this thesis.

Thread-level parallelism

The concept of a thread is basic to computer systems. A thread is a representation of an execution context as provided by an operating system. In its basic form, it provides a stack, a copy of the processors internal register and program counter. Historically, even before the advent of common multiprocessors, threads were used to provide software layers with concurrency. There need not be a one-to-one correspondence between the number of threads and the number of processors – threads can time-share processors through interrupts.

One of the low-level ways to exploit thread-level parallelism in an application is through POSIX threads (Pthreads) [26]– an interface native to UNIX-like operating systems. Pthreads offer the programmer an API for creating threads and joining (synchronizing) them. Because it is minimalistic, programmers are required to manually orchestrate the entire computation, which is often undesirable. Nonetheless, Pthreads still remain one of the more popular approaches to parallelization,

existing in everything from webservers to video decoders [27, 28]. More abstract models such as OpenMP often use Pthreads inside the runtime-system.

A simple example on how to parallelize the iterative calculation of Prime numbers is shown in Figure 2.2:a using Pthreads. We split the original for-loop and place it in a separate function definition (lines: 4-16). We then create three threads, where each thread is responsible for calculating over a subset of the entire iteration span (lines: 24-36); the caller (starter) thread is the last thread to call the `prime_thread_calc()` function. Once all threads have been started, the program waits until all threads are finished before exiting the program (lines: 38-40).

The weaknesses in using Pthreads for parallel computation is shown in the Figure 2.2:c speed-up graph. The speed-up graph is the increase in performance of a multithreaded application over its sequential version. The problem given is fairly simple, yet the performance of the application does not scale with the number of threads (four threads) given to it. Ideally, with the embarrassingly parallel application shown here, the performance should scale linearly with the amount of threads up to the number of physical number of processing elements available in the system. Four physical cores were available as it was executed in a four-core Intel Nehalem processor, yet the performance only barely reaches twice that of the serial version.

Additionally, the programmer is directly exposed to the hardware, requiring to use low-level protection for variables accessible by different threads to avoid data-races [29]. Data-races are one of the challenges that programmers experience when parallelizing software. Data-races occurs when processing elements locally update copies of the same memory location, leading to incorrect results when the data is written back to memory. Protection in the example code is achieved through hardware-supported atomicity (lines: 13-14) but portability is not guaranteed – architectures with no support for atomicity will be not be able to run the code. Similarly, architectures with fewer than four cores will suffer performance degradation (due to over-subscription) because of thread context-switching overheads, while architectures with more cores will be underutilized (due to under-subscription).

The reason for the poor performance in the example case above is load imbalance. Because the iterations in the example are non-uniform in computation cost (calculating higher primes is more expensive than lower primes), some of the threads will have a larger workload to perform while other threads will finish quickly.

Load balancing is but one of the many challenges parallel programmer face today. Other problems include race-conditions, memory consistency, false-sharing and heterogeneity. For example, a Pthreads application written for the x86 total-store-order (TSO) memory model [30] will likely not work on the PowerPC's relaxed consistency memory model [31]. While possible, tackling these problems in Pthreads is difficult and not portable – all architecture behaves differently, and problems can be more or less pronounced when moving across architecture. There is a need to assist programmers to become more productive when writing parallel programs, motivating the creation of new more abstract parallel programming models.

OpenMP [32] is one such programming model, developed to assist programmers in creating portable and well-performing program code. OpenMP is driven by compiler directives (`#pragma's`), requiring little change in the original serial code in the parallelization effort. Figure 2.2:b shows the OpenMP code for the prime-calculation kernel. Here, we started with the original sequential code and only added a compiler directive (lines: 7-8) describing that the for-loop can execute in parallel. The impact on the source-code is marginal. While data protection is still explicit through the `reduction()` clause, the act of portability now lies within the OpenMP runtime-system rather than the programmer. Performance-wise the OpenMP program is better compared to the more manual Pthread version, and the code is portable. For example, the programmer needs not to explicitly state how many threads should be used – the OpenMP runtime-system will automatically detect the number of processor cores available in the system executing the application.

Still, thread-level parallelism exposes the concepts of threads to application developer. Both the Pthreads and the thread-parallel OpenMP model require information concerning what can be scheduled on threads and, in most cases, hints about how the work is scheduled. There was a need for a model where the parallel work exposed by the programmer is decoupled from how the work is scheduled onto the available threads– a task-parallel model.

Task-level parallelism

There are several models that exist for task-level parallelism. OpenMP, starting from version 3.0 [33], supports task-based parallelism. There are several OpenMP implementation supporting the 3.0 standard, including compilers from Intel, GNU and Oracle/Sun, as well as academic frameworks such as OpenUH [34], Rose-based compilers [35] and Mercurium [36].

Cilk-5 [37], the pre-cursor of Intel Cilk+, was one of the early research models for task-level parallelism. Tasks in the Cilk runtime-system are always immediately executed, leaving the parent task available to the runtime-system. This style of scheduling, called strands in Cilk-5 and untied in OpenMP, offers good caching characteristics and has been proved to be optimal [38] when ignoring memory effects and under the influence of infinite amount of processing elements.

Intel's Threading Building Blocks [39] is a C++ library for exposing parallelism. Unlike other compiler directive-driven models, which requires some sort of compiler support, TBB works with any C++ supporting compiler. Wool [40] is yet another task-based runtime-system, specializing in small overheads in a work-stealing environment.

Task-level parallelism offers programmers the possibility to expose parallelism that is asynchronous, composable and unbound to a particular processing element, and is easier to use than the POSIX thread model [41]. A task is a collection of sequential instructions that can be executed asynchronously with other tasks. Tasks can run in parallel with other tasks and can spawn additional parallelism

POSIX Threads
Thread-Level Parallelism (a)

```
1  #include <stdio.h>
2  #include <pthread.h>

3  int primes = 0;

4  void *thread_prime_calc ( void *start)
5  {
6    int i,j;
7    for ( i = *((int *) start);
8          i < (*((int *) start))+25000;
9          i++) {
10     for (j = 2; j < i; j++)
11       if ( !(i%j) )
12         break;
13     __sync_fetch_and_add ( &primes ,
14                            (i == j) );
15   }
16 }

17 int main(int argc, char *argv[])
18 {
19   int i,j;
20   pthread_t Threads[3];
21   int ThreadIteration[4] = {0, 25000,
22                             50000, 75000};
23
24   pthread_create ( &Threads[0] , NULL,
25                    thread_prime_calc,
26                    &ThreadIteration[0]);
27
28   pthread_create ( &Threads[1] , NULL,
29                    thread_prime_calc,
30                    &ThreadIteration[1]);
31
32   pthread_create ( &Threads[2] , NULL,
33                    thread_prime_calc,
34                    &ThreadIteration[2]);
35
36   thread_prime_calc ( &ThreadIteration[3]);
37
38   pthread_join(Threads[0], NULL);
39   pthread_join(Threads[1], NULL);
40   pthread_join(Threads[2], NULL);
41 }
```

OpenMP
Thread-Level Parallelism (b)

```
1  #include <stdio.h>
2  #include <omp.h>

3  int primes = 0;

4  int main(int argc, char *argv[])
5  {
6    int i,j;
7  #pragma omp parallel for reduction(+:primes) \
8                  firstprivate(j) schedule(dynamic)
9    for ( i = 2; i < 100000; i++) {
10     for (j = 2; j < i; j++)
11       if ( !(i%j) )
12         break;
13     primes += (i == j);
14   }
15 }
```

Thread-Level Parallelism
Performance (c)

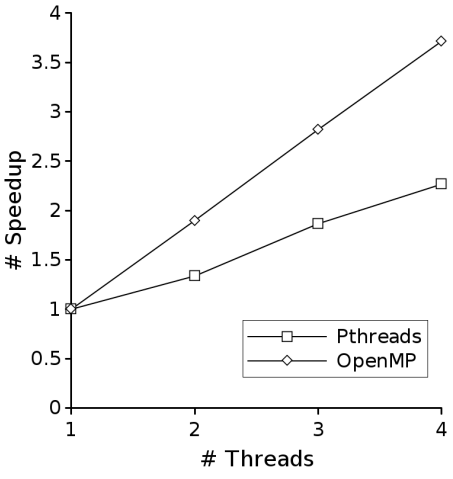


Figure 2.2: An example of a simple prime-number calculation application using POSIX threads (a) and OpenMP (b), and their respective performance (c).

(a) Original code

```
p = listhead;
while (p) {
  process(p);
  p = next(p);
}
```

(b) POSIX Thread-parallel version

```
#include <pthread.h>

p = listhead;
while (p) {
  pthread_t thr;
  pthread_create (&thr,NULL,    \
                  &process, p);
  p = next(p);
}
```

(c) Incorrect OpenMP Thread-parallel version

```
#include <omp.h>

p = listhead;
#pragma omp parallel for
while (p) {
  process(p);
  p = next(p);
}
```

(d) OpenMP Task-parallel version

```
#include <omp.h>

p = listhead;
#pragma omp parallel
#pragma omp single
while (p) {
  #pragma omp task
  process(p);
  p = next(p);
}
#pragma omp taskwait
```

Figure 2.3: A parallel pattern that is unproductive and performance-ineffective to parallelize using thread-parallel models, but easily made parallel with the task model. Problem derived from Ayguade et al. [33].

Tasks allows for parallel patterns that are difficult or impossible to expose using the thread-parallel model. One such motivating example is the parallel list traversal case in Figure 2.3:a. If the programmer has a list that is traversed and the computation performed on each element in the list can be computed in parallel, how can the programmer parallelize such a pattern? One approach would be to spawn a POSIX Thread for each element (Figure 2.3:b), but this would quickly degrade performance because of the overheads in maintaining threads inside the operating system [42]. The OpenMP thread-parallel construct cannot help here (Figure 2.3:c), as they require a-priori (prior to the computation starts) knowledge about how many iterations exist – since the example is traversing a list, the number of elements is unknown.

