UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Parallel architectures and runtime systems co-design for task-based programming models

Emilio Castillo Villar

Parallel Architectures and Runtime Systems Co-Design for Task-based Programming Models

Emilio Castillo Villar

Barcelona, 2018

A thesis submitted in fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Department of Computer Architecture
Technical University of Catalonia

Parallel Architectures and Runtime Systems Co-Design for Task-based Programming Models

Emilio Castillo Villar

Barcelona, 2018

ADVISORS:

Miquel Moretó Planas
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Julio Ramón Beivide Palacio
Universidad de Cantabria

COLLABORATORS:

Marc Casas Guix
Barcelona Supercomputing Center
Lluc Álvarez Martí
Barcelona Supercomputing Center
Abhinav Bhatele
Lawrence Livermore National Laboratory
Nikhil Jain
Lawrence Livermore National Laboratory

A thesis submitted in fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Curso académico:

Acta de calificación de tesis doctoral

Nombre y apellidos

Programa de doctorado

Unidad estructural responsable del programa

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de su tesis doctoral titulada____

____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a	(Nombre, apellidos y firma) Secretario/a

(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

______________________, _______ de ____________________ de _______________

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Comisión Permanente de la Escuela de Doctorado, otorga la MENCIÓ CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente/a de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretario/a de la Comisión Permanente de la Escuela de Doctorado

Barcelona, _______ de ____________________ de _________

Mención Internacional en el título de doctor o doctora

- Como secretario/a del tribunal hago constar que parte de la tesis doctoral, como mínimo el resumen y las conclusiones, se ha redactado y presentado en una de las lenguas habituales para la comunicación científica en su campo de conocimiento y diferente de las que son oficiales en España. Esta norma no se aplica si la estancia, los informes y los expertos provienen de un país de habla hispana.

(Nombre, apellidos y firma) Secretario/a del Tribunal

To my family

Abstract

The increasing parallelism levels in modern computing systems has extolled the need for a holistic vision when designing multiprocessor architectures taking in account the needs of the programming models and applications. Nowadays, system design consists of several layers on top of each other from the architecture up to the application software. Although this design allows to do a separation of concerns where it is possible to independently change layers due to a well-known interface between them, it is hampering future systems design as the Law of Moore reaches to an end. Current performance improvements on computer architecture are driven by the shrinkage of the transistor channel width, allowing faster and more power efficient chips to be made. However, technology is reaching physical limitations where the transistor size will not be able to be reduced furthermore and requires a change of paradigm in systems design.

This thesis proposes to break this layered design, and advocates for a system where the architecture and the programming model runtime system are able to exchange information towards a common goal, improve performance and reduce power consumption. By making the architecture aware of runtime information such as a Task Dependence Graph (TDG) in the case of Asynchronous Task-based Programming (ATaP) models, it is possible to improve power consumption by exploiting the critical path of the graph. Moreover, the architecture can provide hardware support to create such a graph in order to reduce the runtime overheads and making possible the execution of fine-grained tasks to increase the available parallelism. Finally, the current status of inter-node communication primitives can be exposed to the runtime system in order to perform a more efficient communication scheduling, and also creates new opportunities of computation and communication overlap that were not possible before. An evaluation of the proposals introduced in this thesis is provided and a methodology to simulate and characterize the application behavior is also presented.

Abstract

El aumento del paralelismo proporcionado por los sistemas de cómputo modernos ha provocado la necesidad de una visión holística en el diseño de arquitecturas multiprocesador que tome en cuenta las necesidades de los modelos de programación y las aplicaciones. Hoy en día el diseño de los computadores consiste en diferentes capas de abstracción con una interfaz bien definida entre ellas. Las limitaciones de esta aproximación junto con el fin de la ley de Moore limitan el potencial de los futuros computadores. La mayoria de mejoras actuales en el diseño de los computadores provienen fundamentalmente de la reducción del tamaño del canal del transistor, lo cual permite chips mas rápidos y con un consumo eficiente sin apenas cambios fundamentales en el diseño de la arquitectura. Sin embargo, la tecnología actual esta alcanzando limitaciones físicas donde no será posible reducir el tamaño de los transistores motivando así un cambio de paradigma en la construcción de los computadores.

Esta tesis propone romper este diseño en capas y abogar por un sistema donde la arquitectura y el sistema de tiempo de ejecución del modelo de programación sean capaces de intercambiar información para alcanzar una meta común: La mejora del rendimiento y la reducción del consumo energético. Haciendo que la arquitectura sea consciente de la información disponible en el modelo de programacion, como puede ser el grafo de dependencias entre tareas en los modelos de programación *dataflow*, es posible reducir el consumo energético explotando el camíno crítico del grafo. Además la arquitectura puede proveer de soporte hardware para crear este grafo con el objetivo de reducir el *overhead* de construir este grado cuando la granularidad de las tareas es demasiado fina. Finalmente, el estado de las comunicaciones entre nodos puede ser expuesto al sistema de tiempo de ejecución para realizar una mejor planificación de las comunicaciones y creando nuevas oportunidades de solapamiento entre cómputo y comunicación que no eran posibles anteriormente. Esta tesis aporta una evaluación de todas estas propuestas así como una metodología para simular y caracterizar el comportamiento de las aplicaciones.

Acknowledgments

Those who know me know as well the last couple years had been a long and hard path towards this goal. I had the support of many people whom I like to briefly thank in the following lines:

The list of people who have made this possible in one way or another is extremely big, their friendship and unconditional support under all the circumstances that have arose in the past years is something that I will never be grateful enough.

My brother Javier Castillo, thank you for showing me research and always keep me in the right path.

My Ph.D advisor and co-advisors, Miquel Moretó, Ramón Beivide and Marc Casas, thank you for your understanding and support in difficult situations that went against your interests.

Mateo Valero, thank you for your friendship, thank you for making all of this possible.

Cristobal Camarero, most of what I know and I am able to do is thanks to you.

Kotaro Nakayama and Yutaka Matsuo from the University of Tokyo. Your understanding, support and concern has been always beyond words.

José Luis Bosque, thanks for helping me both professionally and personally all these years.

All the RoMoL team, especially Lluc Álvarez, César Allande and Adrian Barredo.

Martin Schulz from the Technical University of Munich, Abhinav Bhatele and Nikhil Jain from the Lawrence Livermore National Laboratory. Thank you for hosting me last year and work hard side-by-side to make this thesis come to a successful end.

Satoshi Matsuoka from Tokyo Tech. Thank you always for your kindness and advice.

My parents supported me throughout all these years and their efforts guaranteed me an education and a future.

Old colleagues at the University of Cantabria, Pablo Fuentes, Iván Pérez, Fernando Vallejo, Enrique Vallejo, Carmen Martínez, Rafael Menéndez and Esteban Stafford.

All my friends in Spain, USA, and Japan. Too many to list all of them here but I will always feel grateful to all of you.

Financial Support

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P, TIN2012-34557, TIN2013-46957-C2-2-P, TIN2016-76635-C2-2-R and TIN2016-81840-REDT), by Spanish Ministry of 41 Education, Culture and Sports under grant FPU2012/2254, by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence. The Mont-Blanc project receives funding from the EU's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 610402 and from the EU's H2020 Framework Programme (H2020/2014-2020) under grant agreement No. 671697 and No. 671610. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Contents

Chapter 1

Introduction

Current trends in computer architecture research are being heavily driven by the end of Moore's Law, which predicted that the number of transistors in integrated circuits doubles every two years. For decades the increasingly available transistors were used to improve the Instructions per Cycle (IPC) in-order processors could execute by adding pipelines, branch-predictors and cache memories that greatly increased the instruction throughput of single-core processors. As the transistor count kept rising in a silicon die, computer architects were able to improve the Instruction Level Parallelism (ILP) of sequencial programs by using superscalar and Out-Of-Order execution creating opportunities to hide latency of long memory accesses.

Furthermore, Dennard's scaling stated that the power density of MOSFET transistors stays constant as their size decreases allowing to increase the frequency while the power remains constant. However, Dennar scaling broke down around 2005 and 2007 and the stagnation of CPU clock frequencies due to the impossibility of dissipating the generated power density led to the rise of multi-core systems thanks to the increasingly available transistors in the silicon die. This made Task Level Parallelism (TLP) possible and allowed real parallelism in single-chip systems.

The high levels of concurrency offered by multi-core systems are hard to exploit through programming models based on thread primitives. Thus, new programming models have emerged to deal with such architectures and the *Programmability Wall*. Asynchronous Task-based Programming (ATaP) models and runtimes such as OpenMP 4.0 [92], Charm++ [2], HPX [66], OmpSs [38], and Legion [11] implement a dataflow execution model where the programmer splits the code in sequential pieces of work, called *tasks*, and specifies the data and control dependences between them creating a Task Dependence Graph (TDG). With this information, the runtime system manages the parallel execution, scheduling tasks to cores and taking care of synchronization among them. These models not only improve programmability, but also can increase performance by avoiding global synchronization points, enabling the user to focus on programming aspects related to their problem domain without worrying about cross-platform performance issues. Under the hood, the runtime systems are designed to automatically optimize for different application scenarios and system specifications.

The algorithmic information that the runtime system holds on ATaP models offers several opportunities to tackle the most common issues that computer architects face today; The *Power Wall*, where processor clock stagnated due to the impossibility of effectively dissipating the power density once that frequency reaches a certain threshold, and the *Memory Wall* where the main memory latency does not decrease fast enough to keep up with the processor timing requirements. This PhD thesis envisions a *Runtime-Aware Architecture* (RAA) in which the software runtime system and the hardware are co-designed so that the hardware layer can provide support for expensive task management operations and some decisions at the architectural level such as power management, or inter-node communication are driven by the information obtained at the runtime system layer.

The Power wall originated due the impossibility of keep increasing the frequency of processors due to the power and thermal dissipation constrains that the CMOS technology imposes. Modern computer systems implement different hardware mechanisms that allow reconfiguring the computational capability of the system, aiming to maximize performance under affordable power budgets. For example, per-core power gating and Dynamic Voltage and Frequency Scaling (DVFS) are common reconfiguration techniques available on commodity hardware [36, 68]. However, the problem of optimally using these reconfiguration mechanisms remains open. The information in the TDG can be exploited to open new power savings opportunities and performance gains taking into account its critical path. Tasks outside of the critical path in the TDG can be executed in simpler processing elements or at lower clock frequencies using reconfiguration mechanisms such as DVFS. Asymmetric chip multiprocessor systems present a power-efficient alternative to classical symmetric chip multiprocessor systems by combining high performance out-of-order cores with more power efficient but slower in-order cores. Tasks can be scheduled to the cores by looking at their criticality in the TDG [32] or the speedup they could obtain from running in the big cores [63]. In a RAA, the architecture can provide the metrics the runtime system needs to efficiently map tasks to processing elements and offer mechanisms to perform transparent migration of such tasks when needed.

The increasing number of cores per processor forces developers to decrease the granularity of the tasks to prevent starvation and expose large degrees of concurrency to the hardware, which favors load balancing and provides more flexibility to exploit constructive interference on shared resources. However, it can also bring large software overheads due to the runtime system activity, which involves creating the tasks, tracking the dependences between them, and scheduling them to cores. All these actions require synchronizing threads to perform complex operations on internal data structures of the runtime system. Different solutions have been proposed to support fine-grained parallelism on multi-cores [39, 59, 60, 71, 116]. These approaches manage fine-grained tasks completely in hardware, relying on specific execution models to scale to large core counts. However, pure hardware solutions suffer from limited adaptability to changes in the software layers. Task support in shared memory programming models is continuously evolving, incorporating new features such as dependence domains or task nesting that are not easy to support at the architecture level. Moreover, implementing a fixed scheduling policy in hardware reduces the adaptability to different application and system characteristics.

Finally, one of the main attractions of ATaP models and runtimes is their potential to automatically expose and exploit the overlap of computation with communication. This refers to the benefit that comes from having multiple tasks on a physical core or process so that when one task is waiting for messages to arrive, another task can use the idle core for useful computation. When ATaP applications are executed on distributed memory systems, inter-node communication is typically handled by calls to a messaging library (in most cases, the Message Passing Interface or *MPI*). Some ATaP models allow explicit calls to MPI whereas in other models, communication primitives are translated to MPI calls by the runtime [38, 113]. The interoperability of the messaging library and the ATaP model has been subject of extensive research, from the programmer explicitly calling communication primitives [82] to the runtime system abstracting the message exchanging [2, 11]. All these works treat the communication layer as a black-box entity and there is a clash on the design philosophies between both, the ATaP and the communication model, preventing to achieve better performance unless the MPI runtime and the ATaP runtime system collaborate together as proposed in this work.

1.1 Thesis Objectives and Contributions

The main goal of this thesis is to study scenarios where the software/hardware co-design of a multi-core architecture and the programming model runtime system can be beneficial and use the underlying resources in an efficient and transparent way to the programmer.

The contributions of this dissertation explore several critical scenarios for parallel architectures: power management, parallel workloads performance and load balancing, And inter-process communication in distributed memory systems. This thesis advocates for an integrated proposal to benefit from the information that lives in the intersection of the runtime system and the architecture, enabling the latter to exploit runtime level information or propose mechanisms to ease bottlenecks at the runtime system layer by exploiting lower layers internal state information.

1.1.1 Efficient Power Management based on Task Criticality

The TDG offers a rich degree of information that can be exploited by ATaP runtime systems opening a wide range of performance and power optimization opportunities.

Based on the observation that task criticality information can be exploited to drive hardware reconfigurations, we propose a Criticality Aware Task Acceleration (CATA) mechanism that dynamically adapts the computational power of a task depending on its criticality. As a result, CATA achieves significant improvements over a baseline static scheduler, reaching average improvements up to 18.4% in execution time and 30.1% in Energy-Delay Product (EDP) on a simulated 32-core system.

The cost of reconfiguring hardware by means of a software-only solution rises with the number of cores due to lock contention and reconfiguration overhead. Therefore, novel architectural support is proposed to eliminate these overheads on future manycore systems. This architectural support minimally extends hardware structures already present in current processors, which allows further improvements in performance with negligible overhead. As a consequence, average improvements of up to 20.4% in execution time and 34.0% in EDP are obtained, outperforming state-of-the-art acceleration proposals not aware of task criticality.

The following sections describe the mechanism and its operational model as well as the integration with the runtime system and a complete evaluation of performance and energy savings of the proposal.

1.1.2 Flexible Hardware Support for Task Dependence Management

The growing complexity of multi-core architectures has motivated a wide range of software mechanisms to improve the orchestration of parallel executions. Task parallelism has become a very attractive approach thanks to its programmability, portability and potential for optimizations. However, with the expected increase in core counts, fine-grained tasking is required to exploit the available parallelism, which increases the overheads introduced by the runtime system.

This work presents Task Dependence Manager (TDM), a hardware/software co-designed mechanism to mitigate runtime system overheads. TDM introduces a hardware unit, denoted Dependence Management Unit (DMU), and minimal ISA extensions that allow the runtime system to offload costly dependence tracking operations to the DMU and to still perform task scheduling in software. With lower hardware cost, TDM outperforms hardware-based solutions and enhances the flexibility, adaptability and composability of the system. Results show that TDM improves performance by 12.3% and reduces EDP by 20.4% on average with respect to a software runtime system. Compared to a runtime system fully implemented in hardware, TDM achieves an average speedup of 4.2% with $7.3\times$ less area requirements and significant EDP reductions. In addition, five different software schedulers are evaluated with TDM, illustrating the flexibility and performance gains of our approach.

1.1.3 Towards a Seamless ATaP-MPI Interoperability

.

Communication in hybrid parallel and distributed applications is usually handled by the MPI Library. In such applications, computation and communication phases are well-defined and isolated from each other limiting the potential computation-communication overlap.

ATaP models are gaining popularity to address the programmability and performance challenges in high performance computing. One of the main attractions of these models and runtimes is their potential to automatically expose and exploit overlap of computation with communication.

ATaP models can deal with communication by either allowing the programmer to place explicit MPI calls in the tasks code, or the runtime system can be communication aware and schedule data transferences when appropiated. Although this exposes a transparent and natural computation-communication overlap, there are still some inefficiencies due to the lack of information sharing between the ATaP and the communication layer runtime systems.

This chapter explore how information about MPI internals can be exposed and used in a task-based runtime system to make better scheduling and task-creation decisions. In particular, we present two mechanisms for exchanging information between MPI and a task-based runtime, and analyze their trade-offs. Further, we present a detailed evaluation of the proposed mechanisms implemented in MPI and a task-based runtime. We show improvements of up to 16.3% and 34.5% for proxy applications with point-to-point and collective communication, respectively.

1.2 Thesis Structure

The document contents are organized as follows:

- Chapter 2 reviews parallel architectures and programming models and provides the necessary information to understand the rest of the chapters.
- Chapter 3 describes all the tools and benchmarks used for the evaluation of the presented proposals.
- Chapter 4 presents *Criticality-Aware Task Acceleration (CATA)* a mechanism to drive the power management of a multi-core chip multiprocessor. The chapter explains the problems that previous alternatives faced and later describes the proposed solutions and their evaluation.
- Chapter 5 introduces *Task Dependence Manager (TDM)* a flexible solution proposing hardware support to accelerate the Task Dependence Graph construction while keeping the task scheduler in software. First the motivation behind the proposal is explained by analyzing the bottlenecks of a runtime system, then the architecture of *TDM* is detailed and a complete design space exploration and evaluation are provided.
- Chapter 6 explores how the blocking time of worker threads can be reduced when using MPI taskified calls by exposing MPI and network interface related events to the ATaP model runtime system. Two alternatives events delivery mechanisms are described and evaluated, as well as a new proposal for performing computation/communication overlap when using collectives.
- Chapter 7 details how all these proposals interact together, lays the theoretical foundations of a Runtime Aware Architecture, and explains the operational model and integration of all the components with their interface.