The task-parallel model solves this by processing each element as a task (Figure 2.3:d). A task can be exposed in nearly the same time it takes to call the computation itself, meaning it is a very lightweight construct [43]. When a task is exposed, it is submitted to a runtime-system. The runtime-system will decide how, when and where the task will be executed. Often the task is placed inside a queue, letting existing threads poll for the work themselves. If the threads are polling uninhibitedly for work, the *scheduling algorithm* [44] is said to be *greedy* [45]. It is the scheduling algorithm that decides which action threads take with respect to tasks. *Work-stealing* [46, 38] is the most commonly used scheduling algorithm, in which idle threads steal tasks from other, more loaded, threads.

Another parallel pattern that is often used with the task-parallel model is recursive parallelism. By creating a single task that by itself create more parallelism, problems can be easily decomposed. Recursive parallelism also offers control over the amount of parallelism exposed, which is a quick way to tune an application when migrating it to a new architecture. Here, control is often in the form of a cut-off, most commonly used a *depth-based* cutoff [47]. When the recursive tree reaches a certain depth, tasks are no longer created but instead are executed serially. The *span* [48] or critical-path (the longest task-path in the task-graph) is controlled through cutoffs.

Another difference between the task- and thread-parallel models is how threads are viewed. Thread-parallel models often rely on the operating system to balance/migrate threads across the system. In the task-parallel models, we prefer threads to be pinned to cores and never relocated, which is why many task-based runtime-system limits the number of threads to the amount of processing elements in the system. Threads in the task-parallel model are often viewed as proxies for processor cores, meaning that each thread should be executed in isolation at a hardware processing core and not be migrated/timeshared.

Recursively calculating Fibonacci's series has historically been the most common way of showing how (recursive) task-level parallelism is used and is repeated in Figure 2.4:a. The first time the `fib` function is called, it recursively spawns two new tasks through the `omp task` directives (lines: 6-8), decomposing the problem and yielding more parallelism. The task-graph (Figure 2.4:b) is hierarchical and unfolds dynamically during executing, yielding more and more parallelism. Once

the tasks have reached the leaves, further recursion is impossible and the graph folds back, synchronizing upwards.

The programmer is responsible to insert task barriers through the `omp taskwait` (lines: 10), which ensures that the program is blocked until all previously created tasks are finished.

These task-barriers are among the limitations of the traditional fork/join task-level model. There are no possibilities to insert dependencies between tasks except for heavy task-barriers. Several data-patterns such as various sliding window patterns or pipelined patters cannot easily be expressed. This limitation has been overcome by including data-flow or data-driven parallelism into the tasking-model.

Data-flow Task-Parallelism

Data-flow task-parallelism can parallelize patterns that are hard to make parallel with the traditional fork/join tasking model. Synchronization in the data-flow model is implicit – no explicit synchronization amongst tasks is required by the user. This is unlike the traditional fork/join tasking model where task-barriers are required for correct execution of the code.

The data-flow programming model relaxes the need for explicit task synchronization by requiring the programmer to provide data-usage information regarding tasks. Now a task not only contains the computation but also information about what data-region the task will use and *how* it accesses the data-region. The how-part is usually expressed in one of three ways: read, write and read/write.

Given the information about data-regions, the runtime- system can now construct a dependency-graph dynamically during execution. Instead of using heavy task-barriers, the runtime-system will insert lightweight dependencies between tasks, preventing parallel execution of tasks that access the same data. When a task finishes, it proceeds to release subsequent, dependent tasks.

Although the programmer now has to decide the data-usage of tasks, the method is worthwhile. The first advantage is that more parallelism can generally be detected. The method for data-flow task-parallelism is closely related to what OO-processors perform on the instruction level. By dynamically finding out dependencies, tasks that are spatially far from each other when invoked can still benefit from parallelism, which often can reduce the *makespan*(the maximum completion time of the tasks) [49]. This is more clearly illustrated in the Figure 2.5, where we show part of the Mergesort algorithm [50].

The program initially creates four tasks (lines: 3,6,9,12), dividing the to-be sorted array into four pieces which are sorted in parallel. When the four tasks finish, two more tasks are created to merge the four array chunks into two arrays, again performed in parallel (lines: 17,20). Finally the two arrays are merged into the final array (line: 25).

In the traditional fork/join tasking model (Figure 2.5:a), there is a need to insert a task-barrier (line: 15,23) in-between the parallel regions to avoid data-races between the phases. This works well as long as the work is uniform in computa-

(a) Example OpenMP kernel

```
1 int fib ( int n )
2 {
3   if ( n < 2) return n;
4   int x,y;
5
6  #pragma omp task shared(x)
7     y = fib ( n-1);
8  #pragma omp task shared(y)
9     y = fib ( n-2);
10  #pragma omp taskwait
11
12  return x+y;
13 }
14
15 int main ( int argc, int argv[])
16 {
17   #pragma omp parallel
18   #pragma omp single
19     fib(30);
20 }
```

(b) Dynamically unfolding task-graph

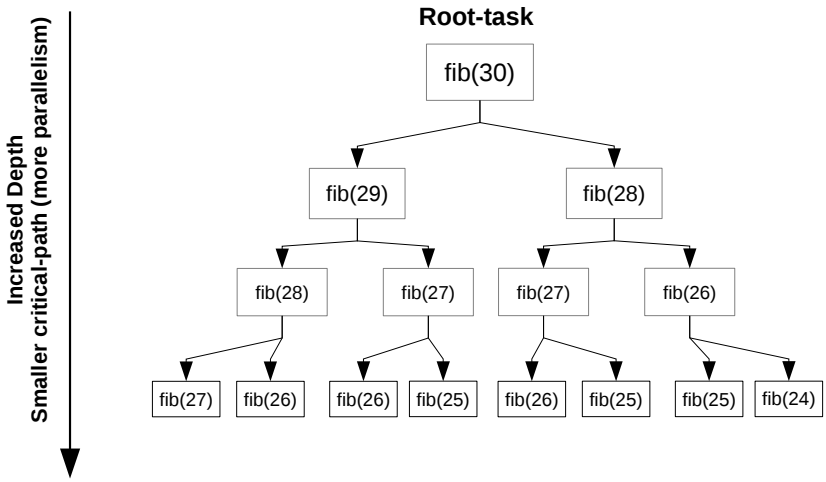


Figure 2.4: Parallelized Fibonacci kernel using OpenMP with recursive task parallelism (a), and how the task graph unfolds dynamically during execution, yielding more but finer grained parallelism (b).

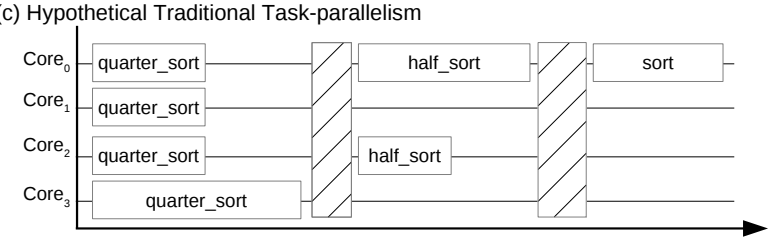
(a) Traditional Task-parallelism

```
1 int array[1024];
2 ...
3 #pragma omp task
4   quarter_sort (array);
5
6 #pragma omp task
7   quarter_sort (&array[256]);
8
9 #pragma omp task
10   quarter_sort (&array[512]);
11
12 #pragma omp task
13   quarter_sort (&array[768]);
14
15 #pragma omp taskwait
16
17 #pragma omp task
18  half_sort (&array);
19
20 #pragma omp task
21   half_sort (&array[512]);
22
23 #pragma omp taskwait
24
25 sort (array);
```

(b) Data-flow Task-parallelism

```
1 int array[1024];
2 ...
3 #pragma omp task depend (inout: array)
4   quarter_sort (array);
5
6 #pragma omp task depend (inout: array[256])
7   quarter_sort (&array[256]);
8
9 #pragma omp task depend (inout: array[512])
10   quarter_sort (&array[512]);
11
12
13
14 #pragma omp task depend (inout: array[768])
15   quarter_sort (&array[768]);
16
17 #pragma omp task depend (inout: array,array[256])
18   half_sort (&array);
19
20#pragma omp task depend(inout:array[512], array[768])
21  half_sort (&array[512]);
22
23 #pragma omp task depend (inout : array, array[256], \
24                          array[512], array[768])
25   sort (array);
```

(c) Hypothetical Traditional Task-parallelism



(d) Hypothetical data-flow execution

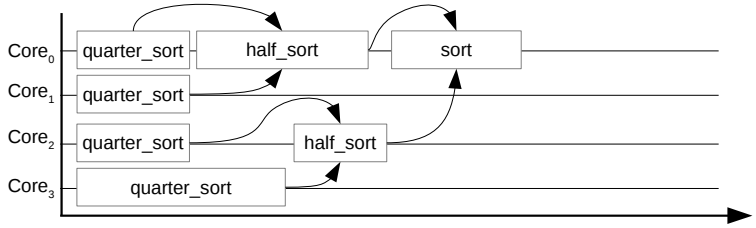


Figure 2.5: Parts of the Mergesort application constructed to expose parallelism using fork/join (a) and data-flow (b) types of parallelism. Execution examples where data-flow (d) is preferred over fork/join (c) due to immediately executing tasks whose dependencies are complete can lead to a shorter makespan

tional cost. However, should the tasks be non-uniform in execution time, there is a high probability load imbalance, as illustrated by the example schedule in Figure 2.5:c. Data-flow (Figure 2.5:b) parallelism can overcome load imbalance in this example because as soon as two of the fast tasks are finished, the merging phase can start(Figure 2.5:d), which can lead to a smaller make-span (reduced execution time).

The second advantage is that patterns that were difficult/impossible to make parallel before can now be parallelized. Wavefront [51] or similar patterns where tasks have very fine dependencies between each other cannot easily be expressed using heavy-weight full-barriers. Data-flow parallelism has been used to for example express the complex patterns in the H264 decoder standard [52], which are difficult to express using fork/join task-parallelism.

Arguably, data-flow programming can be easier to use than the fork/join model. Since parallelism is automatically deduced and detected based on data-usage in the data-flow model, the programmer needs little knowledge on how the problem is parallelized. In the fork/join model, the programmer must know what parts of the application can execute in parallel and delimit the parts from each other to ensure correct execution.