- Finally, Chapter 8 concludes the dissertation by remarking the main contributions and providing an insight of future work.

Chapter 2

Background and Related Work

This Chapter introduces the background context and does an analysis of the state-of-the-art for all the concepts laid out in this thesis. Section 2.1 describes multiprocessors systems focusing on their memory models, shared memory and distributed memory multiprocessor systems, the concept of asymmetric multiprocessor systems is also detailed. Section 2.2 explains parallel programming models, starting from the omniscient threading programming model and OpenMP basic parallel constructs to the more complex ATaP models that can be used in both shared and distributed memory systems. Section 2.3 introduces OmpSs, the programming model used in this thesis and explains its most relevant characteristics. Next we detail hardware support for task-based programming models proposals present in the literature. Finally, we focus on distributed environments and explain the classical MPI programming model and how ATaP models interact with the communication library.

2.1 Parallel Multiprocessor Systems

Multiprocessor systems date from the era of the large mainframes in the 1960s [47]. However, it was not until the 1980s that these systems became common in computing infrastructures. The first systems consisted of bus-based multiprocessors with snooping caches [43] that were primitive implementations of the coherence protocols present on current multiprocessors. Many advances were done in those years, laying the foundations for both shared and distributed memory systems.

2.1.1 Shared Memory Multiprocessor Systems

Most common shared memory systems provide a hierarchical shared memory address space to a series of processors managed through a memory coherence protocol implemented in hardware. The coherence protocol has the responsibility of making one processor changes to the private levels in the memory hierarchy visible to other processors in the system and coordinate the access to the shared memory locations to provide support for atomic operations used to implement inter-processor synchronization primitives.

Usually these systems include multiple full-fledged physical processors holding directories of memory locations that they can access either locally or by requesting them through the interconnection network. We refer to such schema as *distributed shared memory processors* or *Non-Uniform Memory Access (NUMA)* since physically separated memories can be accessed as a logically unified and shared space.

While original shared memory systems consisted of physically independent processors, the increasing number of transistors in a silicon die [85], the ILP limitations and the stagnation of processor frequencies motivated a shift into the design of processors. Rather than devote more transistors to exploit ILP, hardware support for TLP was devised bringing in Simultaneous Multi-Threading (SMT) in which a subset of the processor structures holding the architectural status are replicated [123]. Eventually the increasingly available transistor count per silicon die allowed to integrate several processor into a single die, resulting in the Chip MultiProcessors (CMP), a trend that started with the IBM POWER4 [119] , the world's first dual core processor. Figure 2.1 shows a CMP with four cores and two levels of cache. Each core has a private instruction and data L1 cache, and a shared distributed L2 cache that is accessed through the interconnection network. Nowadays, multiprocessor systems range from 4-8 cores for mobiles and desktop computers and up to 72 cores as in the Intel Xeon Phi or the AMD Threadripper.

2.1.2 Distributed Multiprocessor Systems

While shared memory systems offer the programmer a logical unified shared memory address space between all the processing elements in the system, distributed memory systems have multiple address spaces where data is shared through a message passing protocol using the interconnection network. In this thesis we focus on distributed memory multiprocessor High Performance Computing (HPC) systems, which, are composed of several independent nodes connected together using a low-latency high-bandwidth in-

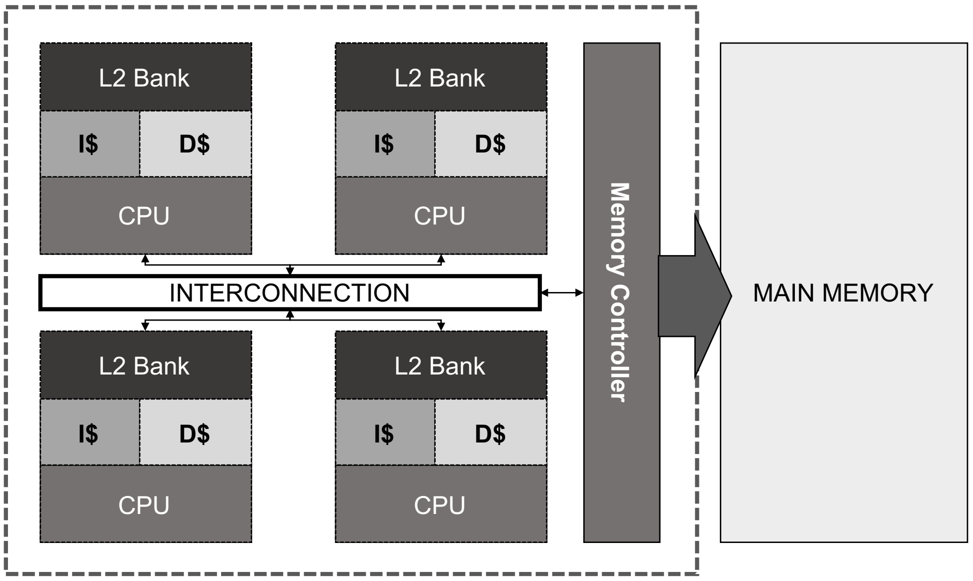


Figure 2.1: Example of a chip multiprocessor system.

terconnection network. Typically, each node houses one, two or four CMPs in a NUMA shared memory fashion and is connected to other nodes using a low latency high bandwidth network protocol such as Infiniband or Myrinet. In several machines, accelerators such as GPUs or FPGAs can be present in a subset or all the nodes providing heterogeneous computing capabilities.

The core count in the world fastest supercomputers has dramatically increased in the recent years. According to the Top500 list [1] which is published twice every year, in 1998 the #1 Supercomputer, ASCI RED had 9,152 cores. Ten years later, in 2008, Roadrunner was made of 122,400 cores, and in 2018, Summit consists of 2,282,544 cores. The growing number of cores in CMPs make possible this continuous increase while keeping energy and cooling costs in a reasonable budget. However, it is complicated for programmers to take advantage of such large-scale system as message sending and reception are left to the application programmer or programming model designers. Heterogeneity is also present in the Top500 systems and is becoming increasingly important in order to achieve Exascale; 27.4% of the systems in the Top500 list of November 2018 uses some kind of accelerator. Summit hosts 27,648 Nvidia Volta GV100 GPU accelerators connected directly to the POWER9 CPUs with the NVLink interconnection.

2.1.3 Asymmetric Multiprocessor Systems

While shared memory multiprocessor systems are usually conceived as symmetric systems where all the processing elements in the system share the same characteristics, the heterogeneity of workloads and current power constraints in embedded and large scale systems design has motivated the design of asymmetric systems, in which the processing elements can exhibit different performance ratios. However, it is worth mentioning that current homogeneous systems can exhibit from negligible to moderate asymmetry in performance due to variability in the manufacturing process, or not proper cooling as demonstrated in [81, 106].

Heterogeneous systems are asymmetric by nature as their elements do not provide the same capabilities. Such systems are usually targeted to certain problem cases such as accelerators for graphics, cryptography, vector operations and communications among others. Another factor that guides the asymmetric system design is power efficiency. In-order cores offer a trade-off in power/performance when compared to power hungry out-of-order architectures. The ARM big.LITTLE processor is an Asymmetric Multi-core (AMC) architecture that combines both classes of cores on two core clusters of 4 cores each; a high-performance cluster based in the Cortex A-57 architecture, and a low-power consumption cluster with Cortex A-53 cores. In this system, high-preforming cores can run critical tasks while the smaller cores are targeted to non-critical low-resources demanding processes guided by the OS scheduler. However, in multi-threaded workloads, is the programmer responsibility to map each thread to a core taking in account the differences in performance.

In general, asymmetric systems have serious drawbacks in terms of flexibility as different phases of an application might be optimally executed in different computation units. Consequently, the overhead of moving tasks and their associated data to different computation units can neglect the expected energy improvements. An alternative is to have a highly reconfigurable hardware that adapts to the actual requirements of the running application. Reconfiguring the underlying hardware eliminates the data movement burden, but adds new issues as the reconfigurations must be coordinated to avoid an explosion in terms of power consumption.

Current systems offer many different hardware mechanisms that allow reconfiguring the computation power of the system. For example, in SMT processors, the number of SMT threads per core or the decode priority can be adjusted [18, 125], while in multi-cores, the prefetcher aggressiveness, the memory controller or the last-level cache space assigned to an application can be changed [33, 62, 102]. More recently, reconfigurable systems that support core fusion or that behave like traditional high performance out-of-order cores, but can be transformed to a highly-threaded in-order SMT core when required, have been shown to achieve significant reductions in terms of energy consumption [56, 70]. Also, per-core power gating and DVFS are common reconfiguration techniques available on commodity hardware [36, 68]. However, the problem of optimally reconfiguring the hardware is not solved in general as all the above mentioned solutions rely on effective but ad-hoc mechanisms that are applicable to a reduced set of reconfiguration problems, they are difficult to combine [125], and they introduce a significant burden on the programmer.

Multiple techniques to exploit heterogeneous architectures and reconfiguration capabilities have been proposed, such as migrating critical sections to fast cores [117], *fusing* cores together when high performance is required for single-threaded code [56], reconfiguring the computational power of the cores [70], *folding* cores or switching them off using *power gating* [125], or using *application heartbeats* to assign cores and adapt the properties of the caches and the TLBs according to the specified real-time performance constraints [52].

2.2 Parallel Programming Models

2.2.1 Threading and Fork-Join Programming Models

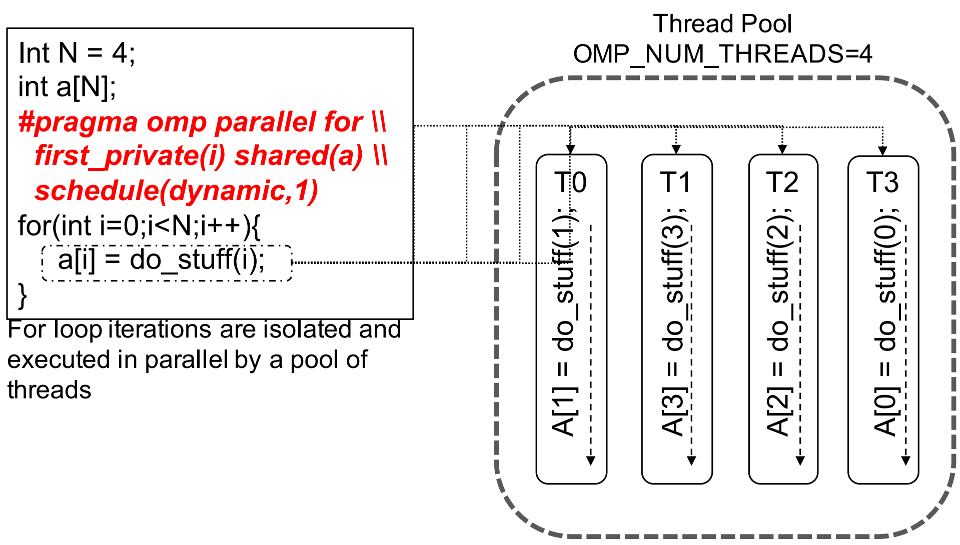
One of the most common parallel programming models in production-grade software nowadays is the explicit use of light-weight processes named threads. A thread is a analogous to a forked process which shares the application text, data and heap segments with its parent process. A thread also retains its own architectural context such as the CPU registers including the instruction pointer, and the memory stack. Most common operating systems provide facilities to create and manage threads through the use of the POSIX Threads (*pthreads*) programming interface [86]. Pthreads or System-Level Threads (SLTs) can be created by a single POSIX API call, and they are scheduled as independent processes that can run in any core of the system, the pthread interface also provides means to synchronize threads when accessing to shared memory regions through the use of semaphores.

An alternative to SLTs are User Level Threads (ULTs) [83]. ULT libraries allocate a set of resources (processors and memory) and manage thread creation and scheduling in user-level space through a runtime system. ULTs are usually mapped to a pool of existing SLTs but providing extended capabilities such as more complex and safer synchronization mechanisms and faster thread creation and context switching or custom scheduling to better exploit locality.

The complexity of programming using threads has driven the creation of mechanisms to ease the programmability of parallel systems. OpenMP is a set of compiler directives for C, C++ and FORTRAN based on *pragma annotations*. The programmer adds the *#pragma omp parallel* directive preceding the blocks of code that will run in parallel. OpenMP offers a *#pragma omp parallel for* directive for parallelizing *for* loops without inter-iteration dependences. Loop iterations scheduling across threads can be configured by the *schedule* directive. Alternatively, work can be distributed among threads by using the *parallel section* annotation and the following block of code will be scheduled for execution in any free thread if no affinity restrictions are placed. In OpenMP, access to the data can be controlled by declaring data as *shared* or *private* in the *omp* directives; *first private* is a special case used to make a local copy of a shared variable before the execution of the forked code. In addition, thread synchronization and access to shared data can be protected with the use of *critical*, *atomic* or *barrier* annotations. Figure 2.2 shows an example of how several threads execute parallel for loop iterations using a *dynamic* scheduler which maps iteration ranges to any available free thread.

In OpenMP source codes, the compiler detects the parallel blocks and abstracts them into functions that are passed as arguments to the OpenMP runtime system API. Moreover, the compiler also inserts the relevant API calls for synchronization in shared data management. The OpenMP runtime system consists of a pool of worker threads and a scheduler with work-queues to execute the parallel functions once they are created. This runtime system is also in charge of coordinating access to the shared data.

OpenMP sets the basis of modern parallel programming models where the cooperation between the compiler and a runtime systems allows programmers to express parallelism in easier and platform-agnostic environments closer to the algorithmic concepts rather than complex operating system level libraries and architectural concepts. Furthermore, this synergy allows to find hidden parallelism thanks to the analysis of the original source code and even increase performance by providing means to exploit data locality that programmers would not be aware of.



```
Int N = 4;
int a[N];
#pragma omp parallel for \\
  first_private(i) shared(a) \\
  schedule(dynamic,1)
for(int i=0;i<N;i++){
   a[i] = do_stuff(i);
}
```

Figure 2.2: OpenMP parallel *for* implementation

2.2.2 Asynchronous Task-based Programming Models

Programming current large scale parallel systems has motivated a resurgence of ATaP models over explicit threading and fork-join approaches due to the programmability-performance trade-off they offer. This section aims to provide an insight on the most relevant ATaP models.

Task constructs are subroutines that can be executed asynchronously and in parallel in different threads. Tasks have synchronizing dependences among them to structure the control flow of the application. OpenMP 3.0 [91] introduced basic task constructs in the programming model that OpenMP 4.0[92] extended with data dependences annotations in order to create a dataflow ATaP programming model. The OpenMP 4.0 runtime system analyzes this annotations and creates the TDG as will be detailed in section 2.3. Programming models such as OmpSs [38] or StarPU [7] follow this schema.

Habanero [113] proposes a set of extensions to the C and Java languages to create asynchronous tasks with dataflow capabilities using futures, a language construction that allows tasks to synchronize and exchange data. Habanero also proposes phasers [112] for control synchronization. Codelets [128] are collections of instructions that are executed atomically and created a Codelet Graph (CDG) through the use of data and control dependences. Codelets rely on a hierarchical and heterogeneous abstract machine model to schedule and execute the work units. Intel Thread Building Blocks (TBB) [103] is a C++ template library with multiple constructions for parallel operations such as *parallel_for*, *parallel_scan* or parallel containers among many others. TBB allows the use of tasks that are created in a recursive manner with control dependences to synchronize them throughout the hierarchy. Furthermore, Intel TBB 4.0 introduces flow graph annotations, which allow to specify a dataflow graph between different tasks. Cilk [17] includes a set of C and C++ extensions that allows to *spawn* and *sync* tasks. However, dataflow annotations have been added outside of the standard language by Vandierendonck et al. [124].

Charm++ [2] decomposes work in units called *chares*. Chares are C++ objects that offer a series of entry points that other chares can directly invoke. The runtime system then sends the appropriate messages to synchronize and exchange data between chares in a transparent way to the programmer. Charm++ offers mechanisms to easily map chares to tensor-like data structures, distribute them, and overdecompose data volumes in more chares than processing elements. Moreover, as chares entry method invocation is non-blocking, Charm++ allows programmers to specify control dependences to ensure structured flow control between chares.

Legion [11] creates abstractions for both computation and data. Computation is written as tasks with data partitions as their inputs and output. The Legion runtime system then builds a TDG using the inputs and outputs information in a dataflow manner. Partitions are logical abstractions of data expressed as array-of-structures or structures-of-arrays. The Legion programming model allows multiple versions of the data to be alive at a given moment and the runtime system is responsible for moving the data across the system to the node where the task requiring that specific version will be executed.

The use of ATaP models provides a hardware abstraction that allows programmers to efficiently write portable code without caring about low-level details while increasing the programmability of parallel systems. Moreover, these programming models can increase the performance of applications through the use of generic optimizations at the runtime system level such as data locality aware scheduling in NUMA systems, or tasks schedulers that favors load balancing. In addition, some of this programming models, such as Charm++ or Legion, allow applications to run in distributed computing environments with virtually no changes to the application code, effectively hiding the complexity of distributed programming to the application developers. To conclude, the benefits of ATaP models, portability, programmability and performance are key in modern computing infrastructures and noteworthy to remark.

2.2.3 Programming Models for Distributed Environments

Early distributed systems programming relies on the explicit use of the communication hardware to send messages across the network interconnecting all the nodes composing the computer. This approach, while providing great levels of performance hindered application portability and programmability as most of the interconnection technologies and interface code are system-specific. As a result, an effort has been made to provide a layer of abstraction on top of the communication one to ease programmability and portability efforts. As a consequence, the current de-facto standard for distributed memory communication is the Message Passing Interface (MPI) [114], an easy-to-use, portable, high-performing abstraction on top of most low-level communication technologies present in distributed memory clusters.

MPI provides an interface to send data by calling point-to-point or collective send and receive operations. Point-to-point primitives involve communication within a pair of nodes, while collective operations involve a group of nodes. Collective operations are highly tuned to the underlying communication technology and they execute different algorithms for a single collective operation based on the network topology and the size of the messages [120].

MPI offers two communication models; blocking and non-blocking communication. In the blocking model, whenever a blocking call is invoked the execution cannot progress until the message has been copied to a safe location. In the alternative non-blocking communication, the call returns immediately, but it is the programmer's responsibility to poll the status of the pending messages to ensure the progress and completion of the communication. Although, using non-blocking calls allows overlapping communication and computation, the MPI layer does not make any progress until MPI_Test or Wait calls on specific requests are invoked, resulting in no effective overlap at all unless explicit mechanisms are used as demonstrated in [50]. Figure 2.4 shows how the actual message is not send until a specific wait is invoked on the MPI request, thus overlap is not guaranteed unless is driven through specific threads.

In order to overcome this limitation, Hoefler and Lumsdaine [50] study the implications of using a dedicated thread to constantly advance the progress engine. They compare this approach to manual and hardware interrupts-driven progression by using the Infiniband RDMA engine. In addition, Buettner et al. [22] propose to taskify communication and offload the MPI_Test call required to move the progress engine to the OpenMP run-

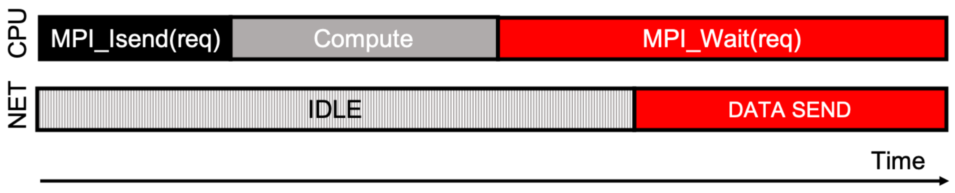


Figure 2.3: Non-blocking communication does not guarantee overlapping

time.

This progress engine limitations arise from MPI design being heavily influenced by Bulk-Synchronous programming models, resulting in limited interoperability with parallel shared-memory programming models. Although the standard [42] and MPI vendors thrive to enhance this interoperability, some of the MPI characteristics require changes to the standard or restrict some capabilities such as wildcards in order to efficiently parallelize the MPI message matching engine.

MPI alternatives such as the Adaptive Message Passing Interface (AMPI) [57], appeared to enhance the multi-threading support of MPI. AMPI implements MPI ranks as user migratable lightweight threads instead of full-fledged OS processes. AMPI is written in top of Charm++ and takes advantage of the dynamic load-balancing and scheduling algorithms it offers, allowing regular MPI applications to benefit from these characteristics without the need of being rewritten.

2.2.4 Communication in Hybrid MPI+ATaP Models

MPI is employed inside ATaP programming models using two different approaches. The *explicit communication* model requires the programmer to insert the relevant MPI calls in the application code and do a manual orchestration of all the MPI ranks as illustrated in figure 2.4 left hand side. OpenMP 4.0 [92], OmpSs [38], Codelets [128], Habanero [113] or StarPU [7] are ATaP models used for shared memory systems that require explicit communication in order to take advantage of distributed systems.

Alternatively, other programming models hide communication from the programmer by letting the runtime detect accesses to remote data and perform the required transferences. This approach is called *implicit communication* or *Runtime managed* as shown in figure 2.4 right hand side. Some examples of this model follows. Charm++ employs a communication interface that relies on active messages built on top of MPI. Legion [11] detects accesses to partitions of data in remote nodes and internally schedules data trans-

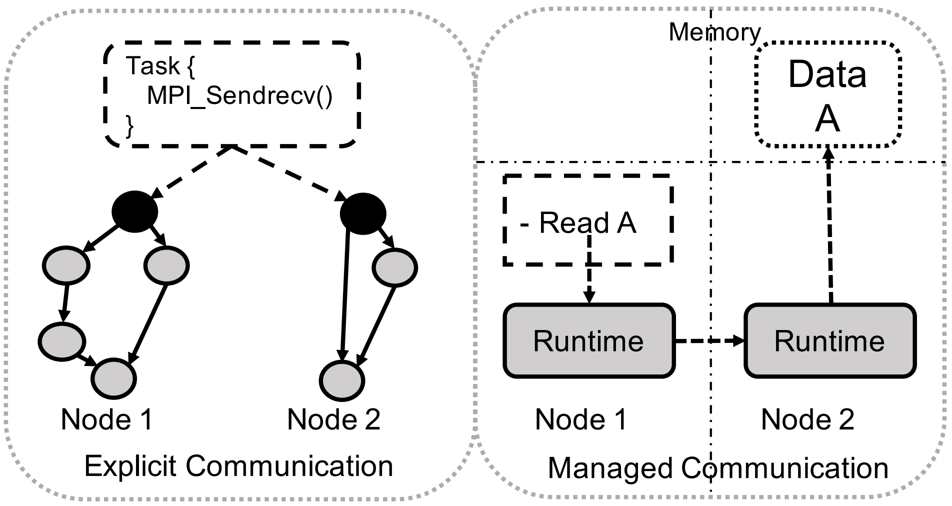


Figure 2.4: Different communication mechanisms for ATaP models

ferences by means of communication threads and active messages. HPX [66] offers a PGAS model in which tasks can directly address memory in any node with the runtime system taking care of data movement.

Improving the MPI and programming model interoperability has been am extensive subject of research, Marjanovic et al. [82] present one of the first works focused on the interoperability of a task-based asynchronous programming models, such as OmpSs, with MPI. Instead of doing synchronized phases of communication and computation, as in the Bulk Synchronous Programming Model, MPI calls are placed inside of asynchronous tasks scheduled by the runtime system effectively opening new opportunities for communication and computation overlap. Chatterjee et al. [29] goes one step further and integrates MPI within Habanero providing wrapped MPI calls that the runtime executes asynchronously in dedicated communication threads. Labarta et al. [73] present the Task-Aware MPI library (TAMPI), a similar approach to improve the interoperability between MPI and OmpSs. TAMPI intercepts blocking calls to MPI and converts them into their non-blocking counterpart. The resulting MPI calls are managed by the TAMPI library, which periodically polls for the completion of the MPI calls and ensures correctness. However, TAMPI is limited to point-to-point communications and requires polling to query for completion of specific calls. Kamal et al. [67] make use of ULTs in the MPICH 2 [45] to build an MPI-aware scheduler for coroutines that are swapped in and out for execution depending on the status of the MPI runtime. Lu et al. [77] follow a similar approach by doing the context switch of ULTs inside the MPI to avoid the expensive MPI locking operations. Stark et al. [115] integrate MPI with Qthreads and convert

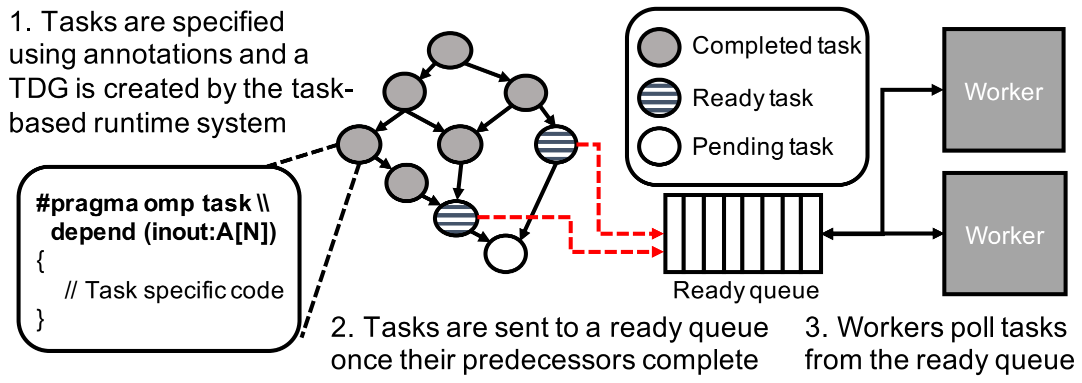

1. Tasks are specified using annotations and a TDG is created by the task-based runtime system
#pragma omp task \\ depend (inout:A[N])
{
// Task specific code
}
Completed task
Ready task
Pending task
Worker
Worker
Ready queue
2. Tasks are sent to a ready queue once their predecessors complete
3. Workers poll tasks from the ready queue

Figure 2.5: Execution flow in an asynchronous task-based programming model that uses a TDG.

blocking MPI calls to non-blocking calls, using their status to drive the scheduler in a similar way to TAMPI.

2.3 The OmpSs Programming Model

ATaP models such as OpenMP 4.0 [92] and OmpSs [38], conceive the execution of a parallel program as a set of tasks that may depend upon each other. Typically, the programmer defines code blocks (functions and/or classes) and adds annotations to declare 1) what constitutes a task, 2) what data is used by each task, called input dependences or input, and 3) what data is produced by each task, called output dependences or output. Based on this information, the runtime system manages the parallel execution using a TDG, a directed acyclic graph where the nodes are tasks and the edges are the dependences between these tasks.

Figure 2.5 shows a simple example of a TDG. A task is marked as ready only when all its predecessors have completed their execution, otherwise it is considered a pending task. Ready tasks are added to a *ready queue* (or another appropriate data structure depending on the scheduling algorithm). When idle, worker threads interact with the scheduler and retrieve tasks for execution. When a task completes, it is marked as such in the TDG and its successors are unlocked. Examples of such a programming model are tasks in OmpSs [38] and OpenMP 4.0 [92] with dependence clause extensions and Legion [11], in which dependences between tasks are expressed using regions.

This thesis uses the task constructions in OmpSs [38], which have been adopted as the task extensions for OpenMP 4.0. The programmer creates tasks using pragma annotations

Algorithm 1: Algorithm for *TDG* creation.

```
Data: task object
Data: task memory locations list
for location in locations do
    if lastWriter of location != NULL then
        lastWriter.successors.add(task) task.predecessors++;
    end
    if location.dir == IN then
        location.readers.add(task);
    end
    if location.dir == OUT then
        for reader in location.readers do
            reader.successors.add(task); task.predecessors++;
        end
        location.readers.flush(); location.lastWriter = task;
    end
end
```

with the input and output dependences specified as shown in Figure 2.5. The compiler replaces these annotations with calls to the runtime system, and the tasks are dynamically created and destroyed during the application execution. In this work, we employ Nanos++ 0.10a [38], the runtime of OmpSs, which uses pthreads bound to specific cores as worker threads.

2.3.1 Task Dependence Graph

In ATaP programming models such as OpenMP4 and OmpSs, tasks are declared in the code by the use of pragma annotations `#pragma omp depend(in : (..), out: (..), inout:(..))`. These pragma annotations precede the code blocks that are executed asynchronously and specify the list of memory locations that the tasks read `in`, write `out` or read and write `inout`. The runtime system takes this information and uses it to construct the TDG as detailed in Algorithm 1, The runtime system holds several data structures such as a list of reader tasks and the last writer task for every memory location expressed as dependences. In addition, tasks hold a list with their successors in the TDG and the count of their predecessors.

Task dependence analysis can be extended to track memory regions when accessing data with multiple dimensionality. In this case, the locations object tracks the dimensions accessed and the TDG creation algorithm uses this information to find the access overlaps within tasks [21].

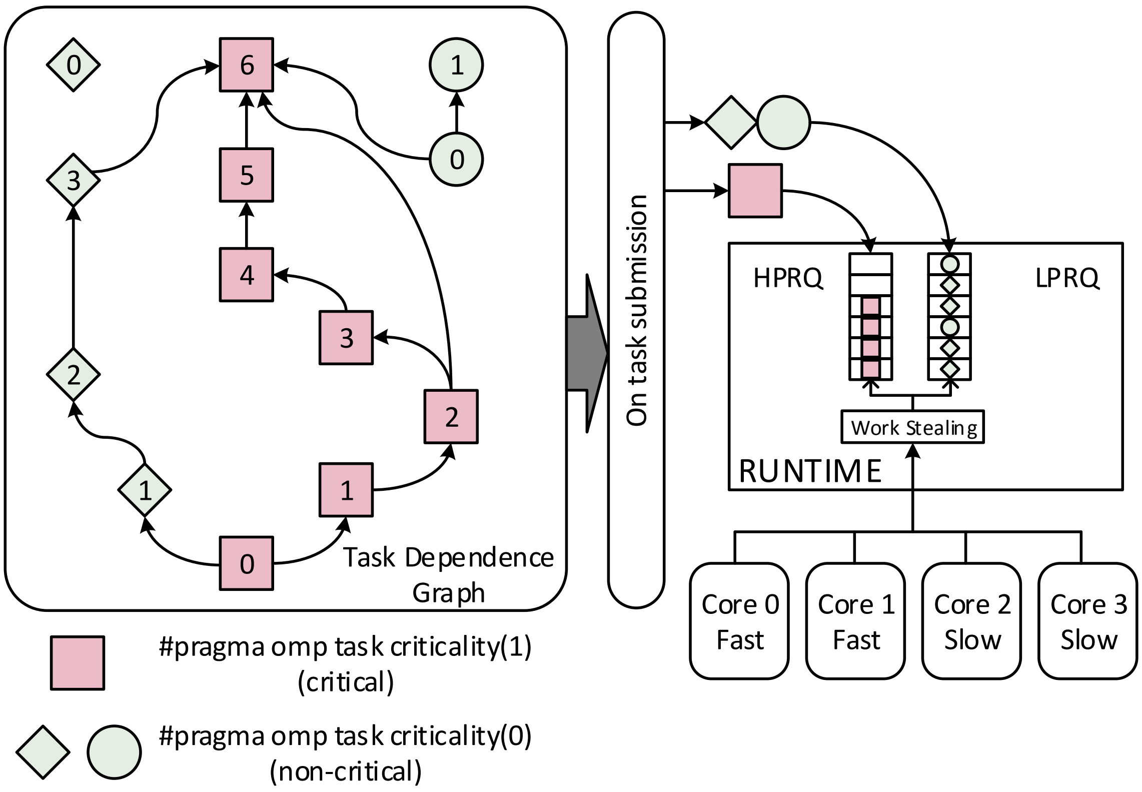


Figure 2.6: Criticality assignment with bottom-level and static policies, and Criticality-Aware Task Scheduling.

Once the TDG is constructed, the runtime system employs it to keep track of the execution. Furthermore, this TDG holds important information such as task criticality that can help runtime decissions like task scheduling.

Criticality represents the extent to which a particular task is in the critical path of a parallel application. In general, criticality is difficult to estimate as the execution flow of an application is dynamic and input-dependent. Two approaches can be used to estimate the criticality of a task.

One approach is to dynamically determine the criticality of the tasks during the execution of the application, exploring the TDG of tasks waiting for execution and assigning higher criticality to those tasks that belong to the longest dependence path [32]. Figure 2.6 shows a synthetic example of this method. In the TDG on the left, each node represents a task, each edge of the graph represents a dependence between two tasks, and the shape of the node represents its task type. The number inside each node is the *bottom-level* (BL) of the task, which is the length of the longest path in the dependence chains from this node to a leaf node. The criticality of a task is derived from its BL, where tasks with the highest BL and their descendants in the longest path are considered critical. Consequently, the square nodes in Figure 2.6 are considered critical.

The bottom-level approach does not require any input from the programmer and it can dynamically adapt to different phases of the application. However, this method has some limitations. First, exploring the TDG every time a task is created can become costly, specially in dense TDGs with short tasks. Second, the task execution time is not taken into account as only the length of the path to the leaf node is considered. Third, only a sub-graph of the TDG is considered to estimate criticality and some tasks marked as critical in such partial TDG may not be critical in the complete TDG.

Another approach is to statically assign criticality to the tasks, either using compiler analysis or allowing the programmer to annotate them. For this purpose, the task directive in OpenMP 4.0 can be extended to specify criticality, `#pragma omp task criticality(c)`. The parameter c represents the criticality level assigned to the given task type. Critical tasks have higher values of c, while non-critical tasks have a value of $c = 0$. The bottom left part of Figure 2.6 shows how this directive is used to assign the criticality of the three different task types in the example, where square nodes are considered critical, while triangular and circular nodes are estimated as non-critical. In this example and for the sake of simplicity, tasks are assigned to the same criticality level with both approaches (static annotations and bottom-level), but this does not happen in general.

The main problem of static annotations is that estimating the criticality of a task can be complex and input dependent. However, by analyzing the execution of the application it is feasible to identify tasks that block the execution of other tasks, or tasks with long execution times that could benefit from running in fast processing elements.

The task criticality information obtained with any of these approaches can be exploited by the runtime system in multiple ways, specially in the context of asymmetric or heterogeneous systems.