The final advantage is that the prerequisite for distributed computing comes for free. Because the runtime-system now knows what data a task will use, it can transfer that data to a distributed device such a GPU and execute the tasks there, offering opportunities for acceleration. While the OpenMP commitee decided that the direction towards acceleration for OpenMP would be by including new `target` directives [53], existing work has successfully demonstrated the use of data-flow directives in heterogeneous computation [54, 55, 56].

Today, data-flow parallelism exists in OpenMP version 4, introduced in 2014 [57]. One of the pioneering frameworks for task-based data-flow parallelism is CellSs [58], which showed how to efficiently program the notoriously complicated IBM Cell processor [59] using the data-flow tasking model. OmpSs [54], a sequel to CellSs has used the data-flow model with GPUs and Intel Xeon PHI [9] to accelerate computations. Various other data-driven model have also been proposed. Runtime-systems with similar goals using other semantics include the OpenStream [60] framework, which uses streams to express producer/consumer relationships between tasks. A more manual approach is the programmer-assisted task-to-task synchronization model proposed in OpenUH [61] is also early work that strengthen the need for data-flow task parallelism.

2.2 Internals of Task Parallelism

This section overviews how task-parallelism is implemented in the compiler and the runtime-system with particular focus on the OpenMP framework BŁYSK that has been developed throughout the thesis.

Compiler

The compiler is an important part of the OpenMP programming model as it bridges the gap between the complex runtime-system API and the programming language. Its primary purpose is to parse and understand the grammatical structure of the target language as well as the semantics of the OpenMP directives and to interface the underlying task-parallel runtime-system.

OpenMP compilers today come in two forms. There are fully-fledged compilers where the OpenMP transformation is a phase like any other inside the compiler, typical in for example GNU's C compiler, Intel's C/C++ compiler and Sun (Oracle) compilers. The other variation, often used in university research, are source-to-source or transcompilers. Source-to-source compilers only perform the OpenMP transformation part and output the transformed file in the same language that it was parsed from, after which any backend compiler can be used to generate the binary executable. Popular research compilers for OpenMP include Mercurium [36] and ROSE-based compilers [35].

We will now look at how BŁYSKCC [62], a contribution of the thesis, transforms various OpenMP directives and how they are interfaced inside the BŁYSK runtime-system. The language of choice here is C. Our input source code, which will be used throughout this section, is shown in Figure 2.6 and corresponds to the well-used recursive Fibonacci series calculation.

pragma omp parallel

The `omp parallel` directive states that a team of threads should be created. In other words, the program should boot a number of threads, where each thread will execute the following structured-block (list of statements). The translation of the `omp parallel` is performed inside the compiler through these four steps:

- Identify all references to variables with no local declaration (called *shared* in OpenMP terminology) inside the parallel region. Create a structure called `__blysk_arg_struct_2` which contains pointers that point to these external declarations. In this example, no variables with external declarations used in the parallel structured block.

- Change all references to shared variable in the structured parallel block so that they are references through the pointer in the structure.

- Extract the structured block and put it into a separate function called `__blysk_omp_parallel_f_2`. The function should take a pointer to structure (of type `__blysk_arg_struct_2`) that contain the variable scoping as an argument.

- Replace the `omp parallel` directive and the structured block with a call to the runtime-system: `BLYSK__parallel()`. We give the runtime-system a

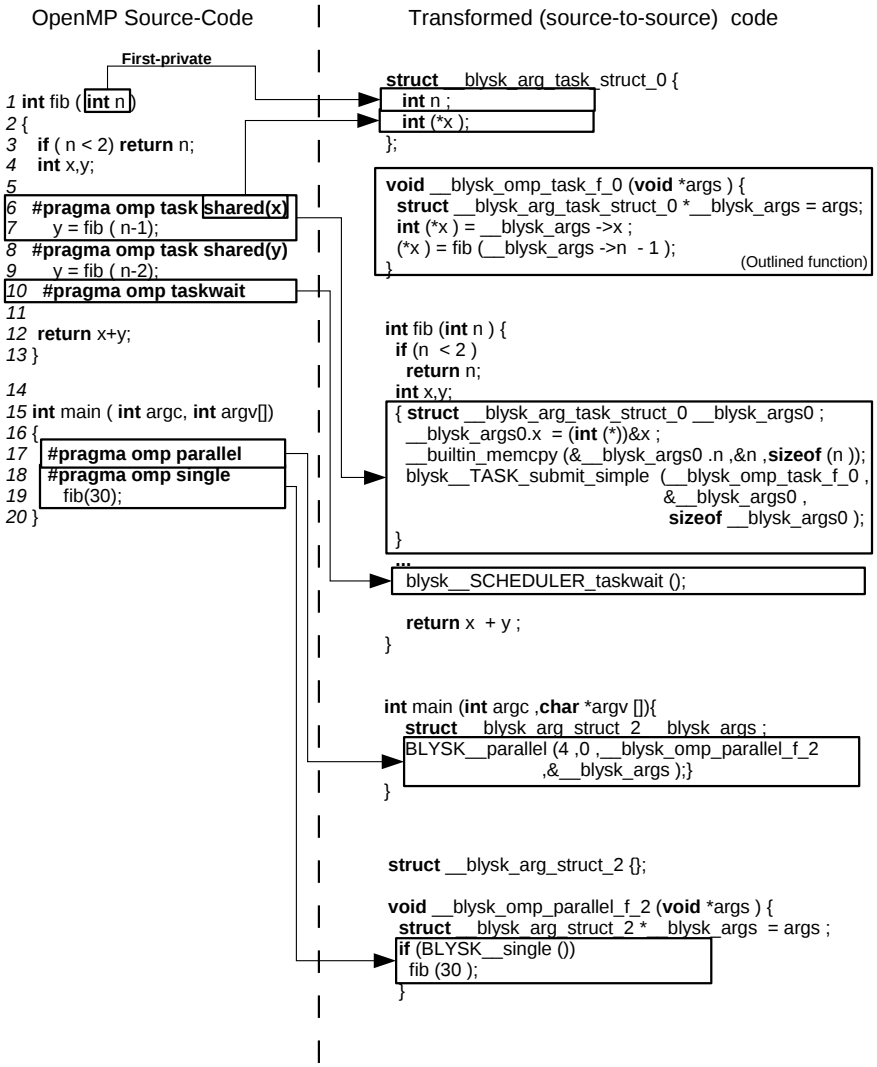


Figure 2.6: Transformation of an OpenMP extended Fibonacci program into using BŁYSK runtime-system calls.

pointer to the parallel structured-block and a pointer to the structure containing variable information. The runtime-system is responsible for booting the threads and forcing them to execute the provided function.

pragma omp single

Next in line is the single statement, which ensures that only one of the threads executing the current context is allowed to execute the following statement. The source-to-source conversion is trivial, since we simply ask the runtime-system if a thread is allowed to execute – it is up to the runtime-system to either return a true or false value and to ensure that only one thread executes the region. The directive is removed and an if-statement is instead composed with a call to the `BLYSK__single()` function inside the runtime-system.

pragma omp task

Being the workhorse of the task-parallel mode, the `omp task` directive exposes the following structured block as an asynchronous task. Similarly to the creation of a parallel region, a task-structure (`__blysk_arg_task_struct`) is created, one for each task. The structure will hold references to out-of-scope variables, unless specified private or first-private. In our case, the `x` variable have been specified as shared, forcing the task-structure to contain a pointer to the original declaration of `x`. Variables that are specified as private will be re-declared inside the task-structure. The variable `n` is by default first-private. First-private variables means that the task should have a local copy of the variable and it should also be initialized to the value of the original declaration. Thus, during task creation, a copy of the declaration for the variable `n` is added to the task-structure and initialized (through `memcpy`) to the original variable's value.

The task's structure block is extracted and placed in a separate function called `__blysk_omp_task_f_0`. All variable references that are not locally declared in the task are replaced with references to the task-structure. Finally a call to the runtime-system is made (`blysk_TASK_submit_simple()`), with the address to the task-function and the structure containing the variables.

pragma omp taskwait

Used to synchronize all previously spawned tasks, this directive provides another simple transformation. The directive is directly replaced with a call to the runtime-system's `blysk_SCHEDULER_taskwait()`.

The Runtime-system

The runtime-system is the most important part of the task-parallel programming model, orchestrating the execution. Any parallel runtime-system must provide:

- Create and manage threads
- Manage parallelism in the form of tasks
- Capable of synchronization of threads and tasks and to have the notion of task scopes
- Schedule work onto processing elements

The four points stated above are common to all task-based runtime-systems. However, a modern state of the art runtime-system will have more advance features:

- Accelerator support. Most commonly GPUs.
- Data-flow tasks. Unlike fork/join tasks, data-flow task parallelism requires more effort to perform well.
- A variety of scheduling algorithms, chosen depending on the parallel workload

An overview over the BŁYSK runtime-system, developed throughout the PhD studies, is shown in Figure 2.7: During execution, an application that uses task-parallelism will continuously call the runtime-system's API to expose asynchronous tasks (Figure 2.7:a). Each of these calls will result in a task being created inside the runtime-system (Figure 2.7:b). The task contains information about what program code it should execute – in most of the cases, the program code is given from the application in the form of a function pointer and its arguments. When the task have been internally created, it is either submitted to the dependency manager or directly to the scheduler.

The dependency manager (Figure 2.7:c) is invoked if the task uses data-flow synchronization. The dependency manager is responsible for ensuring that only dependency-free tasks, that is, those tasks whose dependency has been cleared are allowed to execute. It also houses methods for releasing tasks' dependencies and querying tasks for the dependency information, which are often useful when taking locality-based scheduler decisions (e.g. for GPUs [63]).

Tasks that are ready to be executed will be sent to the scheduler (Figure 2.7:d). The scheduler decides where and when a task is placed onto the available hardware resources. The scheduler also ensures correctness of barriers and task-synchronization points, and maintain communication with the threads that are executed on the cores.

Inspired by the OmpSs runtime-system Nanos++ [54], the BŁYSK scheduler uses scheduler-plugins (Figure 2.7:e) to choose from a variety of scheduler algorithms. Unlike for example GNU OpenMP or Intel's OpenMP implementation, which both only supports one type of scheduler, the user can test a variety of scheduling algorithms without recompiling the application.