2.3.2 Task Scheduling

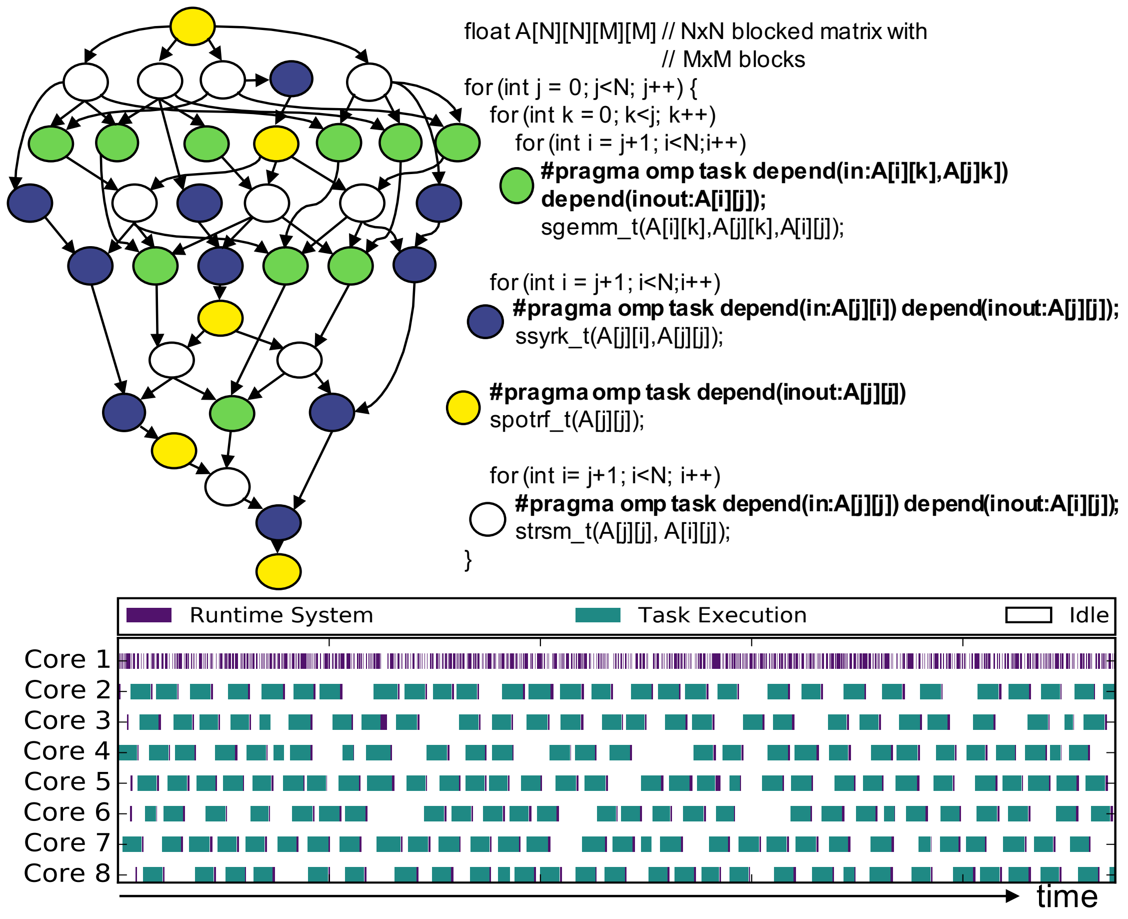
The task scheduler is a fundamental part of ATaP runtime systems. Its goal is to assign tasks to cores, maximizing the utilization of the available computational resources and ensuring load balance. The typical scheduler of ATaP runtime systems assigns tasks to available cores in a *first in, first out* (FIFO) manner without considering the criticality of the tasks. In this approach, tasks that become ready for execution are kept in a ready queue until there is an available core. There are several scheduling policies that can be applied in ATaP models, some of them FIFO variants such as *last in, first out* (LIFO) scheduler which allows to do an in-depth TDG traversal. Priority-based schedulers are other alternatives in which tasks are sorted in a ready queue according to their priority which can be based on aging or explicitly defined by the programmer.

However, since the TDG offers a rich degree of information, it is possible to deploy new scheduling mechanisms that exploit this knowledge in order to increase the system efficiency. The scheduler can apply graph partitioning techniques to the TDG to reduce expensive data transferences in NUMA systems by executing tasks reusing data in the same NUMA socket [109]. Criticality can also be used to guide scheduling decissions as in the *Criticality-Aware Task Scheduler* [32] (CATS), mainly focused on heterogeneous architectures, ensuring that critical tasks are executed on fast cores and assigning non-critical tasks to slow cores. As shown in Figure 2.6, CATS splits the ready queue in two: a *high priority ready queue* (HPRQ), and a *low priority ready queue* (LPRQ). Tasks identified as critical are queued in the HPRQ and non-critical ones in the LPRQ. When a fast core is available it requests a task to the HPRQ, and the first ready task is scheduled on the core. If the HPRQ is empty, a task from the LPRQ can be scheduled on a fast core. If no tasks are ready, the core remains idle until some ready task is available. Similarly, slow cores look for tasks in the LPRQ. Task stealing from the HPRQ is accepted only if no fast cores are idling. Figure 2.6 illustrates the runtime system extensions and the scheduling decisions for the synthetic TDG on the left.

2.3.3 Task Life Cycle

ATaP programming models use a decoupled execution model where tasks are created in program order and are executed asynchronously following the synchronization rules defined by the dependences. All threads may execute runtime system activity as well as tasks defined in the application source code. Figure 2.7 shows the execution timeline of the Cholesky benchmark on an 8-core system. In this experiment, core 1 performs most of the runtime system activities while the other cores mainly execute tasks.

The *master thread* executes the program sequentially and, when it encounters a task creation statement, it enters the *task creation* phase. The new task is assigned a unique *task descriptor* that stores all the relevant information of the task such as its dependences, its number of successors and a pointer to the function to be executed. The address of this task descriptor is used to identify the task. To detect dependences with older tasks, the inputs and outputs of the new and older task are compared. The new task is marked as a successor of older tasks if a RAW, WAR or WAW dependence is found, and is inserted in



```
float A[N][N][M][M] // NxN blocked matrix with
                    // MxM blocks
for (int j = 0; j<N; j++) {
  for (int k = 0; k<j; k++)
    for (int i = j+1; i<N;i++)
      #pragma omp task depend(in:A[i][k],A[j]k])
      depend(inout:A[i][j]);
      sgemm_t(A[i][k],A[j][k],A[i][j]);

  for (int i = j+1; i<N;i++)
    #pragma omp task depend(in:A[j][i]) depend(inout:A[j][j]);
    ssyrk_t(A[j][i],A[j][j]);

  #pragma omp task depend(inout:A[j][j])
  spotrf_t(A[j][j]);

  for (int i= j+1; i<N; i++)
    #pragma omp task depend(in:A[j][j]) depend(inout:A[i][j]);
    strsm_t(A[j][j], A[i][j]);
}
```

Figure 2.7: Cholesky task-based annotated code (right), TDG (left), and execution timeline (bottom).

the TDG accordingly as described by Algorithm 1.

The remaining *worker threads* iterate on the two main phases of the task-based dataflow execution model. The master thread also adopts this behavior when it reaches a global synchronization point.

- In the *task scheduling* phase the thread selects a task to be executed. The runtime system keeps a pool of ready tasks and selects one of them based on a scheduling algorithm. Different scheduling policies may adapt better to the characteristics of an application, and can provide significant benefits in certain contexts [31, 122].

- In the *task execution* phase the thread executes the code of the task that has been just scheduled. After the task is executed, the thread notifies the runtime system that the task has finished. The outputs of this task become available and its successor tasks may become ready if all its dependences are satisfied. In such case, it is added to the pool of ready tasks and will be selected for execution in future scheduling phases.

Apart from these phases, threads can experience *idle time*. In parallel regions idle time occurs when a thread enters the task scheduling phase and the pool of ready tasks is empty. This happens if the pace at which tasks are created is lower than the pace at which tasks are executed, or when threads reach a barrier. In addition, idle time happens in sequential parts of the program, where only one thread executes the sequential code and the other threads are waiting.

2.4 Architecture and ATaP Runtime Co-Design

2.4.1 Runtime System Hardware Support

Reducing the bottlenecks of ATaP runtime systems has been an extensive topic of research in the literature. Early works propose dataflow architectures like Monsoon [96], *T [89], or EARTH [53] including hardware support for dependence management and communication between instructions or threads. These architectures are programmed with specific-purpose programming models where the compiler statically generates the TDG and establishes the dependences between producers and consumers [6, 88]. Scheduling is either done statically at compile time using graph partitioning techniques or dynamically in hardware using a fixed FIFO queue.

There are recent contributions following the trend of architectures that provide hardware support for the complete task life-cycle. Task Superscalar [39] offloads all the runtime system activities to the architecture, including task dependence management and task scheduling with a fixed FIFO policy. Its hardware support consists of a gateway, a ready queue, and distributed tables to track tasks and dependences. Picos [118] is an actual implementation of Task Superscalar using FPGA devices. Swarm [60] relies on speculative task execution and conflict detection to preserve dependences. Swarm requires hardware support for speculation instead of for dependence management and uses either a priority-based scheduler using timestamps fixed in the architecture [59]. Espresso [61] is based on Swarm and allows to run speculative and non-speculative tasks concurrently ordered through timestamps.

Another trend is to provide hardware support for specific parts of the runtime system that can become a bottleneck. Carbon [71] implements the task scheduler at the hardware level and task dependence management is done in software by the runtime system. Carbon provides ISA instructions that allow threads to add and request ready tasks, and the hardware support consists of a set of distributed hardware queues to keep ready tasks and a fixed FIFO scheduling policy with work stealing. Similar to Carbon and Task Superscalar, other architectures use hardware task schedulers. These approaches rely on programmers or programming model semantics to establish dependences between tasks, so they do not offer hardware support for dependence management in ATaP programming models. GPUs use hardware schedulers for the kernels that can be synchronized with CUDA streams [90] or with queues and barrier packets in HSA [105]. In Pangaea [127], the CPU schedules tasks on the GPU, and both communicate via user-level interrupts.

Similarly to ADM [107], some works propose to add architectural support for thread synchronization primitives, reducing the overheads caused by concurrency. CAF [126] provides hardware support to optimize core-to-core queue-based communications, adding a specialized accelerator that supports various queue management functionalities. An implementation for lock primitives based on distributed queues is proposed in QOLB [65] where the waiting cores spin locally, preventing unnecessary network traffic. Active Memory Operations [41] extend the memory controllers of distributed shared-memory systems so that synchronization and heavy write sharing operations can be executed in the node where the data resides. These solutions allow implementing different scheduling policies in software with reduced hardware complexity, but they do not accelerate all the operations of the dependence management and task scheduling phases, so they are less effective in mitigating runtime system overheads.

Another way to mitigate the task creation bottleneck is parallelizing it with nested parallelism. Although most parallel programming models support nesting, the practical usage of this paradigm requires a hierarchical decomposition of the algorithm and limits the visibility of dependences across different nesting levels. Perez et al. [98] tackle this issue by adding specific notations to expose dependences across different levels and allows the runtime system to uncover hidden parallelism. Fractal [116] extends Swarm to allow nested parallelism by means of task domains, which can be ordered or unordered to avoid over-serialization. Due to the difficulty of expressing nested parallelism, it is much more appropriate to alleviate the task creation bottleneck via hardware support as this thesis proposes instead of transferring this responsibility to the software stack.

2.4.2 Exploiting Runtime System Information In the Architecture

Exploiting the algorithmic information available in the runtime system to drive low-level architectural decisions has been another prolific research topic. Some of the most relevant works target how to improve the memory hierarchy performance or resilience through the knowledge available at the runtime system. The input and output information allows the runtime system to transparently manage systems with multiple address spaces, such as GPUs [7, 99], multi-node clusters [21], heterogeneous memory systems [3, 75], and scratchpad memories [4, 12]. RADAR [79] uses the memory addresses of task dependences to predict dead blocks in the last-level cache and evict them, while Pan et al. [95] uses similar information to guide the partitioning of last-level cache. Caheny et al. [23, 25] aim to reduce coherence traffic movement in NUMA systems by combining NUMA aware scheduling and data allocation. The same authors also propose to deactivate coherence for non-shared data as specified by the runtime system [24]. Sanchez et al. [109] apply graph partitioning techniques to the TDG in order to reduce data transferences in NUMA systems. Dimic et al. [35] reduce the miss ratio of the last level cache with a runtime aware replacement policy. Similarly, Papaefstathiou et al. [97] propose a prefetcher and a replacement policy guided by the task lifetime that is able to distinguish between data from different tasks. Manivannan et al. [78, 80] study the NoC utilization of dataflow programming models and how optimizations to producer-consumer communication patterns can be applied. Finally, Jaulmes et al. [58] use the implicit redundancy in iterative solvers to asynchronously release tasks in ATaP models to recover partially corrupted data on memory failures.

Other relevant contributions have explicitly targeted how to improve performance and power consumption. Brumar et al. [20] use the dependences information to trigger value prediction and automatic memoization for OmpSs applications. Chasapis et al. [27] explore how the manufacturing variability and power-constraints lead to heterogeneous performance that can be controlled by the runtime system scheduling policies in order to minimize the performance degradation. Chronaki et al. [31] study how to efficiently assign task to cores in asymmetric multiprocessor systems by exploiting the critical path of the TDG. Moreover, the same authors target the bottleneck of task creation and dependence analysis by offloading the TDG construction to an already existing specialized hardware and studying how the runtime controls this hardware [30]. LibPRISM [93] is used to autotune the prefetcher policy and SMT levels in OpenMP parallel applications by profiling online the parallel regions or task characteristics..

Finally, the unique characteristics of the ATaP models such as the presence of a TDG defining the execution, or the explicit annotations of the memory regions that each task access to have motivated the development of profiling and simulation tools to provide insight of the behavior of ATaP models in large-scale systems. The TDG can be used as the input for complex multicore system simulators such as TaskSim [104], which is able to scale the execution to thousands of simulated cores. In addition to TaskSim, MUSA [44] is a multiscale simulator for distributed systems using execution-driven simulators for the shared memory parts of the workload and trace-driven simulation for the distributed nodes.

Chapter 3

Experimental Framework and Toolset

3.1 Simulation Infrastructure

The hardware extensions proposed in this thesis have been modelled by using the gem5 simulator [16]. Gem5 is a execution-driven multi-core full system simulator that can do a cycle accurate execution of a complete operating system. Gem5 supports various ISAs with different CPU and memory models ranging from pure functional ones to highly detailed and cycle accurate.

Gem5 adds support for checkpointing and KVM Emulation [110] to accelerate system and benchmark initialization using less detailed CPU and memory models. In this thesis we employ the checkpointing capabilities so that simulations start right at the parallel sections of the benchmarks.

The experiments in this thesis have been done using two different configurations as listed in Table 3.1. Chapter 4 uses the x86-64 configuration, while Chapter 5 relies on the ARMv8 experimental setting. The ARM architecture lacked of support for running more than 8 CPU cores in the early versions of the simulator employed in the thesis. In 2016, ARM made an effort to improve their support to the simulator and this motivated the architecture switch as the x86-64 ISA lacks from any official support. All the experiments are run with the most detailed configurations available for each architecture trying to resemble a real system. The gem5 simulator has been extended with a module called the Runtime Support Unit (RSU) that will be detailed in Chapters 4 and 5. A *cpufreq* driver to manage the gem5 DVFS controller as in real system was developed and added to the x86-64 kernel. This driver allows to change frecuency of a given core from userspace and enables the experiments of Chapter 4.

During the development of this thesis, several corrections were made to the simulator in order to be able to carry out the experiments:

- Several errors in the MESI coherence protocol when using the x86 locked read modify write operations. Eviction of locked lines, and incorrect protocol transient states were the major issues.
- Interrupt clobbering in the out-of-order pipeline.
- Corrections to the memory consistency model where remote invalidations where not propagated to the CPU Load Store Queue (LSQ).
- Vectorial registers were not saved in context switches due to a missing interruption triggered by the APIC.
- Clock-sources not correctly attached to the L1-cache, greatly increasing access time.
- Clock-sources not correctly synchronized between cores.

Many of these issues have been corrected or reported in the recent gem5 code. Power and area estimations for the multi-core and added hardware structures are evaluated using McPAT 1.3 [74], a power and area modelling tool built on top of CACTI 6.0 [87]. McPAT offers models that range from low-power configurations using in-order cores and low voltage designs, to high performance processors based on aggressive out-of-order configurations. McPAT also models the cache and interconnection network power consumption. For this thesis, the 22nm technology node with the default clock-gating schema is employed.

3.2 HPC Cluster

We use the Marenostrum 4 supercomputer at the Barcelona Supercomputing Center for running our experiments on real machines. Marenostrum consists of 3,456 compute nodes; every node has two Intel Xeon Platinum 8160 processors each with 24 cores and 96 GB of DDR4-2667 main memory. The interconnection network is a 100 Gb Intel OmniPath full bisection fat-tree. The software stack comprises SUSE Linux Enterprise Server 12 SP2 with kernel version 4.4.120-92.70 and includes MVAPICH 2.2 running on top of Intel PSM2 with our modifications.

Table 3.1: Processor Configuration.

Chip Details		
Core count Core type	32 Out-of-order single threaded	
ISA	x86-64	ARMv8
Core Details		
DVFS configurations	Fast cores: 2 GHz, 1.0 V Slow cores: 1 GHz, 0.8 V $25\mu s$ reconfiguration lat.	2GHz
Fetch, issue, commit bandwidth	4 instr/cycle	
Branch predictor	4K selector, 4K G-share 4K bimodal 4-way BTB 4K entries RAS 32 entries	Tournament 2K local pred. 8K global and choice pred. 4-way BTB 4k entries RAS 16 entries
Issue queue	Unified 64 entries	
Reorder buffer	128 entries	
Register file	256 INT, 256 FP	
Functional units	4 INT ALU (1 cyc), 2 mult (3 cyc), 2 div (20 cyc) 2 FP ALU (2 cyc), 2 mult (4 cyc), 2 div (12 cyc) 2 Ld/St unit (1 cyc)	
Instruction L1	32KB, 2-way, 64B/line (2 cycles hit)	
Data L1	64KB, 2-way 64B/line (2 cycles hit)	32KB, 2 way 64B/line (2 cycles hit)
Instruction TLB	256 entries fully-associative (1 cycle hit)	
Data TLB	256 entries fully-associative (1 cycle hit)	
NoC and shared components		
L2	Unified shared NUCA banked 2MB/core, 8-way 64B/line 15/300 cycles hit/miss	Shared L2 cache 4MB 16-way 64B/line
Coherence protocol	MESI 4-way cache directory 64K entries	MOESI Fast Atomic Snooping
NoC	4×8 Mesh, link 1 cycle	gem5 VExpress
Software Stack		
Operating System Kernel	Gentoo 2.6.28-4	Ubuntu 14.04 4.3
ATaP model runtime system	Nanos++ v.07a	Nanos++ v0.10a

We use 16, 32, 64 and 128 nodes of the cluster. For all benchmarks, 4 MPI processes are spawned per node, each of which creates 8 worker threads.

3.2.1 Workload Management

The evaluation experiments carried out during this thesis required thousands of executions of individual experiments in a iterative process of continuous feedback. All these executions are based on combinations of different parameters feed to an executable by either configuration files or command line. The combinatorial explosion resulting from these configurations requires the use of tools for managing all the input parameters and results. While common shell scripts are used, the need of creating or adapting scripts when using different environments or programs is still present.

For this thesis, we abstract the characteristics of such workloads and create a software called Tizona [26]. Tizona relies on platform independent JSON configuration files that are able to launch all the individual executions of an experiment. The file specifies the possible values for the parameters and the software creates all the independent executions to cover the paramete search space. Once experiments are done, results can be retrieved in CSV files that can be filtered by specifying parameter or execution produced values. Tizona has support for GridEngine and SLURM workload managers and is able to run experiments completely out of the box in several supercomputers. Moreover, Tizona is opensource and available at github [26].

3.3 Softare Stack

3.3.1 Operating System and Build Toolchain

All the simulations are performed under a realistic software environment. A complete Linux kernel is used for both the x86-64 kernel and ARMv8 configurations with versions 2.6.28-4 and 4.3 respectively. The x86 kernel has been modified to support synchronized clock sources between cores operating at different frequencies in order to avoid desynchronizations when processes are migrated from one core to another. Moreover, we have developed a cpufreq framework [94] driver to interface with the gem5 DVFS controller and integrated it into the x86-64 kernel to do all the experiments of Chapter 4. The ARM kernel has been compiled with the gem5 support extensions activated. Otherwise it is not possible to run simulations with more than 8 cores. The Operating System is a complete

Gentoo Linux built from scratch using a Stage-2 tarball for the x86-64 architecture, and Ubuntu 14.04 for ARM. We employ the default set of system libraries and the compiler used to compile the ATaP runtime system and the environment is gcc 4.6.4 and Mercurium 1.99 [8] with gcc as the backend for the benchmarks.

The Chapter 6 executions are done in a real environment using the software stack described in Section 3.2

3.3.2 ATaP Model Runtime System

Nanos++ is selected as the task-based runtime system for Chapters 4, 5 and 6. Nanos++ is a runtime system compatible with the OpenMP 4.0 task semantics and the additional OmpSs constructs. Nanos++ supports different architectures such as SMPs, hybrid GPU systems or Cell Synergistic Processing Units (SPEs) through a plugin based interface. The runtime system consits of a core in which abstract worker elements ask for tasks to be executed by executing a worker idle loop which invokes the scheduler plugin. The worker specific implementations are defined in the architecture plugins and Nanos++ offers a plugin system as well to support different scheduling algorithms. Moreover, different dependence analysis algoritms such as plain dependences, or regions are abstracted as plugins. This thesis provides plugins to implement the scheduling strategies defined in Chapters 4 and 5. A dependence analysis plugin to control the hardware dependence module described in Chapter 5 is also developed so the runtime core functionality remains unchanged.

We developed a stripped version of the Nanos++ runtime for Chapter 6 that allows the programmer to explicitly define the TDG and uses lock-free structures for task management structures. This runtime implementation is devised to seamlessly integrate with the MPI runtime in order to implement and evaluate all the mechanisms proposed.

3.3.3 MPI & PSM2

The selected base MPI implementation is MVAPICH 2.2 [121]. MVAPICH is an open-source implementation of MPI that delivers the best peformance for systems using the Infiniband interconnection technology as the one used in this thesis. MVAPICH offers complete support of the MPI 3.0 standard and adds multiple additional extensions targetted to PGAS, GPU and Intel MIC interoperability. Finally, MVAPICH is well integrated with Intel PSM2 library.

This thesis relies on the PSM2-CH3 interface that provides support for the Intel OmniPath interconnection technology. PSM2 [55] offers a matched queue point-to-point message passing library built directly on top of the OmniPath NIC driver. PSM2 is highly optimized for both inter-node and intra-node communication patterns offering different optimized algorithms based on the size of the messages. MVAPICH directly relies on PSM2 for tag-matching and point-to-point message sending and delivery, providing just a layer of the MPI capabilities such as communicators, datatypes or collective communications.

3.4 Performance Analysis Tools

Debugging parallel workloads running on a full system simulator is an added difficulty to the development process. Simulators usually do not provide any feedback on the parallel execution of an application, and the typical lenght of simulations (up to one week) makes interactive debugging not feasible. Moreover, errors in the simulator such as an incorrect coherence protocol can leave the running application in a frozen state without any clue of when the error happened. The only solution is to add print statements to determine a timestamp of the error and then obtain traces with the architectural state from the simulator.

In this thesis, we propose a new approach to debug the architectural status of the simulator and the application by using Paraver traces [10]. We have created an interface to produce traces where the simulator can directly output architectural status such as ROB occupancy, coherence invalidations, NoC packet latency, among many other statistics. The application can use this interface through an ISA extension that outputs an event and a value. In this fashion, Paraver traces are enrichened and allow the architectural designer to inspect reorder-buffer occupancy per task type, the number of branch misspredictions per task, ACPI power status through time, or any other combination of architectural and application state considered relevant for profiling or debugging. With our infrastructure, we can report hundreds of microarchitectural events.

Another advantage of this approach is that the use of a simulator allows to produce traces without altering the simulated application timing. Real hardware executions need to use tools such as Extrae [9], which end up altering the application execution time and polluting cache state. In a simulated system, the application only needs to execute a special instruction that emits the traceable event, and is in the simulator layer where all the heavy operations (such as memory traversal and output flushing) happen, almost eliminating the overhead of the tracing process.

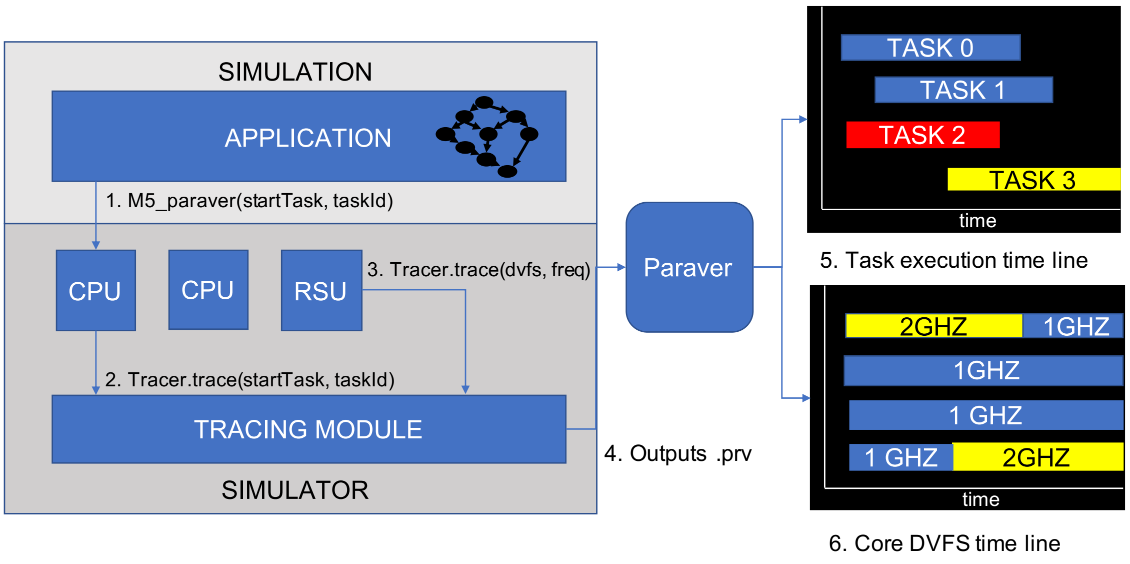


Figure 3.1: Paraver tracing facilities in the gem5 simulator.

Figure 3.1 shows the architecture of the approach, the application executes the instruction to emit an event. Then, the CPU object in the simulator translates that instruction into a call to the simulator tracer module. Later, the simulator detects an architectural change (DVFS) and reports it to the tracer. The resultant trace contains the task execution time line and the core DVFS status and allows to gather statistics on the number of accelerated tasks, etc.

Figure 3.2 shows the execution of Ferret from the PARSECSs suite on an asymmetric multi-core architecture with 4 out-of-order cores and 4 in-order cores. The task execution time line and the associated IPC per tasks can be seen in the upper and lower part of the figure. The upper part of the figure shows the task execution timeline. Each task shows a different color depending on their task type, and idle time is represented in a light blue color. The lower part is the IPC for the associated core during exeuction. The higher the IPC the darker the color becomes, being brilliant green a low IPC less than 0.5 and dark blue an IPC greater than 1. This visualization allowed us to debug some errors in the memory model that were lowering the IPC of certain tasks in the out-of-order cores while the final IPC reported by the simulator was high due to the idle loop of the runtime system.

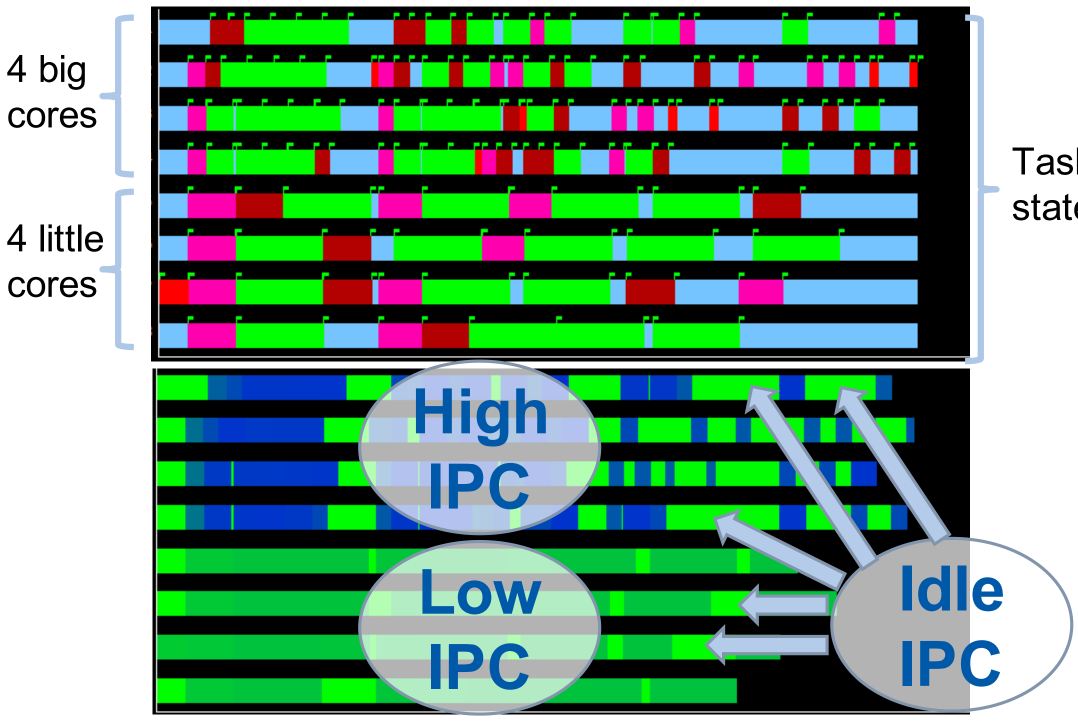


Figure 3.2: Paraver trace of an asymmetric multicore with 4 big and 4 little cores running Ferret. Upper part shows the task execution timeline and Lower part shows the IPC of each core for that task.

3.5 Benchmarks

3.5.1 Shared Memory Workloads

The following benchmarks are used to evaluate performance in the shared memory proposals of this thesis. Seven benchmarks of the PARSEC [13] suite are used in the evaluation. These benchmarks are representative state-of-the-art parallel algorithms from different areas of computing that use two well-known parallelization approaches. We use the task-based parallel implementations proposed by Chasapis et al. [28]

We evaluate up to four benchmarks using fork-join parallelism: Blackscholes solves the Black-scholes Partial Differential Equation to calculate the prices for a portfolio of European options. The work is divided in blocks with the blocksize being a parameter and each task is assigned a block, this favours load balancing as the number of tasks can be greater than the number of threads. In this benchmark, dependences are only between timesteps: tasks depend on a task doing the previous timestep of the block and there are no dependences within tasks of different blocks, making this benchmark embarrasingly parallel.

Fluidanimate simulates the incompressible fluid interactive animation using the Smoothed Particle Hydrodynamics method. Fluidanimate operates on a 3D volume partitioned along the X and Z axis with an undefined number of particles per block. Five special kernels are taskified and executed on every block to rebuild the spatial index, compute the fluid densities and forces, handle collisions with the scene geometry and update the particle locations. Tasks depend on the tasks previously executed on the same block and its neighbors, making it a 9-point stencil.

Swaptions uses the Heath-Jarrow-Morton (HJM) method for pricing derivatives with Montecarlo simulations. Data is stored in arrays that are partitioned with a single task doing the calculations on portions of the array. There are no data dependences between the tasks.

We use thre benchmarks using pipeline parallelism: Streamcluster is a kernel to solve the online clustering problem that groups a stream of points in a predefined number of clusters or partitions. The task based implementation keeps the number of tasks independent of the number of partitions, with them processing chunks of the input stream. In such schema, barriers are needed to synchronize the updates of partitions.

Bodytrack is a computer vision application to track the human body without markers. Several cameras are used through an image sequence and an annealed particle filter is used with the edges and foreground silhouette as the main features. All the frames are analyzed in parallel by using coarse-grain tasks that spawn nested tasks to do the particle filter updates. Once that calculation for a frame finishes, the output writing is also taskified in order to exploit computation and I/O overlap.

Dedup relies on the deduplication method [101] to compress a data stream using local and global compression. Dedup parallelization is a pipeline of tasks with 4 tasks doing computation (Fragment, Refine, Deduplication, Compress) and a task that reorders the fragments and writes them to a file. Data is divided in chunks and computation tasks can run in parallel for different chunks. The reorder and output writing tasks impose an order that serializes execution ensuring that chunk *N-1* will be written before chunk *N*.

Ferret is a content similarity search application that focuses on images. The parallelization is similar to Dedup as for every image a pipeline of 5 tasks performing computation is spawned with a final task to output the results. The last task imposes a serialization and reorder of the output writing in the same fashion as Dedup.

In addition to the PARSECSs benchmark suite, some popular and extensively used linear algebra kernels are also employed due to their relevance in HPC and their extensive use of dependences. The task-based implementation of these algorithms applies tiling so that tasks process 2D blocks of the matrices. The Cholesky factorization decomposes a hermitian definite positive matrix A in the product of two matrices $L \cdot L^*$ where L is a lower triangular matrix and L^* its the L conjugate transpose. Tasks are in charge of executing LAPACK routines on different tiles of the matrix. Histogram computes a cumulative histogram for all pixels of an image using a cross-weave scan [100]. Multiple calculation tasks acting on individual blocks for different images are overlapped, and reduction task are executed later based on vertical and horizontal halos obtained by the computation tasks. The QR factorization of a matrix is a product $A = QR$ with Q orthogonal and R upper triangular. The multiple tasks implementation relies on LAPACK as cholesky. LU does a $A = L \cdot U$ decomposition of a matrix with L being a lower diagonal matrix and U being upper diagonal. As cholesky and QR tasks executes BLAS/LAPACK routines operating on different tiles.