The thread-manager (Figure 2.7:f) is a collection of user-level threads abstracting the underneath hardware (Figure 2.7:g). The abstraction is most commonly implemented using POSIX Threads, where each thread is a proxy for a core. In

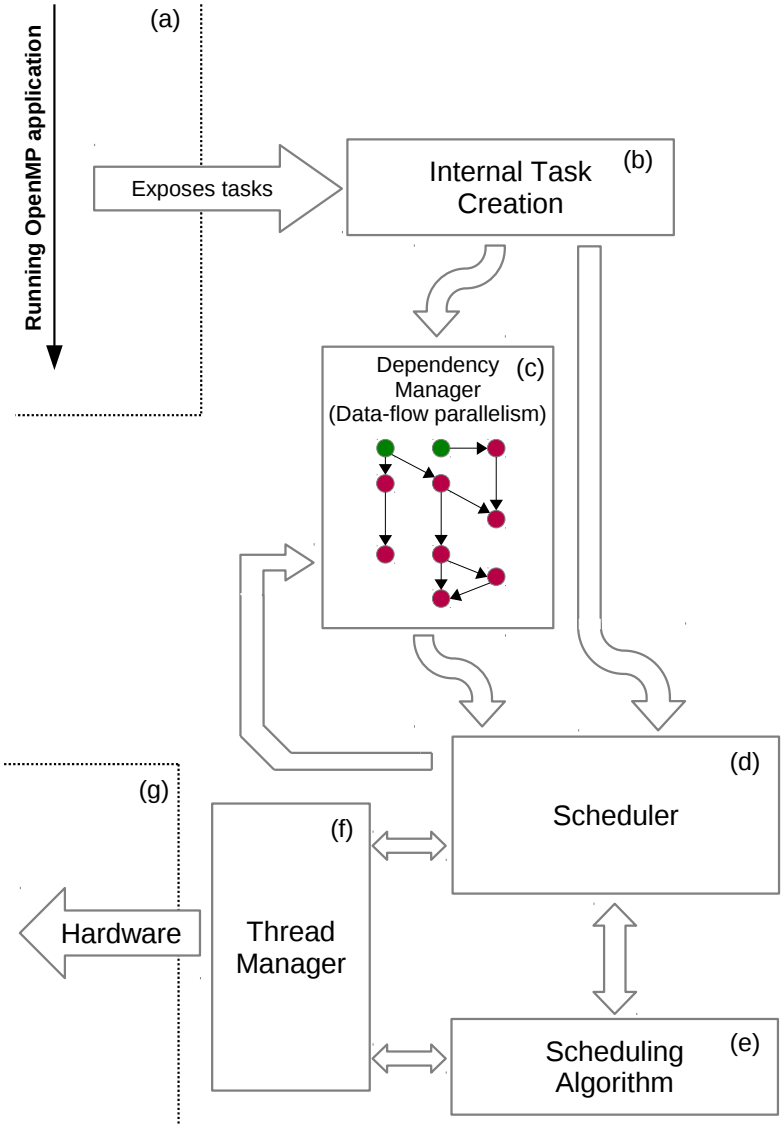


Figure 2.7: Overview over the components included in the BŁYSK task-parallel runtime-system.

BLYSK, each thread-context usually contain a task-queue on which the scheduler enqueues work into. Threads will continuously consult the scheduling policy concerning what they should be doing when they finish executing a task.

2.3 Task-parallel Challenges

While the task-model has matured throughout the years since its inception, there are still many challenges.

Overheads

It is generally agreed upon that the trend for homogeneous processors will continue to include more and more cores. More cores should yield a faster execution speed, which is often proportional to the number of cores used when the entire application is made parallel. Often, more cores also means decomposing the application into using finer grained parallelism.

Maintaining tasks in a runtime-system comes with a cost – the task must be created, dependencies analyzed, scheduled and executed. This overhead cost is not free. When the overhead cost consumes a large fraction of the overall work for a task, performance gains are reduced – sometimes performing slower than its serial counterpart.

There are many engineering efforts that reduces overheads in a runtime-system, allowing for better performance of fine-grained tasks. One such runtime-system is Wool [40], which strives to minimize overheads through cache-aligned task-structures, an efficient work-stealing scheduling policy and fast queue implementations.

Lock-free structures are another broad research field where performance improvements can be obtained [64]. Data-structures shared amongst multiple threads are often protected by locks. Locks are often expensive and can lead to severe degradation in performance [65]. A better way is to remove the locks and implement those using atomic instructions. Atomic instructions perform a certain computation on a memory location atomically – it appears as if the sequence of instructions were performed atomically. Atomic instructions are fast compared to general spin-locks, and can dramatically increase performance because they are performed in the memory hierarchy (rather than in the processor).

Another method is to use transactional memory [66] (TM) in software or hardware instead of locks. Where atomic instructions can only handle small calculations, TM can be used as a direct replacement for normal locks. TM is usually implemented through the cache consistency mechanism, where regions of protected code called transactions are repeated if any memory hazard occur.

The problem of overheads increase when more complex task-parallel functions are used. Data-flow parallelism experience much larger overheads because now each task must pass through the dependency manager to solve dependencies.

In his paper that revisits Amdahl's law, John Gustavsson wrote [67]: We should be "... scaling the problem to the number of processors, not by fixing problem size". His intent was to show that one cannot achieve the level of parallelism required for Amdahl's law to scale to a thousand cores. Interestingly, his conclusion can reduce the problem of large overheads. Runtime-system overheads are relative – they only pose a problem if the work within the tasks are small in comparison. By scaling up the problem size, the ratio of overhead to work decreases, and the problem is likely to scale. Unfortunately, not all problems can be scaled up (nor do we want them to). For example, if the intention is to solve a matrix multiplication with particular dimension, then there is no way to scale the problem up. Instead, we want the runtime-system to be able to handle as much parallelism that we can potential exploit in our application, with no degradation in expected performance.

Objective-based scheduling

The most common objective that task-parallelism is used for is performance – we want to execute the application as fast as possible given a certain multiprocessor system. But there are many other objectives that can be desirable to strive for within the task-parallel model.

One such objective is the need to control temperature. Temperature today governs the performance of the processor. Processors today employ a variety of techniques to cool the processor down. For example, when an Intel processor becomes overheated, the hardware automatically decreases the operating frequency to decrease power consumption [68]. This method, employed in many designs today, is called dynamic-voltage-and-frequency-scaling (DVFS) and is frequently used today. Another example where temperature is the limiting factor is space applications, where the ability to conduct heat away is limited.

Reason and intuition tells us that scheduling can make an impact on the overall temperature of the processor. How can we schedule tasks onto available cores such that the performance can be improved under a thermally constrained environment?

Another objective that does not exist in mainstream task-parallel models is attaining some form of real-time guarantees. Unlike traditional real-time scheduling, where the work-load is known a-priori and a schedule can be developed offline to provide hard real-time services, the task-parallel model has dynamically unfolding parallelism during runtime. The main challenge here concerns enabling the semantics for expressing real-time execution in the task-parallel model and how to use the new timing information to make good scheduling decisions.

Heterogeneity

The trend of processors moves towards more cores and more heterogeneity. For example, the cores in the processor may be different (but with same ISA) such as the ARM bigLITTLE [69] cores. They can also be of different type, such as mixing GPUs with CPUs on the same die or hybrid FPGA/CPU approaches.

Independent on how the future will look, it will include special-purpose accelerators for speeding up some parallel patterns. The programmer now faces the challenges of using these accelerators.

The task-parallel model works well with heterogeneity. Often the programmer is asked to provide different versions for the same task for the different hardware, letting the runtime-system decide which task to invoke. The main research question is how to where the tasks should be scheduled? Incorrectly scheduling work on a CPU while a GPU is inactive may lead to substantial loss in performance.

Using FPGAs can offer improvements in both performance and power consumption over GPUs or CPUs[70, 71, 72]. Seeing how future systems will contain a mixture of ASICs and FPGAs, such as Xilinx ZYNQ and the Intel HARP program, there is a need to tightly incorporate the functionality for tailoring FPGAs to fit the needs of the task-parallel application. The main challenge is how to bridge and break the typical hardware synthesis flow so that it becomes available for the any software programmer. One direction explored in the thesis is to automatically create hardware accelerators based on the task-parallel source code. The main challenge here is how the hardware accelerators work: we want to have as parallel hardware as possible.

Chapter 3

Improving Task-Parallel Performance and Quality-of-Service

The following chapter summarizes and discusses the peer-reviewed contributions of the thesis.

3.1 Performance of Task-Parallel Programming Models and Libraries (Paper I)

The task-parallel model has gained momentum and popularity, and today there are several different commercial and research runtime-systems. How do these different runtime-system differ? What are the key insights to performance and why do the performance vary across them? What impact does a centralized scheduling policy have on the performance? Does a work-stealer thrash the cache?

This work quantitatively compares several different runtime-system implementation in terms of power and performance, including a thorough study of the overheads related with them.

The study focuses on task-parallelism, particularly but not exclusively viewed through the eyes of OpenMP. Most previous work concerned thread-parallel performance of OpenMP and related programming models [73, 74]. In particular, existing work focuses on synchronization, thread-barrier and critical section overheads. Since the inception of the OpenMP task-parallel model, several authors have created new runtime-systems adhering to the OpenMP standard. Most (if not all) of the new runtime-system implementation feature a performance evaluation with respect to existing implementations [75, 76, 77, 60]. However, our study provides a unique insights gained due to the independent nature of the evaluation and includes more evaluated runtime systems than previous work. Our work is

also among the few to evaluate what impact various runtime-systems have on the power- and energy consumption.

Method

The Barcelona OpenMP Task Suite(BOTS) [50] is a benchmark suite targeting OpenMP tasking performance. The BOTS benchmark suite was ported by us to Intel Cilk Plus [78], Intel Threading Building Block (TBB) [39] and Wool [40].

The evaluated OpenMP implementations were Intel's runtime-system, GNU's OpenMP (GOMP), Nanos++ [54] (today: OmpSs), Sun's (today Oracle) OpenMP [79] and OpenUH [34].

The TBB and Nanos++ library was also ported to the Tilera TILEPro64 processor.