The input sets for the benchmarks are described in table 3.2

3.5.2 Distributed workloads

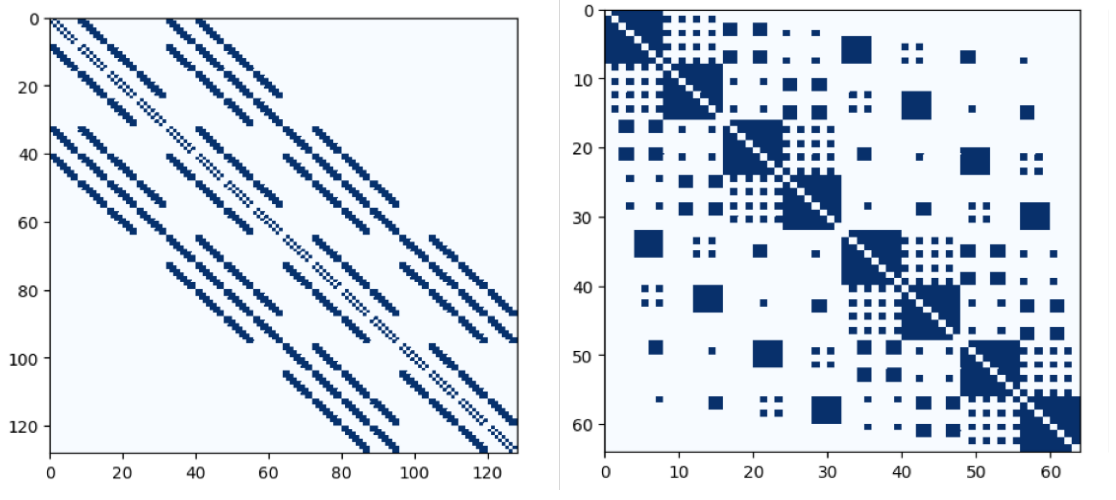
3.5.2.1 Point-to-point Benchmarks

We have implemented two stencil-based benchmarks using task semantics. The first benchmark is based on HPCG [37], a multi-grid Conjugate Gradient solver with a Gauss-Seidel preconditioner. HPCG uses a 27-point stencil where every block performs a total of 11 halo-exchanges with its neighbors in each iteration due to the preconditioning step. In addition, an `MPI_Allreduce` collective operation is performed at the end of each iteration. The resulting communication pattern of HPCG is shown in Figure 3.3a, where darker colors display a larger communications volume between two processes. The `MPI_Allreduce` pattern is represented by a light background color as it just involves communication of a scalar value among all the nodes. In our experiments, we apply weak scaling and solve global problem sizes of $1024 \times 512 \times 512$, $1024 \times 1024 \times 512$, $1024 \times 1024 \times 1024$ and $2048 \times 1024 \times 1024$ on 64, 128, 256, 512 MPI processes respectively.

The second benchmark is based on MiniFE, a finite element solver using a non-preconditioned Conjugate Gradient. In contrast to HPCG, MiniFE only performs a single halo exchange per iteration and has a more irregular communication pattern between pro-

Table 3.2: Input Sets for Shared Memory Benchmarks

Benchmark	Input Size
Shared Memory Benchmarks	
Blackscholes	Simlarge
Cholesky	2048x2048 dense matrix
Dedup	Simlarge
Ferret	Simlarge
Fluidanimate	Simlarge
Histogram	4096x4096 image
LU	2048x2048 sparse matrix
QR	1024x1024 dense matrix
Streamcluster	Simlarge
Distributed Memory Benchmarks	
HPCG	$1024 \times 512 \times 512$ volume
	$1024 \times 1024 \times 512$ volume
	$1024 \times 1024 \times 1024$ volume
	$2048 \times 1024 \times 1024$ volume
MiniFE	$1024 \times 512 \times 512$ volume
	$1024 \times 1024 \times 512$ volume
	$1024 \times 1024 \times 1024$ volume
	$2048 \times 1024 \times 1024$ volume
FFT-2D	$16384^2, 32768^2, 65536^2, 131072^2$, and 262144^2 elements
FFT-3D	$1024^3, 2048^3$, and 4096^3 elements
Map-Reduce WordCount	$262 * 10^6, 524 * 10^6$, and $1048 * 10^6$ words
Map-Reduce MatrixVector	$1024^2, 2048^2$, and 4096^2 matrix



(a) HPCG (b) MiniFE

Figure 3.3: Communication patterns of HPCG and MiniFE. Dark colors indicate volume of communication between MPI processes, while white indicates absence of communication.

cesses, as shown in Figure 3.3b. The lack of a preconditioner step in every iteration reduces the granularity of the computation tasks, thus providing insights on how the proposed mechanisms behave in fine-grain task environments. Similar to HPCG, each iteration of MiniFE also ends with an `MPI_Allreduce`. As input, we use $1024 \times 512 \times 512$, $1024 \times 1024 \times 512$, $1024 \times 1024 \times 1024$ and $2048 \times 1024 \times 1024$ unstructured implicit finite volumes.

In both benchmarks, each processor is assigned a sub-block of the initial 3D domain. Each sub-block maps to a set of rows in the sparse matrix to be solved by the conjugate gradient step. In order to effectively overlap communication and computation in an execution driven by our stripped version of the Nanos++ runtime system, the sub-block assigned to a processor is further overdecomposed into smaller sub-blocks. We consider decomposition factors between $1\times$ (one sub-block per core) and $16\times$ (16 sub-blocks per core), and report runtime for the best performing decomposition for every configuration.

3.5.2.2 Benchmarks with Collective Communications

To evaluate performance with collective communications, we have implemented several benchmarks. The first benchmark is a two-dimensional (2D) FFT using a parallel zero-copy algorithm [49]. In 2D FFT, we initially divide the matrix among MPI processes using row-wise 1D block partitioning. This enables creation of tasks for executing 1D

FFTs for each row in parallel. Next, we perform an `MPI_Alltoall` to transpose the matrix. Finally, 1D FFTs are calculated again for each row of the transposed matrix. The matrix is transposed during communication by using MPI derived datatypes, as described by Hoefler et al. [49]. In order to avoid multiple copies of data, the strides and MPI extent within rows are specified in two derived datatypes, one for sending and another for receiving, so the MPI implementation is able to transpose the matrix on-the-fly.

When transposing the matrix using the `MPI_Alltoall` collective with derived datatypes, each process receives partial row data from every other process. Typically, it is not possible to overlap the collective with the computation tasks because tasks computing the 1D FFT require the entire matrix row. However, it is possible to further divide the 1D FFT into smaller tasks that process data blocks as soon as they are received. The block size is set to be the size of a row divided by the number of MPI processes, allowing the execution of partial 1D FFT tasks as the `MPI_Alltoall` progresses. We evaluate the performance of 2D FFT for square matrices with $16384^2, 32768^2, 65536^2, 131072^2$, and 262144^2 elements.

The second benchmark is a three-dimensional (3D) FFT. Initially, the 3D volume is divided into subsets created by 2D decomposition in Y and Z dimensions. 1D FFT computations are performed along the x-axis, and are followed by `MPI_Alltoall` calls within subcommunicators defined along the y-axis. This transposes the volume such that the subsets are now decomposed in the X and Z dimensions, and 1D FFTs along the y-axis are performed. Next, `MPI_Alltoall` calls within the subcommunicators defined along the z-axis transposes the grid to create the final set of subdomains in which 1D FFT can be performed along the Z dimension. Thus, while 2D FFT requires one `MPI_Alltoall`, the 3D version needs two `MPI_Alltoall` calls in order to rotate the volume. We have chosen a 2D decomposition over a 1D decomposition because of its better scalability in terms of memory and communication [111]. For 3D FFT, we test cubic volumes with $1024^3, 2048^3$, and 4096^3 elements.

We also evaluate performance for two MapReduce [34] applications — a simple word-count algorithm, which counts the occurrence of each word in a text, and a dense matrix vector product. In MapReduce, the input data is split into independent chunks processed by the map tasks in parallel. Each map task produces a series of tuples in the form (Key, Value) (K, V). The values $V_{0..N-1}$ associated to the same K_i are coalesced in a list and each process sends its $(K_i, V_{0..N-1})$ tuples to another process determined by a function of the key $Node_{id} = hash(K_i)$ in the shuffling stage. Shuffling is done by using the

`MPI_Alltoallv` collective as a process may have different number of keys that map to other processes. Finally, every process applies the reduction operation to the list of values $(V_{0..N-1})$ associated with each key; the reductions for different keys can be done in parallel.

We have implemented a baseline MapReduce framework that uses `MPI_Alltoallv` for data shuffling in *OmpSs* and MPI. In the baseline MapReduce implementation, the reduction of a single key list of values is a serial operation, while reduction for different keys can be performed in parallel. However, using our scheme, the reduction tasks can start to execute as soon as the `MPI_Alltoallv` receives data from any single rank. This leads to the creation of several parallel reduction tasks for the same key as multiple list of values for a single key might be received from different processes.

Chapter 4

Improving Power Consumption Through Task Criticality

This work advocates an integrated system in which the task-based runtime system controls hardware reconfiguration according to the criticality of the different tasks in execution. As such, the runtime can either schedule the most critical tasks to the fastest hardware components or reconfigure those elements where the highly-critical tasks run. In this way, the programmer only has to provide simple and intuitive annotations and does not need to explicitly control the way the load is balanced, how the hardware is reconfigured, or whether a particular power budget is met. Such responsibilities are mainly left to the runtime system, which decouples the software and hardware layers and drives the design of specific hardware components to support such functions when required. To reconfigure the computation power of the system, we consider DVFS, as it is a common reconfiguration capability on commodity hardware. However, our criticality aware approach can target reconfigurations of any hardware component, as no DVFS specific assumptions are made.

The most relevant contributions of this Chapter are:

- We compare two mechanisms for estimating task criticality with user-defined static annotations and with a dynamic solution operating at execution time. Both approaches are effective, but the simpler implementation of the user-defined static annotations provides slightly better performance and EDP results.

- We introduce *Criticality Aware Task Acceleration* (CATA), a runtime system level technique that reconfigures the frequency of the cores while keeping the whole processor under a certain power budget. Average improvements reach 18.4% and

30.1% in execution time and EDP respectively, over a baseline scheduler on a simulated 32-core system.

- For some applications, the DVFS reconfiguration penalties caused by inherent serialization issues can become a performance bottleneck. To overcome this problem, we introduce a hardware component denoted *Runtime Support Unit* (RSU), which relieves the runtime system of carrying out frequency reconfigurations and can be easily incorporated on top of existing solutions [5, 54, 76]. For sensitive applications, up to an additional 8.5% improvement in performance is obtained over CATA.

4.1 Limitations of scheduling algorithms

Task scheduling is a critical phase in the runtime system of ATaP programming models since the scheduling algorithm is in charge of mapping the actual tasks to the processing elements. In asymmetric systems where processing elements exhibit different performance rations, a bad decission at schedule time in such systems known as *blind assignment* can decrease performance as shown in [32].

CATS or *Criticality-Aware Task Scheduler* solves the blind assignment problem of FIFO schedulers by employing the task criticality on scheduling decissions mapping the most critical tasks to the processing elements with the highest performance ratio when possible. However, even if it considers the criticality of the tasks, it may present the following misbehaviors in the scheduling decisions that lead to load imbalance in heterogeneous architectures:

- *Priority inversion*: when a critical task has to be scheduled and all the fast cores are in use by non-critical tasks, it is scheduled to a slow core.
- *Static binding* for the task duration: when a task finishes executing on a fast core, this core can be left idle even if other critical tasks are running on slow cores.

These problems happen because the computational capabilities of the cores are static and, once a task is scheduled to a core, it is not possible to re-distribute resources if the original circumstances change. In order to overcome these limitations, this thesis proposes a runtime-driven *criticality-aware task acceleration* scheme, resulting in a responsive system that executes critical tasks on fast cores and re-distributes the computational capabilities of the architecture to overcome the priority inversion and static binding problems.

To reconfigure the computation power of the system, we consider DVFS, as it is a common reconfiguration capability on commodity hardware. However, our criticality aware approach can target reconfiguration of any hardware component, as no DVFS specific assumptions are made.

4.2 Criticality-Aware Task Acceleration Using DVFS Reconfigurations

This section proposes to exploit reconfiguration opportunities that task criticality information can provide to the runtime system to perform *Criticality-Aware Task Acceleration* (CATA) in ATaP models. First, a pure software approach where the runtime system drives the reconfigurations according to the criticality of the running tasks is introduced. Then, hardware extensions required to support fast reconfigurations from the runtime system are described in detail.

DVFS is selected as a proof-of-concept for the reconfiguration mechanism, as it allows to accelerate different cores and it is already present in the majority of current architectures. Nevertheless, the proposed ideas and the runtime system extensions are generic enough to be applied or easily adapted to other reconfiguration techniques. We further assume that two frequency levels are allowed in the system, which can be efficiently implemented with dual-rail V_{dd} circuitry [84]. Extending the proposed ideas to more levels of acceleration is left as future work. In addition, the background Chapter 2 discusses other reconfiguration approaches that could benefit from the ideas proposed in this chapter.

4.2.1 Criticality-Aware Runtime-Driven DVFS Reconfiguration

The runtime system is extended with several structures to manage hardware reconfiguration according to the criticality of the tasks. Figure 4.1 shows these extensions. Similar to the CATS scheduler, the runtime system splits the ready queue in a HPRQ for the critical tasks and a LPRQ for the non-critical tasks. To manage the reconfigurations, the Reconfiguration Support Module (RSM) tracks the state of each core (*Accelerated* or *Non-Accelerated*), the criticality of the task that is being executed on each core (*Critical*,

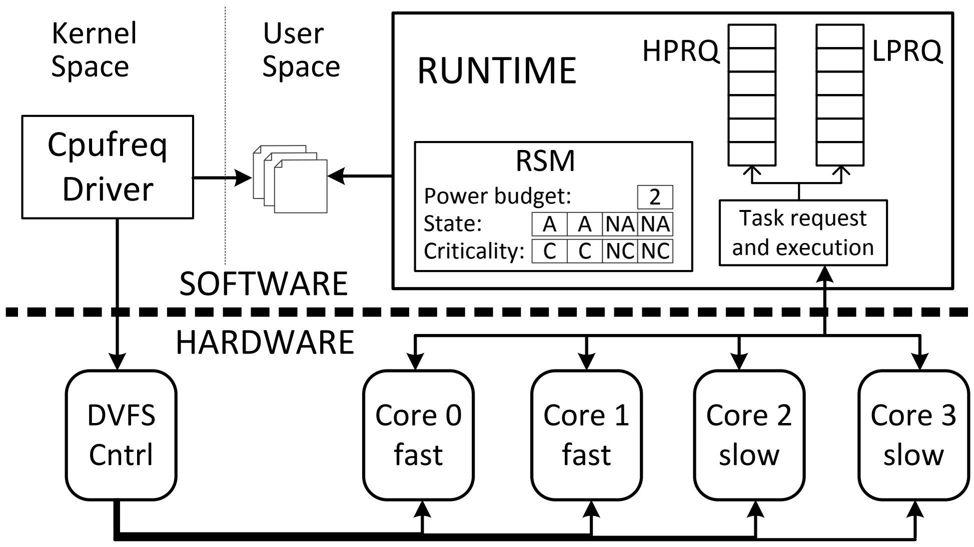


Figure 4.1: Runtime system support for CATA using DVFS reconfigurations. The runtime maintains status (Accelerated, Not Accelerated) and criticality (Critical, Non-Critical, No Task) information for each core in the RSM.

Non-Critical, or *No Task*), and the power budget. The power budget is represented as the maximum amount of cores that can simultaneously run at the fastest frequency, and is provided to the runtime system as a parameter.

When a core requests a new task to the scheduler it first tries to assign critical tasks from the HPRQ and, if no critical tasks are ready, a non-critical task from the LPRQ is selected. If there is enough power budget the core is set to the fastest power state, even for non-critical tasks. If there is no available power budget and the task is critical, the runtime system looks for an accelerated core executing a non-critical task, decreases its frequency, and accelerates the core of the new task. In the case that all fast cores are running critical tasks, the incoming task cannot be accelerated, so it is tagged as non-accelerated. Every time an accelerated task finishes, the runtime system decelerates the core and, if there is any non-accelerated critical task, one of them is accelerated.

To drive CPU frequency and voltage changes, the runtime system uses the standard interface provided by the `cpufreq` daemon of the Linux kernel. The `cpufreq` daemon governor is set to accept changes from user space. Figure 4.1 shows how the runtime system communicates with the `cpufreq` framework. Frequency and voltage changes are performed by writing the new power state in a configuration file mapped in the file system, having one file per core. The `cpufreq` daemon triggers an interrupt when it detects a write to one of these files, and the kernel executes the `cpufreq` driver. The driver writes the new power state in the DVFS controller, establishing the new voltage and frequency values for the core, and then the architecture starts the DVFS transition.

Finally, the kernel updates all its internal data structures related to the clock frequency and returns the control to the runtime system.

Although this approach is able to solve the priority inversion and static binding issues by reconfiguring the computational capabilities assigned to the tasks, it raises a new issue for performance: *reconfiguration serialization*. Some steps of the software-driven reconfiguration operations inherently need to execute sequentially, since concurrent updates could transiently set the system in an illegal state that exceeds the power budget. Furthermore, invoking an interrupt and running the corresponding `cpufreq` driver in the kernel space can become a performance bottleneck. As a result, all the steps required to reconfigure the core frequency can last from tens of microseconds to over a millisecond in our experiments, becoming a potential point of contention for large core counts.

4.2.2 Architectural Support for DVFS Reconfiguration

With the trend towards highly parallel multicores the frequency of reconfigurations will significantly increase. This will be exacerbated by the increasing trend towards fine-grain task programming models with specific hardware support for task creation, data-dependences detection and scheduling [40, 72, 108]. Consequently, software-driven reconfiguration operations will be inefficient in future multicores. In such systems, hardware support for runtime-driven reconfigurations arises as a suitable solution to reduce contention in the reconfiguration process.