The evaluation took place on an AMD Opteron 48-core system (AMD Opteron 6172) and a TILEPro64 processor. Additionally, the power consumption was measured for the TILEPro64 using by measuring the voltage crossing the resistor on the voltage-supply pin. The power consumption setup was identical to the work by Själander et al. [80].

The performance was measured and compared by evaluating the speed-up of the different runtime-systems. The speed-up is the parallel execution time improvement over the serial execution time:

$$Speedup = \frac{t_{serial}}{t_{parallel}}$$

We only considered the parallel region when measuring performance – serial work such as initialization, memory allocation, etc. prior to the parallel region was excluded.

The cache performance of each benchmark was measured using PAPI [81] calculated using the weighted-average (task execution times are the weights) over all tasks and their execution time. Load-balancing was calculated by averaging about how many threads were simultaneously running tasks during the execution. Task creation and synchronization overhead was measured using a microbenchmark created for the purpose.

Conclusion and Discussion

We found that overheads had a leading role in terms of performance for the parallel applications. While most runtime-systems managed coarse-grained parallelism fairly well, most of them failed handling finer grained parallelism. Load-balancing correlated with granularity and using finer grained tasks often leads to better load-balancing property assuming that the runtime-system could manage the overheads. Among the runtime-systems that were unable to manage fine-grained parallelism were Nanos++ and GNU's OpenMP implementation – both had magnitudes larger overhead times compared too for example Wool. These high overheads materialized

in slower execution time on the benchmarks, particularly when exposing more (and finer) parallelism.

Overall, we found that work-stealing was preferable for benchmarks that utilized recursive parallelism. Work-stealing on recursive applications helped to balance the load more evenly across cores and to reduce negative cache-effects. The caching performance for work-stealers were better than that of centralized approaches. This was most evident when comparing the Nanos++ work-stealer against their centralized approach– the same framework with identical overheads yet vastly different cache performance.

Performance on the TILEPro64 was similar to that of the AMD Opteron server. On the power-consumption side, we found that faster is usually better. Wool, a framework that focuses only on execution time without explicitly putting threads to sleep, had the highest power-consumption while consuming the least amount of energy for the benchmarks. GCC's OpenMP had the lowest power-consumption, partially due to the explicit thread-sleeping mechanisms (Fast Userspace Mutexes), but also had the highest amount of energy consumption due to the increase in execution time.

3.2 Quality of Service in Task-Parallel Runtime-systems (Paper II and III)

Temperature is an ever present constraint in today computer architectures. The power consumption scales linearly with the frequency and with quadratic growth on the voltage. Because increasing the clock frequency has historically been one of the driving forces behind performance while at the same time decreasing transistor sizes, the power-*densities* (the temperature) have continued to grow – more transistors one a continuously shrinking area increases the temperature.

While there are many autonomous systems such as frequency-scaling [82] or clock-gating [83] that reduce power consumption in modern processors, we wanted to look at the runtime-system side of the matter, in particular task scheduling. Intuitively, a scheduling algorithm that is obliviously greedy will not be consistently thermally most efficient. Because of the greedy property, a core will always execute a task if there is one to be found, independent of its current temperature.

Scheduling algorithm for minimizing thermal effects exist, but most of these reside in the operating system. One way is to let a thread jump between cores– the thread will start on one core, heat that core up, and migrate to a cooler core [84]. Another approach that works for applications that execute for a long while is to predict the temperature, either by estimating the time before a certain core will reach a certain temperature or by predicting the thermal impact of the application on the underlying core [85, 86]. While these approaches work in a thread-centric view, they may be inadequate in a task-parallel model, where work is fine-grained enough to occupy all cores and can be very dynamic.

Paper II concerns scheduling tasks with their power-consuming properties in mind onto processors core to reduce negative thermal-related effects. More specifically, we developed a scheduling algorithm that strives to keep core-temperature below a certain thermal throttling threshold.

Paper III takes a different approach in which there is a fixed objective – a soft-real time constrain – placed on tasks. The question now becomes: how do we schedule tasks onto the available resources such that we minimize the temperature while meeting the soft real-time constraints?

Thermal-aware OpenMP design (Paper II)

Method

Our approach was to model the TILEPro64 processor. The TILEPro64 is a VLIW MIPS-type processor with 64 cores but we limited ourselves to 16 cores. A simplified HotSpot [87] thermal model was created where we modeled each core in the system independently with no thermal leakage between them. A McPAT [88] model was developed for the power-consuming properties of the TILEPro64 and a number of benchmarks from the BOTS [50] suite were executed to estimate their power-consumption; a strong correlation between an applications Instruction per Cycle (IPC) and the power-consumption was found, which was used to drive the thermal model. The thermal model was continously running on the platform, modeling the temperature of each core and stalling program execution if the processor's throttling temperature was reached.

We used Nanos++ [54] to develop our temperature aware scheduler. Our scheduler is called TTI (Thermal-Task-Interleaving) and probes tasks for their power-consuming properties (their IPC). When a task with known power-consuming properties is created, it is sorted into either a low-, medium- or high-power queue. Cores then greedily pull the low-power queues when they are near the thermal throttling threshold to cool-off (while still performing useful work) or the high-power queues when they are already cool.

Evaluation was performed by mixing applications from the BOTS benchmark suite. Comparison between the TTI scheduler and the existing Nanos++ default and breadth-first schedulers were performed by evaluating the speed-up.

Results

The TTI scheduler could improve the execution time performance when compared to the existing task-schedulers by as much as 27%. The impact of the TTI scheduler is particularly visible when there is a large variety of high- and low-power tasks that can be used to heat-up or cool-down the cores. The performance when there is only high-power tasks was low, and there were was no significant improvement in scenarios where only low-power tasks were present.

Soft-real time OpenMP extensions (Paper III)

Method

We used the Nanos++ [54] framework as a starting point. The Mercurium [36] compiler was extended to support three new task clauses:

- `deadline(t)` specifies the latest time a task can start
- `release_after(t)` specifies the earliest time a task can start
- `on_error(OMP_DROP)` specifies if the task can be dropped by the runtime-system. A task that can be dropped should not have any impact on the correctness of the application.

We also allowed the programmer to set a global constraint on the real-time requirements: the number of deadlines that were allowed to be missed (in percentage). In a real-life scenario, the operating system would control the global real-time constraint across different running application in order to maximize the experience perceived by the user.

We implemented a PID [89] controller that continuously monitors the amount of deadlines missed and adjust the amount of time a core is active (or put to sleep). For example, if the user allows 2% of the tasks to be missed, and the controller detects that currently 1% of the deadlines are missed, then the cores will be proportionally decreased. The opposite will happen if the controller detects that too many deadlines are missed. The sampling rate for the amount of deadlines missed was 100 milliseconds.

The scheduler implementation consisted of a global earliest-deadline first (EDF) queue [90] from which the threads pull tasks from.

We evaluated our scheduler with two benchmarks: a raycaster written in the classical fork/join task-model and a H.264 decoder [52] written in data-flow style. A 256^3 sized voxel image and a movie was used for the evaluation, both using frames-per-second (FPS) as a metric for performance.

The temperature was measured using the CoreTemp kernel module on a Intel Nehalem i7 processor. The running temperature of our algorithm was compared against a temperature and real-time unaware scheduler – the commonly used work-stealer.

Results

Our scheduler showed little deviation in terms of providing the real-time service that was expected. It either provided the expected real-time service or slightly over/under-shot the expectation. The work-stealer always over-provided the expected real-time service, which led to a hotter chip. Our scheduler could lower the average temperature of processor cores by up-to 17.7% degrees Celsius compared to a work-stealing algorithm while keeping comparable real-time performance.

Discussion

Paper II showed how to regulate the temperature of processor cores through identification and scheduling of tasks' potential power consumption and balancing the load accordingly. Our method holds promise where temperature cannot easily be led away, such as space-oriented application. One may wonder why thermal regulation should reside inside the user-level OpenMP runtime-system and not use existing OS thread-centric thermal schemes [85, 91, 92]. One reason is that future systems might not have a fully-fledged operating system running on each core but instead be partitioned into clusters of cores running a thin OpenMP (or similar) runtime-system layer on top. A second reason is that the Operating System and the task-parallel runtime-system does generally not synergize very well; the runtime-system's view of the architecture is that the threads are proxies for cores – disturbing the runtime's viewpoint by letting the Operating System migrate threads or allowing them to time-share the same core will upset the task-parallel runtime-system view of the system.

OpenMP has historically been used for high-performance computing. Paper III showed another side of OpenMP – it can be used to describe and achieve soft real-time behavior in general purpose systems. The work conducted is novel, and makes no distinction between fork/join and data-flow (previous work only considered fork/join [93]). While OpenMP is ill suited for hard/firm real-time, we showed the benefits it can give in a soft-real time, everyday scenario – such as playing a movie on the desktop computer.

Limitations to the study include the feedback control (configured empirically) and the scheduling implementation, which uses a global queue; global queues are known to cause lock-contention. Future work would benefit from the reduced overheads in a distributed (work-stealing) EDF schedule policy. Nonetheless, the contribution is among the few to attempt to include real-time properties into the OpenMP programming model.

An important limitation to both papers is that they only work well when running in isolation – something that is currently true for most user-level task-parallel application. Two OpenMP applications running simultaneously on a system will have no awareness of each other, often disturbing each other's execution due to oversubscription of threads, context switching or cache thrashing. Both contribution suffer this limitation, although the work in Paper II is slightly more resilient due to the dynamic feedback controller. While addressing this limitation is out of the papers' scope, it is actively being worked on in the community, although usually to improve execution performance [94].

3.3 Improving Performance of Task-Parallel Runtime-systems (Paper IV-VII

Task-parallel execution time performance can be increased in several ways. One is to take advantage of new and exotic hardware, such as GPUs or other acceleration

devices. One of the research challenges is the change in programming language to allow these accelerators to work within the task-parallel framework in a user-friendly way. A common solution is to extend the task-parallel framework to include syntax to specify multiple implementations of the very same task [54, 95, 56], for example a CUDA [96] and an x86 version. Another alternative is to use tools to automatically translate the OpenMP directives into runnable accelerator code [97, 98].