We propose a new hardware unit, the *Runtime Support Unit* (RSU), which implements the reconfiguration algorithm explained in the previous section. The RSU avoids continuous switches from user to kernel space, reducing the latency in reconfigurations and removing contention due to reconfiguration serialization. As illustrated in Figure 4.2, the RSU tracks the state of each core and the criticality of the running tasks to decide hardware reconfigurations and notify per-core frequency changes to the DVFS controller.

4.2.2.1 RSU Management

The RSU stores the criticality of the task running on each core (*Critical*, *Non-Critical*, or *No Task*), the status of each core (*Accelerated* or *Non-Accelerated*) and the corresponding *Accelerated* and *Non-Accelerated Power Levels* to configure the DVFS controller, together with the overall power budget for the system.

To manage the RSU, the ISA is augmented with initialization, reset and disabling

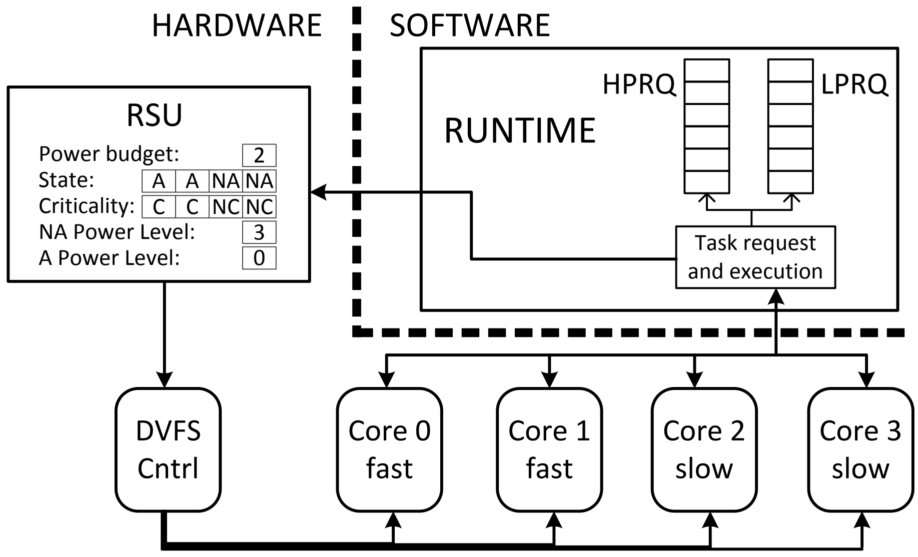


Figure 4.2: Architectural and runtime system support for CATA using DVFS reconfigurations. The RSU module implements the hardware reconfiguration functionality, and stores the same information as the RSM, plus the DVFS levels to use with Accelerated and Not Accelerated tasks.

instructions (`rsu_init`, `rsu_reset`, and `rsu_disable`, respectively), and control instructions (`rsu_start_task(cpu,critic)`) to notify the beginning of the execution of tasks and the completion of tasks (`rsu_end_task(cpu)`). Finally, another instruction is added to read the criticality of a task running in the RSU to deal with process virtualization (`rsu_read_critic(cpu)`).

An alternative implementation could manage the RSU through a memory mapped schema. We have selected the ISA extension approach due to its simple implementation. As the RSU is only accessed twice per executed task, both solutions are expected to behave similarly.

4.2.2.2 RSU Operation

The RSU reconfigures the frequency and the voltage of each core upon two different events: task execution start and end. Whenever one of these two events occurs, the RSU inspects its current state to determine which cores have to be accelerated and which ones decelerated. This decision is taken with the same algorithm presented in Section 4.2.1. When a task starts and there is available power budget the core is accelerated. If the task is critical and there is no power budget available but a non-critical task is accelerated, the core of the non-critical task is decelerated and then the core of the new task is accelerated. If all the other tasks are critical the new task is executed at low frequency. When a task finishes, the RSU decelerates its core and, if there is a critical task running on a non-accelerated core, it is accelerated.

4.2.2.3 RSU Virtualization

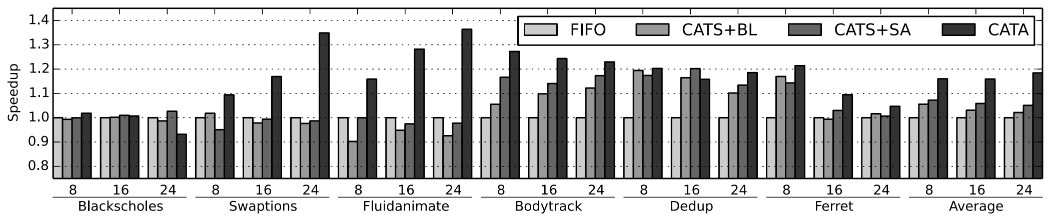
The OS saves and restores the task criticality information at context switches. When a thread is preempted, the OS reads the criticality value from the RSU and stores it in the associated kernel `thread_struct` data structure. The OS then sets a *No Task* value in the RSU to re-schedule the remaining tasks. When a thread is restored its task criticality value is written to the RSU. This design allows several concurrent independent applications to share the RSU.

4.2.2.4 Area and Power Overhead

The RSU requires a storage of 3 bits per core for the criticality and status fields, and $\log_2 num\_cores$ bits for the power budget. In addition, two registers are required to configure the critical and non-critical power states of the DVFS controller. These registers require $\log_2 num\_power\_states$ bits and are set to the appropriate values at OS boot time. This results in a total storage cost of $3 \times num\_cores + \log_2 num\_cores + 2 \times \log_2 num\_power\_states$ bits. The overhead of the RSU has been evaluated using CACTI [87]. Results show that the RSU adds negligible overheads in area (less than 0.0001$\%$ in a 32-core processor) and in power (less than $50\mu W$).

4.2.2.5 Integration of RSU and TurboMode

The RSU can be seen as an extension to TurboMode implementations such as Intel's Turbo Boost [54], AMD Turbo Core [5], or dynamic TurboMode [76]. TurboMode allows active cores to run faster by using the power cap of the sleeping cores. A core is considered active as long as it is in the C_0 or C_1 ACPI power states: C_0 means that the core is actively executing instructions, while C_1 means that it has been briefly paused by executing the `halt` instruction in the OS scheduler. If a core remains in a C_1 state for a long period, the OS suggests to move the core to C_3 or a deeper power state, considering it inactive. Transitions between different power states are decided in a hardware micro-controller integrated in the processor die. Whenever a core is set to C_3 or deeper power state, some of the active cores in the C_0 state can increase their frequency as long as it does not exceed the overall power budget. Thus, the RSU registers could be added to



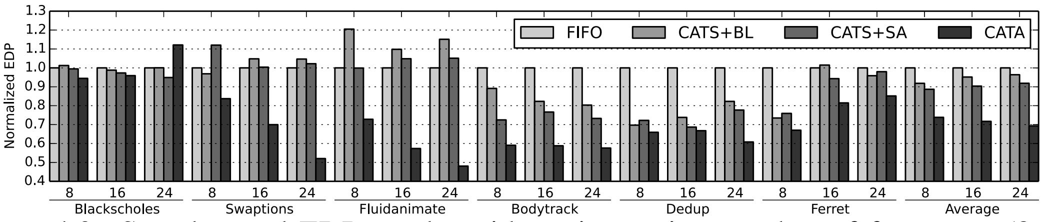


Figure 4.3: Speedup and EDP results with an increasing number of fast cores (8, 16, 24) on a 32-core processor. CATS+BL makes use of bottom-level and CATS+SA of static annotations methods to estimate task criticality. Results are normalized to the FIFO scheduler.

the TurboMode microcontroller to accelerate parallel applications according to the task criticality with minimal hardware overhead.

4.3 Evaluation

This section shows a detailed evaluation of the proposals presented above using the gem5 Simulator with the x86-64 architecture and McPat 1.3 configured as shown in Chapter 3 Section 3.1. Full system simulation is employed and a DVFS driver for the OS Kernel CPUFreq framework has been developed as detailed in Section 3.3.1 in order to ensure a realistic environment. The softwre stack comprises the Nanos++ runtime and six of the PARSECSs benchmarks described in Sections 3.3.2 and 3.5.1 respectively.

4.3.1 Criticality-Aware Task Scheduling

Figure 4.3 shows the execution time speedup and the normalized EDP of the four different software-only implementations of the system: FIFO, two variants of CATS which employ bottom-level (CATS+BL) and static annotations (CATS+SA) as criticality estimation methods, and CATA, which is analyzed in the next section. All results are normalized to the FIFO scheduler.

Results show that CATS solves the *blind assignment* problem of FIFO, providing average speedups of up to 5.6% for CATS+BL and up to 7.2% for CATS+SA with 8 fast cores. Static annotations perform better in these applications than bottom-level, which was originally designed and evaluated for HPC applications [32]. This happens because the static annotations approach does not suffer the overhead of exploring the TDG of the application, in contrast to bottom-level.

However, not all benchmarks benefit from exploiting task criticality in CATS. Fork-join or stencil applications (Blackscholes, Swaptions and Fluidanimate) present tasks with very similar criticality levels. As a result, scheduling critical tasks to fast cores does not significantly impact performance. In fact, the overheads of the bottom-level approach can degrade performance, reaching up to a 9.8% slowdown in Fluidanimate, where each task can have up to nine parent tasks.

Applications with complex TDGs based on pipelines (Bodytrack, Dedup and Ferret) benefit more from CATS. These applications contain tasks with significantly different criticality levels. For example, in the case of Dedup and Ferret there are compute-intensive tasks followed by I/O-intensive tasks to write results that are in the critical path of the application. In these cases, a proper identification and scheduling of critical tasks yields important performance improvements, reaching up to 20,2% in Dedup. In the case of Bodytrack, task duration can change up to an order of magnitude among task types. Since CATS+BL identifies critical tasks based only on the length of the critical path in the TDG, it obtains smaller performance improvements than CATS+SA. In the case of Dedup and Ferret, both schedulers perform similarly, although the lower overhead of CATS+SA slightly favors performance in some cases.

Figure 4.3 also shows the normalized EDP of all the mechanisms. We observe that the improvements in execution time translate into energy savings. CATS+SA obtains average EDP reductions between 8.2% and 11.4%, while CATS+BL EDP reductions ranges between 3.7% and 8.2%. Fork-join or stencil applications do not obtain significant EDP reduction. It is noticeable the effect of CATS+BL overhead in the case of Fluidanimate with 8 fast cores, as EDP increases by 22.1% over the baseline. In contrast, significant EDP improvements occur in the benchmarks with complex TDGs, achieving EDP reductions up to 31.4% in Dedup with 16 fast cores.

4.3.2 Criticality-Aware Task Acceleration

CATA can dynamically reconfigure the DVFS settings of the cores based on the criticality of the tasks they execute, avoiding the *static binding* and *priority inversion* issues of CATS, as discussed in Section 4.1. Figure 4.3 also shows the performance and EDP improvements that CATA achieves over FIFO. Based on the results in Section 4.3.1, we evaluate CATA using static annotations for criticality estimation.

Results show that CATA achieves average speedups of 15.9% to 18.4% over FIFO, and from 8.2% to 12.7% better than CATS+SA. The main improvements of CATA are obtained in fork-join or stencil applications, in particular Swaptions and Fluidanimate. In these applications, when tasks finish their execution before a synchronization point, CATA reassigns the available power budget to the remaining executing tasks, reducing the load imbalance. In contrast, in Blackscholes the number of tasks is very large and the load imbalance is low. This causes CATA to provide minimal performance benefits and even to present slight slowdowns with 24 fast cores. The slowdown is due to the overhead of frequency reconfigurations. In the applications with pipeline parallelism the performance improvement over CATS is lower, but still CATA obtains noticeable speedups of up to 28% in Bodytrack with 8 fast cores. CATA average improvements in EDP are significant, ranging from 25.4% to 30.1%. These gains are larger than the improvements in execution time as CATA reduces the power consumption of idle cores while it avoids priority inversion and static binding problems. Benchmarks with a large amount of load imbalance such as Swaptions and Fluidanimate dramatically reduce EDP, halving the baseline with 24 fast cores. When a task finishes and there are no other tasks ready to execute, CATA decelerates the core reducing the average number of fast cores decreasing power consumption.

4.3.3 Architecturally Supported CATA

Despite the significant performance and power benefits, CATA can be further improved by reducing the overhead of reconfiguring the computational power of the cores. As described in Section 4.2.2, frequency reconfigurations have to be serialized to avoid potentially harmful power states. In CATA this is done using locks and, as a result, it suffers from *reconfiguration serialization* overheads as the number of cores increases. This issue can become a bottleneck when one of the two following conditions holds: i) the amount of time spent performing reconfigurations is significant, or ii) the distribution of reconfigurations over time has a bursty behavior, which is the case in applications with synchronization barriers.

An analysis of the execution of the applications shows that the average reconfiguration latency of CATA ranges from 11 μs to 65 μs. However, maximum lock acquisition times in Blackscholes, Fluidanimate and Bodytrack reach several milliseconds (from 4.8 ms to 15 ms) due to lock contention. Additionally, although the average overhead of the reconfiguration time of the six applications ranges acceptable values of 0.03% to 3.49%, this overhead can be in the critical path and introduce load imbalance, increasing execution time significantly more than this average percentage as a result.

The RSU hardware component introduced in Section 4.2.2 speeds up reconfigurations and avoids taking locks as it centralizes all the reconfiguration operations. Figure 4.4 shows performance and EDP results using CATA, CATA+RSU and also TurboMode, which is discussed in the next section. Results are normalized to the FIFO scheduler to ease the comparison with Figure 4.3. On average, CATA+RSU further improves the performance of CATA, reaching an average 20.4% improvement over FIFO on a 32-core processor with 16 fast cores (it is 3.9% faster than CATA). Performance improvements are most noticeable in applications that suffer lock contention (Blackscholes, Fluidanimate and Bodytrack), reaching an average speedup over CATA of 4.4% on the analyzed applications, significantly reducing the performance degradation shown by Blackscholes with 24 fast cores, and achieving 8.5% speedup over CATA in Bodytrack with 24 fast cores. CATA+RSU reaches a maximum speedup over FIFO of 40.2% in Fluidanimate with 24 fast cores. Regarding the other applications (Swaptions, Dedup and Ferret), the additional improvements are on average small as lock contention is very low. Observed performance differences are mainly caused by changes in scheduling decisions induced by reconfigurations.

In EDP the average improvements range from 29.7% to 34.0% over FIFO and from 5.6% to 7.4% over CATA. The main reasons behind EDP reduction are the performance improvements and faster reconfigurations. Furthermore, in applications with a high lock contention the EDP reductions against CATA range from 4.0% to 9.4%. This proves the effectiveness of the proposed CATA+RSU and justifies the usefulness of such architectural support.

4.3.4 Comparison with Other Proposals

Finally, we compare CATA and CATA+RSU with an implementation of TurboMode [76]. For a fair comparison, our implementation of TurboMode considers the same two frequencies as in the previous experiments, with an overall power budget assigned in terms of maximum number of fast cores. TurboMode is not aware of task criticality, so the base FIFO scheduler is employed and all active cores (in state C_0) are assumed to be running critical tasks. Whenever an accelerated core executes the `halt` instruction triggered by the OS to transition from C_0 to C_1 state, the core notifies the TurboMode controller. The

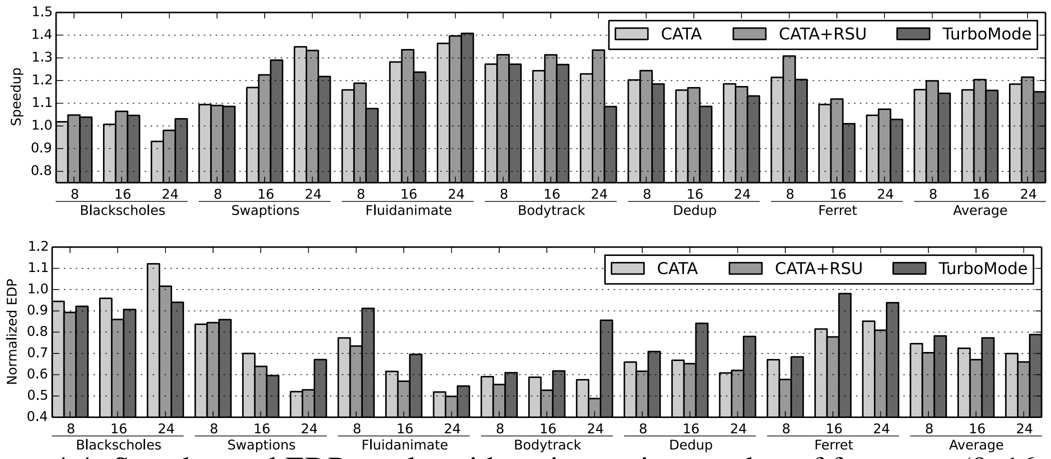


Figure 4.4: Speedup and EDP results with an increasing number of fast cores (8, 16, 24) on a 32-core processor. Results are normalized to the FIFO scheduler.

TurboMode controller lowers the frequency of the core, selects a random active core, and accelerates it. When the OS awakes a sleeping core, it notifies the TurboMode controller, and the core is accelerated only if there is enough power budget. Being able to quickly accelerate or decelerate at the C_1 state benefits applications with barriers or short idle loops, which do not need to wait for deeper sleeping states to yield their power budget to other cores.