A method for improving performance is improving task-scheduling. Because the performance of accelerators differ vastly from general purpose processors [99], scheduling decisions can drastically affect the makespan of an application. One method is to balance the work according to estimate kernel finish execution time for all processing elements in the system [100]. This approach, called HEFT (Heterogeneous Earlier Finish Time), is similar to our contribution in Paper II but with timing rather than power, and has been implemented in StarPU [95]. It has been shown to work well for multiple-GPU systems. Another approach is to dynamically detect the critical path [101] of the executing task-graph and schedule tasks that are on the critical path on faster accelerators; because the critical path dictates the makespan of the application, reducing the critical path also reduce the execution time of the application.

Another way to improve performance is to tackle overheads in the runtime-system. Paper I revealed that overheads limit the performance of task-parallel applications, particularly with fine-grained tasks. Tasks of fine-granularity are often required to enable smaller workloads to scale. While overheads in fork/join task-parallel runtime-system can be large, they can still be dealt with by various cutoff mechanisms [50, 47]. Cutoff mechanisms does not work as well with data-flow tasks, as there is no way to inline tasks who are not yet ready to execute. Here, every task must go through the effort of analyzing its dependencies with other tasks before taking action. The time taken for the dependency analyzing step is crucial, and can vary much between implementations [102]. Paper V presents a way to reduce the cost of analyzing dependencies through reuse and a fast runtime-system implementation.

Our work ends with improving performance by transcending the architectural parallelism offered by today's off-the-shelf processors and instead automatically generate the hardware. By creating hardware absent of logic unused in the task-parallel application, our system could contain a higher degree of architecture-level parallelism compared to commodity processors.

GPUs and distributed devices (Paper IV)

Paper IV introduces UnMP, the precursor to the BŁYSK framework. UnMP is a task-parallel runtime-system aiming to utilize distributed memory accelerators such as the GPUs and TILEPro64 devices dynamically. The programmer is responsible for providing three different kernel implementations (similar to OmpSs [54]) for the same task: an x86 host version, a GPU CUDA [96] version and a TILEPro64

version. The UnMP runtime-system can, when a task is submitted for execution, decide on which device the task will be executed.

Because TILEPro64 never was intended for host-initiated execution, a small loader-kernel was developed. The kernel is booted on the TILEPro64 upon application execution and handles communication between the board and the host system through PCI. Executing a kernel on the TILEPro64 involves sending the position-independent kernel and the associated data to the TILEPro64 together with a pointer (on the device side) to the arguments. The load-kernel will invoke the task in a SPMD-fashion on the device, enabling all 60 cores to execute the kernel.

A number of known homogeneous scheduling algorithms were evaluated with the three-device setup (x86,GPU, TILEPro64) on the device: a random work-stealer, a random work-dealer and a weighted-random work-dealer [95]. The weighted-random work-dealer used a-priori calculated weights; the weights for each devices were obtained by executing the task-kernels in isolation on each device.

We developed a new scheduling algorithm, called FCRM, that complements the weighted-random scheduling algorithm by constructing dynamically adjusting the weights. The weight can change at runtime because of several reasons. For example, we utilize a Least Recently Used (LRU) soft-cache [56, 103] to minimize transfer overheads between the host processor and the accelerators. Depending on where data is located, certain device should have a higher probability of received a task. Our weights are calculated through segmented linear regression.The data for the linear regression is obtained by probing task execution times on the different devices. The function regresses on any user-defined variable, which can be for example the data-size of the kernel.

Results

Our FCRM scheduler performed much faster than the alternatives. While being initially slower than the weighted-random work-dealer, the FCRM method of scheduling gradually improves as the regressed function builds up. By avoiding scheduling tasks on slower devices after probing them, the scheduler schedules tasks onto devices that execute the kernel quickly.

Runtime-system design for data-flow task parallelism (PAPER V)

One of the main conclusions in Paper I was that overheads can be a substantial bottleneck in fork/join type of task-parallelism, limiting the speed-up that can be attained. With the introduction of data-flow parallelism in OpenMP, the matter of overheads became even more severe. Initial experiments that we performed showed that granularities that worked with fork/join parallelism experienced worse than sequential performance in the data-flow model, which motivated us to work on reducing data-flow task overheads.

Unlike existing work, which use the data-flow model to improve scheduling decisions or use accelerators (e.g. GPUs) to gain performance, our goal to reduce the overheads in solving dependencies between tasks. It was towards this aim that the BŁYSK infrastructure was developed, using knowledge gained from Paper I and IV. The BŁYSK infrastructure was made with fast dependency resolution in mind.

We observed that many data-flow patterns were dynamic and not dependent on the content that the data used for the dependencies had. Our hypothesis was that performance could be gained by preserving the task dependency-patterns across applications runs. Our hypothesis was driven by the fact that many data-flow kernels' dependency structure remain unchanged and are static, such as for example Matrix Multiplication and various wave-front applications.

Method

We created BŁYSKCC, a compiler capable of transforming OpenMP version 4.0 directives into API calls into the BŁYSK runtime-system. A new clause was added: `dep_pattern(*name*)`, which allowed the programmer to record the dependency-graph of the subsequent task-group and name it. The dependency-graph could then be optimized to remove unnecessary dependencies such as Read-after-Read (RAR) dependencies. Once the dependency-graph had been recorded, it could be reapplied on the next invocation of that same group of tasks to reduce dependency-analysis overhead. Several improvements were done inside the runtime-system. A fast, multilocked bitwise trie was created to reduce the overheads of solving dependencies. The Hypergraph scheme [102] independently created by us to quickly solve dependencies.

Performance of the proposed methods were evaluated using standard benchmarks and compared to the GCC OpenMP 4 version as well as OmpSs (known for its data-flow task implementation). The evaluation particularly focused on tasks with fine-granularity, as these were the problem areas detected with state of the art data-flow runtime-systems.

Results

The evaluated performance of BŁYSK surpassed that of OmpSs and GCCs OpenMP library. Performance could reach twice that of OmpSs or GCC, solely because we could handle task granularities that other runtime-systems could not. Our approach further leverage performance by reusing patterns, which allowed data-flow task granularities to be on par with the original fork/join model. Our method also work with heterogeneity because the dynamic information concerning the each task's data region is preserved.

Hardware acceleration of OpenMP task parallelism (Paper VI and VII)

Processors have continously grown faster and better, integrating more and more cores, enabling better performance. However, often the sheer amount of parallelism that exists in parallel applications (in particular benchmarks) does not match the parallelism offered by today's general purpose systems. An abundance of parallelism (especially if it is fine-grained) can lead to performance degradation as shown in Paper I.

Rather than using commodity hardware, why not generate the cores and tune them such that they correspond to what is needed for the computation? The generated processing elements could then be simulated using Field Programmable Gate Arrays (FPGAs) and include a large number of cores.

Commercially available processors contain many redundant and unused units such as translation-lookaside buffers, floating point units, large register files etc – while all the aforementioned units are necessary for an operating system to work properly, they are not necessary needed for the actual task-parallel application to work. An automatically generated core would only contain the logic needed for the computation, thus enabling a large amount of these small cores to be present in an FPGA.

Paper VI and VII focuses on High-Level Synthesis of OpenMP application into custom System-on-Chips (SoC) with an order of magnitude more processing elements than what current shared-memory systems allow.

High-Level Synthesis have been well-studied before. One direction exploited is to improve sequential performance of soft-cores (processors suitable for FPGAs) by adding custom-build circuitry that perform parts of the application [104, 105]. Another approach is to synthesize the equivalence to threads in hardware [106, 107], and use the obsolete thread-parallel model to exploit parallelism. A third popular way is to use the SIMT paradigm to auto-generated synthesizable hardware [108, 109].

Our work is exclusive to the task-parallel model, enabling all three of Flynn's [14] parallel architecture when generating the hardware. The proposed methodology in Paper VI and VII includes SISD, SIMD, MIMD and SPMD.

Method

The BŁYSK compiler (Paper V) was extended to support an intermediate format (IR) in standard form. A compiler backend was created to transform the IR into actual hardware in the VHDL language. OpenMP task primitives are converted into hardware units called *hyper-tasks*. Each hyper-task is very application-specific and only perform the computation within the task. All other OpenMP directives are source-to-source transformed to exploit the hyper-tasks.

The cores that are generated work with Altera's tool-flow. They use Altera's components such as the floating-point units [110] and sit on the Avalon [111] bus.

The system is very similar to the IBM Cell [59] architecture, where there is a master processor and a number of slaves. The master processor used was the Altera NiosII soft-core [112] and the slaves are the hardware materialization of OpenMP task-constructs. The master runs the BŁYSK runtime-system and orchestrates the execution, scheduling work onto the slaves.

We call a cluster of hyper-tasks that shares a memory interface controller as an *accelerator*. Inside an accelerator, individual components can be shared between hyper-tasks. Sharing components is done through arbiters that are automatically inserted as decision makers between requesting hyper-tasks. SIMD units (vector units) and more complex components such as dot- and cross-product calculations are supported. Chaining [113] is supported, where the output of components can be directly connected to the inputs of other components (rather than being saved in intermediate registers).

The execution is governed by a finite-state machine (FSM), composed during the compilation of hardware tasks. Decisions regarding which components are shared and which are private was solved through a constraint satisfactory problem (CSP). The CSP was implemented in Gecode [114]. The CSP takes information from both the IR code (e.g. instruction dependencies) and the hardware backend (e.g. latencies and area occupancy). The model calculates the schedule (the states that instructions should map into the FSM), component sharing and commutative properties of instructions. The goal is to maximize the area-performance, estimated through the number of instructions per cycle divided by the area of the hardware.

The Altera Stratix V [115] was used for evaluation, comparing the generated systems to existing soft- and hard-cores such as the NiosII [116], AMD Opteron 6172 and Xeon PHI systems. We compared against the Intel OpenMP and GNU's OpenMP task-parallel implementations.

Results

The systems generated through the BŁYSK compiler was comparable in performance of both the AMD Opteron and the Xeon PHI processor, while being several order of magnitudes faster than the NiosII soft-core. I found that the property limiting the performance the most was the DSP-blocks– hard-blocks in the FPGA that are used for intensive computations (e.g. Multipliers). Future generations of FPGA will likely contain more DSP logic, enabling our methodology to further improve performance.

Discussion

Exploring heterogeneity for performance purpose constituted majority of the work in papers IV-VII. Both existing off-the-shelf heterogeneous systems and application-specific generated hardware were studied. A continuation would tie all these systems together: targeting GPUs for highly-parallel branch-less kernels, using general-purpose host processors for orchestrating the execution and auto-generating custom

accelerators for the gray-area between the two. A particularly interesting direction for future research would be to investigate energy efficiency under power- and/or real-time constraints.

Another use for the HLS direction used in papers VI and VII is for evaluating scheduling mechanisms or benchmarks. Simulation using existing tools such as GEM5 [117] or Simics [118] is very slow. Using our approach for simulation purposes would not slow simulation speed down (compared to a software simulator) and be able to quickly access the performance of a processor. This would require compiler changes to support setting constraints on for example the maximum number of instructions in flight (the throughput) such that it can mimic the behavior of the intended simulated system.

Another direction is to use microcode [119] rather than our hardcoded FSM to allow more freedom. While microcoded logic on FPGAs can consume more area [120], the benefit of reconfigurability without re-synthesis could be desirable. A microcoded solution would most probably loose performance-wise.

Runtime-systems' inability to cope with fine-grained tasks was identified in Paper I as one of the limiting factors of future application scaling with the task-parallel model. This inability, which we show is amplified with data-flow tasks, was the target of Paper V. For patterns whose dependency graph is static, the performance is on par with traditionally fork/join tasks, removing the extra overheads associated with solving dependencies. Our approach would also work to improve the programmer-friendly (but slow) region-based data-flow implementation [121].

Continuing this approach would involve adding scheduling decisions or information to the static dependency graphs, based on on- or offline knowledge, which would improve cache behavior, load-balancing or adding firmer real-time constraints.

Chapter 4

Conclusions and Future work

We have studied the quality-of-service and performance aspects of the task-parallel programming model.

In the first study, performance in modern runtime-system was scrutinized. Through the use of microbenchmarks and profiling libraries, we quantified the performance and identified the major bottlenecks in the runtime-systems. The bottlenecks turned out to be runtime-system overheads and the scheduling policies. Incorrect scheduling decisions led to reduced cache performance and increased load imbalance. We also found that power-usage was much higher in the well-performing runtime-systems but led to a lower total energy consumption.

Expanding the study would include scrutinizing performance when parallelism is low. Our work primarily focused on scalability when parallelism was abundant. Unfortunately, real-life applications do not have the amount of parallelism that is exposed by the benchmarks we used. A complementary study would thus evaluate the performance where limited amount of parallelism is exposed. Such a study would identify decisions taken by the runtime-system with respect to architectural details.

Another direction of the study would be to evaluate runtime-systems that focus on heterogeneity, in particular the use of GPUs or Xeon PHI accelerators to improve performance. There is a need for such a study given the heterogeneous trend of today.

We then studied temperature for quality-of-service reasons. The motivation for the study was that current runtime-systems ignore thermal effects in the architecture. We developed two strategies for dealing with temperature.

The first was to balance the power-consumption of processing cores by estimate the power-consuming properties of various tasks. Our method successfully allowed cores to regulate their temperature based on the task they execute. Continuing this study would integrate a more complex thermal model, possibly taking future scenarios such as the challenging 3D stacked die into considerations. From the software perspective, one could allow the programmer to provide several implementations

for the same task, and reformulate our problem statement: is it beneficial to use a less power-hungry implementation to cool cores or do we allow the processor to throttle performance?

The second strategy was to reduce the overall temperature of the processors by providing just-enough performance needed to meet soft real-time objectives. Our method was to use a feedback mechanism that takes samples the expected performance of the application and regulates the number of processor active in order to reduce power-consumption. The study holds promise, as it is among the few to allow some form of real-time service to the programmer – a majority of the task-parallel literature focus solely on performance aspects, particularly when running in isolation. Continuing this study would move the responsibility for setting the expected real-time performance from the programming into the operating system. A processor manager (often called CPU brokers) would be implemented to handle communication from several running OpenMP runtime-systems and negotiate the number of processors each application would need in order to gracefully degrade the real-time performance. For example, OpenMP applications running in the background would be allowed to miss more deadlines and applications running in the foreground would have stricter needs.

We finally studied performance. While there are many aspects that limit task-parallel performance, we chose to study runtime-system overheads, heterogeneity and high-level synthesis.

We created the BŁYSK framework because we were unsatisfied by how current runtime-systems handle data-dependent tasks. We found that the overheads in current runtime-systems were far to large to exploit the parallelism in the application – more often than not, the application would scale worse than the serial version when the parallelism theoretically allowed far more.

Our method for handling the overheads was to engineer a fast, multilocked approach to data dependency. We also complemented the engineering effort by allowing task-graphs to be sampled. The sampled task-graphs were then optimized offline, removing unnecessary dependencies and improving how the dependency structure map to the various locks inside our runtime-system to further reduce overhead. The sampled task-graph could then be re-applied the next time the pattern was encountering in the application, removing the need to dynamically resolve the dependencies.

There are many possible future strategies for out method. One possible scenario would be to extend the sampled task-graphs with (worse-case) executions times and use it together with our real-time scheduler. Such an study would strive to allow more firm real-time guarantees. Another similar direction would be to partially compute the schedule of the task-graph offline given the architectural information. These hints would help the runtime-system scheduler in making better decisions with regards to for example memory hierarchy performance. A final direction would be to improve the compression of the task-graph. One method would be to insert control-flow information into the task-graph to allow the runtime-system to jump between different points into the task-graph, reducing size of the task-graph.

Heterogeneity as it is most-commonly deployed today was studied. We overcame limitations in historically homogeneous scheduling algorithms by introducing a method for dynamically probing and predicting the performance of a particular task on a device. We used the predicted information as weights to randomly schedule the tasks on the device that is likely to perform the computation the fastest, taking memory transfers into account. The objective for the study was performance. The natural continuation of this study would be to schedule for power-consumption. Because the different processors perform varying on the different tasks they are likely to have different power characteristics. Intuitively, the fastest device for a particular kernel will also consume the least energy, but this may very well not hold when the power budget itself is limited.

The final study also concerned heterogeneity, albeit now the heterogeneous hardware was created automatically using the application as a template rather than relying on off-the-shelf hardware. We continued with the BŁYSK framework and extended its compiler with the ability to generate a custom system-on-chip. The systems that were generated only contained the logic required for the tasks to operate, which allowed us to create systems with a large amount of parallel potential.

There are several different paths to continue this study. For example, to overcome the bottlenecks with regards to the management of data-dependent tasks (which is very slow on soft-core in the FPGA), a hardware module could be created that enables low-overhead data-flow parallelism. The auto-generated tasks could also communicate with the hardware task-module to create more parallelism. Another direction would be to focus on the memory hierarchy; something that is missing from the current study. Through trace-driven profiling, a more optimal memory hierarchy could be designed to exploit the spatial or temporal locality in the task-parallel application. Tuning both the processing elements and the memory hierarchy could have a substantial effect on the performance. Another direction would be to couple the hardware with real-time scheduling of tasks – since we control what hardware is generated, we could more easily provide bounds on the worse-case with regards to tasks, possibly enabling hard real-time guarantees.

Bibliography

[1] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[2] D. Liu and C. Svensson, "Power consumption estimation in CMOS VLSI chips," *Solid-State Circuits, IEEE Journal of*, vol. 29, no. 6, pp. 663–670, 1994.

[3] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEICE Transactions on Electronics*, vol. 75, no. 4, pp. 371–382, 1992.

[4] R. H. Dennard, V. Rideout, E. Bassous, and A. Leblanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[5] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs journal*, vol. 30, no. 3, pp. 202–210, 2005.

[6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.

[7] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 365–376, IEEE, 2011.

[8] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[9] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[10] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *HighLevel Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[11] Xilinx, *Zynq-7000 All Programmable SoC*, 2015.

[12] Intel/Altera, *Intel Altera Heterogeneous Architecture Research Platform*, 2015.

[13] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the 40th annual Design Automation Conference*, pp. 338–342, ACM, 2003.

[14] M. J. Flynn and K. W. Rudd, "Parallel architectures," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 67–70, 1996.

[15] C. MEMORY, C. MEMORY, C. M. MEMORY, and P. P. P. PERIPHERAL, "Parallel operation in the control data 6600," *Readings in computer architecture*, p. 32, 2000.

[16] J. A. Fisher, *Very long instruction word architectures and the ELI-512*, vol. 11. ACM, 1983.

[17] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, *et al.*, "Tile64-processor: A 64-core soc with mesh interconnect," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 88–598, IEEE, 2008.

[18] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon DSP: An architecture optimized for mobile multimedia and communications," *Micro, IEEE*, vol. 34, no. 2, pp. 34–43, 2014.

[19] H. Sharangpani, "Intel® Itanium processor microarchitecture overview," in *Microprocessor Forum*, 1999.

[20] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE micro*, no. 2, pp. 39–55, 2008.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[23] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, no. 2, pp. 56–69, 2010.

[24] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[25] B. Wilkinson and M. Allen, *Parallel programming*, vol. 999. Prentice hall New Jersey, 1999.

[26] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[27] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Experiences parallelizing a web server with OpenMP," in *OpenMP Shared Memory Parallel Programming*, pp. 191–202, Springer, 2008.

[28] E. Baaklini, H. Sbeity, and S. Niar, "H. 264 macroblock line level parallel video decoding on embedded multicore processors," in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pp. 902–906, IEEE, 2012.

[29] R. H. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992.

[30] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[31] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[32] L. Dagum and R. Enon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[33] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 404–418, 2009.

[34] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: An optimizing, portable OpenMP compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

[35] C. Liao, D. J. Quinlan, T. Panas, and B. R. De Supinski, "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pp. 15–28, Springer, 2010.

[36] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos mercurium: a research compiler for openmp," in *Proceedings of the European Workshop on OpenMP*, vol. 8, 2004.

[37] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *ACM Sigplan Notices*, vol. 33, no. 5, pp. 212–223, 1998.

[38] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[39] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[40] K.-F. Faxén, "Wool-a work stealing library," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 93–100, 2009.

[41] C. C. Chi, T. Dallou, V. Garcia, C. Gou, S. Lyberis, X. Martorell, A. Mendelson, A. Muddukrishna, M. Pavlovic, M. R. Puzovic, *et al.*, "D5. 3 Final System Performance Evaluation Version 1.0,"

[42] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–11, IEEE, 2010.

[43] A. Podobas, M. Brorsson, and K.-F. Faxén, "A comparative performance study of common and popular task-centric programming frameworks," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 1, pp. 1–28, 2015.

[44] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *OpenMP in a new era of parallelism*, pp. 100–110, Springer, 2008.

[45] R. D. Blumofe and C. E. Leiserson, "Space-efficient scheduling of multithreaded computations," *SIAM Journal on Computing*, vol. 27, no. 1, pp. 202–229, 1998.

[46] F. W. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," in *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pp. 187–194, ACM, 1981.

[47] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–11, IEEE, 2008.

[48] C. E. Leiserson, "The Cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.

[49] M. A. Bender and M. O. Rabin, "Scheduling Cilk multithreaded parallel programs on processors of different speeds," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pp. 13–21, ACM, 2000.

[50] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Parallel Processing, 2009. ICPP'09. International Conference on*, pp. 124–131, IEEE, 2009.

[51] N. Manjikian and T. S. Abdelrahman, "Scheduling of wavefront parallelism on scalable shared-memory multiprocessors," in *Parallel Processing, 1996. Vol. 3. Software., Proceedings of the 1996 International Conference on*, vol. 3, pp. 122–131, IEEE, 1996.

[52] M. Andersch, C. C. Chi, and B. Juurlink, "Programming parallel embedded and consumer applications in OpenMP superscalar," in *ACM SIGPLAN Notices*, vol. 47, pp. 281–282, ACM, 2012.

[53] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the openmp accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators*, pp. 84–98, Springer, 2013.

[54] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[55] A. Podobas, M. Brorsson, and V. Vlassov, "Exploring heterogeneous scheduling using the task-centric programming model," in *Euro-Par 2012: Parallel Processing Workshops*, pp. 133–144, Springer, 2013.

[56] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 1299–1308, IEEE, 2013.

[57] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta, "Extending the OpenMP tasking model to allow dependent tasks," in *OpenMP in a New Era of Parallelism*, pp. 111–122, Springer, 2008.

[58] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, pp. 5–5, IEEE, 2006.

[59] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, *et al.*, "The design and implementation of a first-generation CELL processor-a multi-core SoC," in *Integrated*

Circuit Design and Technology, 2005. ICICDT 2005. 2005 International Conference on, pp. 49–52, IEEE, 2005.

[60] A. Pop and A. Cohen, "OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 53, 2013.

[61] P. Ghosh, Y. Yan, and B. Chapman, "Support for dependency driven executions among OpenMP tasks," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012*, pp. 48–54, IEEE, 2012.

[62] A. Podobas, M. Brorsson, and V. Vlassov, "TurboBŁYSK: Scheduling for Improved Data-Driven Task Performance with Fast Dependency Resolution," in *Using and Improving OpenMP for Devices, Tasks, and More*, pp. 45–57, Springer, 2014.

[63] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Locality-aware work stealing on Multi-CPU and Multi-GPU architectures," in *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, 2013.

[64] G. Barnes, "A method for implementing lock-free shared-data structures," in *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pp. 261–270, ACM, 1993.

[65] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *journal of parallel and distributed computing*, vol. 51, no. 1, pp. 1–26, 1998.

[66] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[67] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[68] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pp. 29–40, IEEE, 2002.

[69] H. Cho, P. D. P. Engineer, K. Chung, and T. Kim, "Benefits of the big. LITTLE Architecture," *EETimes, Feb*, 2012.

[70] S. Kestur, J. D. Davis, and O. Williams, "BLAS Comparison on FPGA, CPU and GPU.," in *ISVLSI*, pp. 288–293, 2010.

[71] R. Scrofano, S. Choi, and V. K. Prasanna, "Energy efficiency of FPGAs and programmable processors for matrix multiplication," in *Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on*, pp. 422–425, IEEE, 2002.

[72] D. Gohringer, M. Birk, Y. Dasse-Tiyo, N. Ruiter, M. Hubner, and J. Becker, "Reconfigurable MPSoC versus GPU: Performance, power and energy evaluation," in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pp. 848–853, IEEE, 2011.

[73] J. M. Bull and D. O'Neill, "A microbenchmark suite for OpenMP 2.0," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 5, pp. 41–48, 2001.

[74] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos, "A microbenchmark study of OpenMP overheads under nested parallelism," in *OpenMP in a New Era of Parallelism*, pp. 1–12, Springer, 2008.

[75] V. V. Dimakopoulos and A. Georgopoulos, "The OMPi OpenMP/C Compiler," in *Proc. PCI2005, 10th Panhellenic Conference on Informatics, Volos, Greece*, pp. 153–162, 2005.

[76] C. Addison, J. LaGrone, L. Huang, and B. Chapman, "OpenMP 3.0 tasking implementation in OpenUH," in *Open64 Workshop at CGO*, vol. 2009, 2009.

[77] F. Broquedis, T. Gautier, and V. Danjean, "LIBKOMP, an efficient openMP runtime system for both fork-join and data flow paradigms," in *OpenMP in a Heterogeneous World*, pp. 102–115, Springer, 2012.

[78] A. D. Robison, "Composable Parallel Patterns with Intel Cilk Plus.," *Computing in Science and Engineering*, vol. 15, no. 2, pp. 66–71, 2013.

[79] Oracle, "Sun Studio 12: OpenMP API User's Guide." http://docs.oracle.com/cd/E19205-01/819-5270/, 2010.

[80] M. Själander, S. McKee, B. Goel, P. Brauer, D. Engdal, A. Vajda, *et al.*, "Power-aware resource scheduling in base stations," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pp. 462–465, IEEE, 2011.

[81] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Department of Defense HPCMP Users Group Conference*, 1999.

[82] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 international conference on Power aware computing and systems*, pp. 1–8, USENIX Association, 2010.

[83] H. Jacobson, P. Bose, Z. Hu, A. Buyuktosunoglu, V. Zyuban, R. Eickemeyer, L. Eisen, J. Griswell, D. Logan, B. Sinharoy, *et al.*, "Stretching the limits of clock-gating efficiency in server-class processors," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 238–242, IEEE, 2005.

[84] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose, "Thermal-aware task scheduling at the system software level," in *Proceedings of the 2007 international symposium on Low power electronics and design*, pp. 213–218, ACM, 2007.

[85] A. K. Coskun, T. S. Rosing, and K. Whisnant, "Temperature aware task scheduling in MPSoCs," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 1659–1664, EDA Consortium, 2007.

[86] I. Yeo, C. C. Liu, and E. J. Kim, "Predictive dynamic thermal management for multicore systems," in *Proceedings of the 45th annual Design Automation Conference*, pp. 734–739, ACM, 2008.

[87] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan, "HotSpot: A compact thermal modeling methodology for early-stage VLSI design," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 5, pp. 501–513, 2006.

[88] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, IEEE, 2009.

[89] N. Minorsky, "Steering of Ships," 1984.

[90] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[91] M. Gomaa, M. D. Powell, and T. Vijaykumar, "Heat-and-run: leveraging SMT and CMP to manage power density through the operating system," in *ACM SIGARCH Computer Architecture News*, vol. 32, pp. 260–270, ACM, 2004.

[92] L. Xia, Y. Zhu, J. Yang, J. Ye, and Z. Gu, "Implementing a thermal-aware scheduler in linux kernel on a multi-core processor," *The Computer Journal*, vol. 53, no. 7, pp. 895–903, 2010.

[93] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pp. 259–268, IEEE, 2010.

[94] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu, "Adaptive work-stealing with parallelism feedback," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 3, p. 7, 2008.

[95] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[96] D. Kirk *et al.*, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, pp. 103–104, 2007.

[97] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[98] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "accULL: an OpenACC implementation with CUDA and OpenCL support," in *Euro-Par 2012 Parallel Processing*, pp. 871–882, Springer, 2012.

[99] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, *et al.*, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 451–460, ACM, 2010.

[100] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, 2002.

[101] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures," 2015.

[102] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "Analysis of dependence tracking algorithms for task dataflow execution," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, p. 61, 2013.

[103] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the StarSs programming model for platforms with multiple GPUs," in *Euro-Par 2009 Parallel Processing*, pp. 851–862, Springer, 2009.

[104] A. Filgueras, E. Gil, D. Jimenez-Gonzalez, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. Vissers, "OmpSs@Zynq All-programmable SoC Ecosystem," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, (New York, NY, USA), pp. 137–146, ACM, 2014.

[105] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot, "Constraint-driven identification of application specific instructions in the DURASE system," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 194–203, Springer, 2009.

[106] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36, ACM, 2011.

[107] Y. Leow, C. Ng, and W.-F. Wong, "Generating hardware from OpenMP programs," in *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pp. 73–80, Ieee, 2006.

[108] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 531–534, IEEE, 2012.

[109] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*, pp. 35–42, IEEE, 2009.

[110] Altera, "Altera Megafunction Overview User Guide," 2015.

[111] Altera, *Avalon Interface Specifications*, mar 2015.

[112] Altera, "NiosII Instruction Set Reference," 2015.

[113] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[114] C. Schulte, M. Lagerkvist, and G. Tack, "Gecode," *Software download and online material at the website: http://www. gecode. org*, 2006.

[115] Altera, *Stratix V Device Handbook Volume 1*, 2015.

[116] J. Ball, "The Nios II Family of Configurable Soft-Core Processors," *Hot Chips, Altera*, 2005.

[117] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[118] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[119] F. Nowak, M. Bromberger, and W. Karl, "An Architecture Framework for Porting Applications to FPGAs," in *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, pp. 1–7, VDE, 2014.

[120] N. Q. M. Noor, A. Saparon, and Y. Yusof, "An overview of microcode-based and FSM-based programmable memory built-in self test (MBIST) controller for coupling fault detection," in *Industrial Electronics & Applications, 2009. ISIEA 2009. IEEE Symposium on*, vol. 1, pp. 469–472, IEEE, 2009.

[121] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing ompss support for regions of data in architectures with multiple address spaces," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 359–368, ACM, 2013.