Figure 4.4 shows the performance and EDP results of TurboMode. On average, TurboMode obtains slightly worse results than CATA, reaching between 14.4% and 15.7% performance improvements over FIFO. CATA+RSU outperforms TurboMode, with speedups between 4.0% and 5.3% and equivalent hardware cost. TurboMode presents competitive performance with CATA+RSU in fork-join and stencil applications (Blackscholes, Swaptions and Fluidanimate), but at the cost of higher energy consumption. This happens because CATA+RSU reconfigures the frequencies right at the moment that a task finishes its execution, while with TurboMode the reconfiguration must wait until the thread goes to sleep and triggers the `halt` instruction in the OS scheduler. In pipeline applications that overlap different types of tasks (Bodytrack, Dedup and Ferret), TurboMode performs worse than CATA+RSU with performance degradations up to 18.7% in Bodytrack with 24 fast cores. In the case of EDP results, pipeline applications obtain moderate improvements, with average results close to the ones obtained with CATS+SA.

TurboMode significantly improves performance over FIFO as it can solve the *static binding* issue. Since TurboMode is not aware of what is being executed in each core and its corresponding criticality, it may accelerate a non-critical task or runtime idle-loops. In contrast, CATA and CATA+RSU always know what to accelerate, effectively obtain-ing performance improvements. However, we have observed that TurboMode exhibits some characteristics that our proposals could benefit from. A thread executing a task can suddenly issue a `halt` instruction if the task requires any kernel service that suspends the core for a while; I/O operations, contention on locks that protects the page-fault and memory allocation routines are some examples that we have measured in Swaptions, Dedup and Ferret applications. CATA approaches are not aware of this situation causing the halted core to retain its accelerated state. On the contrary, TurboMode can drive that computing power to any other core that is doing useful work.

4.4 Remarks

Hardware mechanisms that allow reconfiguring the computational capabilities of the system are a common feature in current processors, as they are an effective way to maximize performance under the desired power budget. However, optimally deciding how to reconfigure the hardware is a challenging problem, because it highly depends on the behavior of the workloads and the parallelization strategy used in multi-threaded programs. In ATaP models, where a runtime system controls the execution of parallel tasks, the criticality of the tasks can be exploited to drive hardware reconfiguration decisions in the runtime system.

This chapter presents an integrated solution in which the runtime system of ATaP models performs Criticality Aware Task Acceleration. In this approach the runtime system schedules tasks to cores and controls their DVFS settings, accelerating the cores that execute critical tasks and setting the cores that execute non-critical tasks to low-frequency power-efficient states. Since performing DVFS reconfigurations in software can cause performance overheads due to serialization issues, this chapter also proposes a hardware component, the RSU, that relieves the runtime system of carrying out DVFS reconfigurations and can be seen as a minimal extension to existing TurboMode implementations [5, 54, 76]. With this hardware support, the runtime system informs the RSU of the criticality of the tasks when they are scheduled for execution on a core, and the RSU reconfigures the voltage and the frequency of the cores according to the criticality of the running tasks.

Results show that CATA outperforms existing scheduling approaches for heterogeneous architectures. CATA solves the blind assignment issue of FIFO schedulers that do not exploit task criticality, achieving improvements of up to 18.4% in execution time and 30.1% in EDP. CATA also solves the static binding and priority inversion problems of CATS, which results in speedups of up to 12.7% and improvements of up to 25% in EDP over CATS. When adding architectural support to reduce reconfiguration overhead, CATA+RSU obtains an additional improvement over CATA of 3.9% in execution time and 7.4% in EDP, while it outperforms state-of-the-art TurboMode as it does not take into account task criticality when deciding DVFS reconfigurations.

Chapter 5

Improving Performance Through Fine-Grained Tasking

Using different scheduling policies is key to maximize the efficiency of applications and systems [107]. Considering task criticality [31, 122] or data locality [59] provides significant benefits in certain contexts. Moreover, the adaptability granted by software task schedulers is essential in modern high-performance computing systems with off-chip accelerators and multi-socket configurations can further improve performance and energy efficiency, but require software intervention for task scheduling and data motion.

We present Task Dependence Manager (TDM), a hardware/software co-designed mechanism that accelerates the most time consuming activities of the runtime system with specialized hardware while allowing flexible task scheduling policies in software. TDM minimally extends the ISA to allow the runtime system to communicate task creation, task dependences and task finalization, and to request ready tasks. At the architecture level, TDM introduces a Dependence Management Unit (DMU) that maintains the information of the in-flight tasks and the dependences between them by means of a set of tables and lists. Tasks ready for execution are exposed to the runtime system, which has the freedom for deploying any software scheduling policy. The main contributions of this proposal are:

- A novel hardware/software co-designed mechanism to accelerate task creation and dependence tracking while supporting flexible software schedulers. The hardware design includes novel architectural techniques to minimize conflicts in associative structures and to reduce the hardware cost with respect to previous proposals.
- A detailed evaluation of TDM on a full-system simulator that includes application, runtime system, operating system and architecture layers. On a 32-core processor,

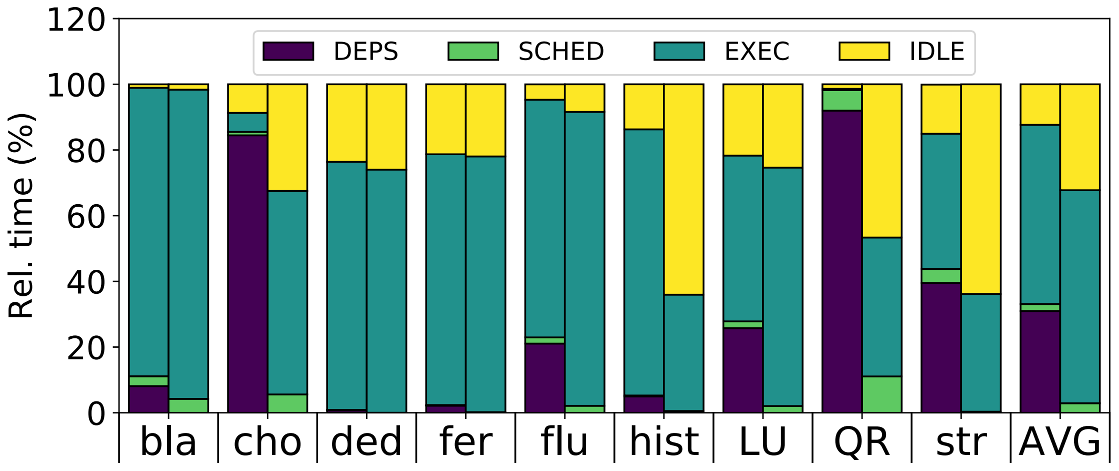


Figure 5.1: Execution time breakdown of the master and worker threads during the parallel execution. Different states represent dependence management operations during task creation and task finalization (DEPS), scheduling (SCHED), task execution (EXEC), and idle time (IDLE).

TDM achieves a 12.3% average speedup and a 20.4% reduction in EDP with respect to a baseline implemented in software.

- A proof of the potential of TDM when combined with five software schedulers that exploit the characteristics of different applications. Thanks to this flexibility, TDM outperforms a runtime fully implemented in hardware by an average 4.2%, improves EDP by an average 6.2%, and reduces the area overhead by $7.3\times$.

5.1 Characterizing Runtime System Activity

Performance and scalability of parallel programs is fundamentally limited by the overheads introduced in the form of idle time and runtime system phases to manage tasks and dependences [48]. These two sources of overheads are tightly related to the granularity of the tasks. On the one hand, coarse-grained tasking reduces the overheads of task creation and dependence management, but compromises load balancing and scalability on large-scale multi-cores. On the other hand, fine-grained tasking favors load balancing, but increases the overheads of the runtime system in dependence management and task scheduling phases. In addition, many operations in the runtime system phases need to be serialized to avoid race conditions, potentially becoming a bottleneck as concurrency increases with higher core counts.

We characterize the cost of the runtime system phases in 9 representative task-based parallel benchmarks running on a simulated 32-core processor. The optimal task granularity in each experiment has been carefully selected to minimize execution time[1]. Figure 5.1 shows a complete break-down of the time spent in the main program phases for the master (left bars) and the worker threads (right bars): task creation and dependence management (DEPS), scheduling (SCHED), task execution (EXEC), and idle time (IDLE).

The master thread spends a significant portion of the time in DEPS for Cholesky, QR and streamcluster (84%, 92% and 40%, respectively). In these cases, illustrated in the timeline of Figure 2.7 for Cholesky, the bottleneck of the execution is the pace at which tasks are created by the master thread, that limits the amount of available tasks for the worker threads and causes idle time. DEPS has a lower impact in the rest of benchmarks, below 25.8%, but idle time is still relevant in the worker threads due to load imbalance. Most of the time spent in DEPS is devoted to identify the dependences of a task when it is created, which requires comparing the inputs and outputs of the new task against the ones of the older tasks. Thread synchronization overheads are negligible, as they only represent 0.9% of the DEPS time and 2.2% of the SCHED time. Overall, worker threads spend most of the time executing tasks (65% of the time on average) or idle (32% of the time), and the master thread spends a significant amount of time running tasks in the majority of benchmarks, while scheduling time is much less significant.

Adding architectural support for the runtime system can mitigate the overheads of fine-grained tasking. Approaches such as Carbon [71] move the task scheduler to the hardware level, while Task Superscalar [39] offloads all the runtime system activities to the architecture, including dependence management and task scheduling. The main drawback of these schemes is that the task scheduler is fixed in the architecture, which compromises the flexibility of the system. The system flexibility provided by software runtime systems is of paramount importance in modern systems with multiple sockets and off-chip accelerators, since the task scheduler needs to off-load tasks to external components that are only visible to the software and often require software-initiated actions such as data movement between address spaces. To maintain these advantages, approaches such as ADM [107] add architectural support for asynchronous exchanges of short messages between cores that can be used to implement low-overhead thread synchronization primitives.

All these solutions drastically reduce runtime system overheads, even in scenarios

[1]Chapter 3 describes in detail the experimental setup, and Figure 5.5 explores the optimal task granularity of each benchmark.

with extremely fine-grained tasks running on hundreds of cores. However, in scenarios with mid-grained or less extreme fine-grained tasks[2], the cost of task scheduling is relatively low, less than 11% in all benchmarks in Figure 5.1, so the benefits of flexible software scheduling can be achieved with minimal performance impact. In contrast, the cost of dependence management operations during task creation is crucial for performance because it determines the idle time in the whole execution, so adding hardware support to perform this operation can effectively reduce the runtime system overheads.

5.2 TDM Design

TDM is a hardware/software co-designed mechanism to support the runtime system. TDM addresses the performance bottlenecks of pure software dataflow runtime systems by proposing a hardware/software co-designed mechanism that performs dependence management operations efficiently in hardware and allows the usage of different task scheduling policies in the runtime system. Thanks to this separation of concerns, TDM is able to mitigate the performance overheads introduced in runtime system phases while providing flexibility to the software layers, so the resulting system is more adaptable, composable, and is able to capitalize on the benefits of different scheduling policies for different applications.

TDM balances the higher cost and performance of implementing mechanisms in hardware, with the higher flexibility and adaptability of implementing policies in software. At the architecture level TDM introduces a DMU that keeps a representation of the TDG and allows the runtime system to offload costly dependence tracking operations, while leaving scheduling decisions to the runtime system. As a result, TDM avoids the overheads of software runtime systems and maintains the flexibility of supporting software schedulers.

The runtime system interacts with the DMU to communicate task creation, the data dependences of the tasks, and task finalization. With this information, the DMU generates the TDG, tracks dependences between tasks, identifies tasks ready for execution, and exposes them to the runtime system. The runtime system can request ready tasks to the DMU, organize them in software data structures, and schedule them to the cores according to any scheduling policy.

[2]In this paper we use task granularities up to 3 orders of magnitude bigger than other works of the literature.

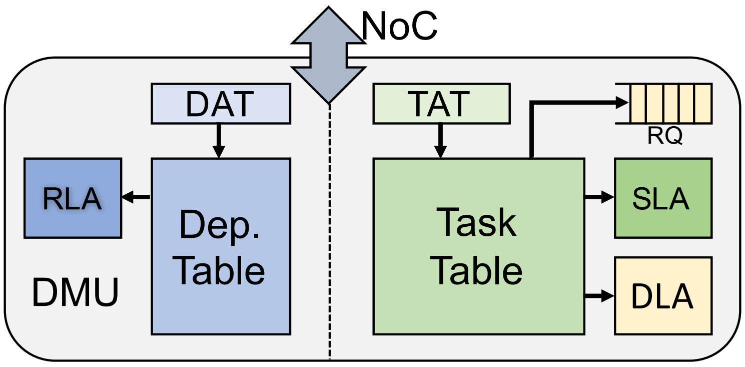


Figure 5.2: DMU architectural support overview.

5.2.1 Runtime System - Architecture Interface

TDM offers an interface to the runtime system so that it can cooperate with the DMU in the management of tasks. The interface between the DMU and the runtime system consists of four new ISA instructions. These instructions are issued by the runtime system in the task creation and task finalization phases to exchange information with the DMU.

- *create_task(task_desc)*: In the task creation phase, the runtime system uses this instruction to inform the DMU that a new task is being created. The DMU receives the task descriptor address of the new task.

- *add_dependence(task_desc, dep_addr, size, direction)*: After creating a task, the runtime system traverses its list of dependences and uses this instruction to inform the DMU of the dependences of the task, sending the task descriptor address, the address of the dependence, the size, and the direction (input or output). With this information the DMU tracks tasks and dependences and builds the TDG to ensure dependences between tasks are fulfilled.

- *finish_task(task_desc)*: When a task finishes its execution, the runtime system uses this instruction to notify it to the architecture. The DMU wakes up the successors of the task and cleans up the information of the task and its dependences from its internal structures.

- *get_ready_task()* $\rightarrow$ *task_desc, #succ*: Just after notifying a task has finished, the runtime system uses this instruction to request to the DMU the successors of the finished tasks that have just become ready. This instruction returns the task descriptor address and its number of successors.

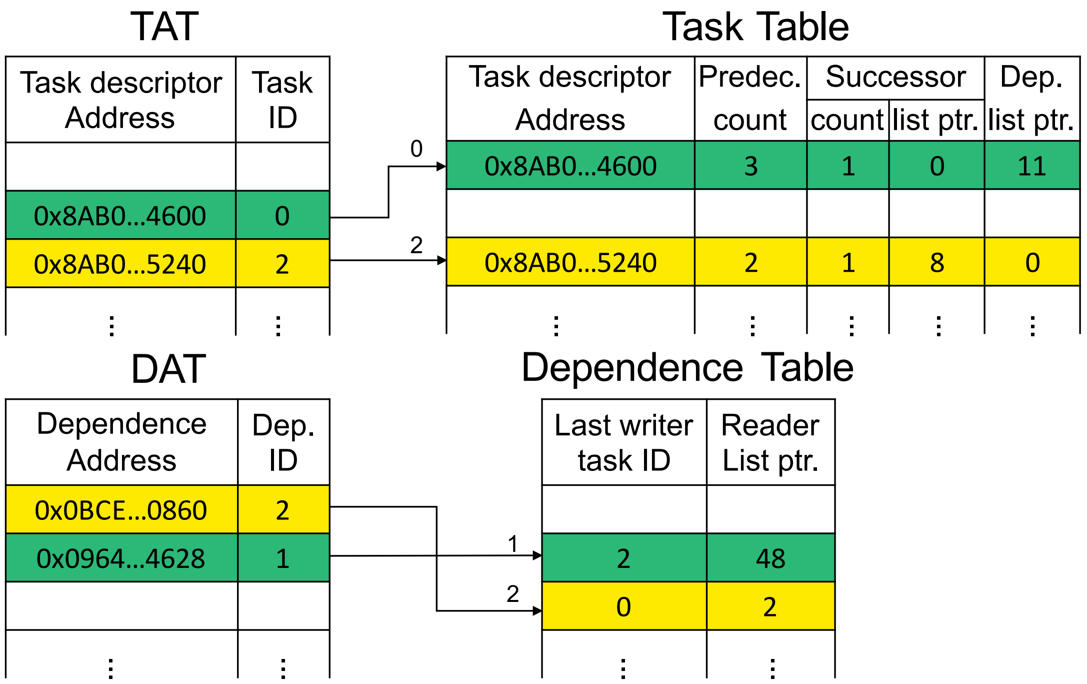


Figure 5.3: Overview of TAT, DAT, Task and Dependence Table. Two active elements are presented in each table.

5.2.2 DMU Hardware Design

The DMU is a centralized module connected to the network-on-chip whose main goal is to keep all the relevant information of the in-flight tasks, track the dependences between them, and expose ready tasks to the runtime system. Figure 5.2 presents its different components. Each task or dependence is internally identified by an ID, which maps to its location in the corresponding table. Tables and list arrays employ SRAM memories, addressed by the task or dependence IDs. Two set-associative structures, TAT and DAT, are used to map task descriptor and dependence addresses to internal DMU IDs. The general behavior of each module follows:

- The *Task* and *Dependence Alias Tables* (TAT and DAT) keep a translation of task descriptor addresses or dependence addresses to internal task or dependence IDs.
- The *Task Table* and the *Dependence Table* track all the information of the in-flight tasks and dependences.
- The *List Arrays* (*Successor, Dependence* and *Reader*) contain lists of elements associated to in-flight tasks or dependences. The *successor* and *reader* lists store task IDs, while the *dependence* list stores dependence IDs.

- The *Ready Queue* (RQ) is a FIFO queue that contains task IDs ready to be executed.

5.2.2.1 Task and Dependence Identifier Renaming

The alias tables are depicted in Figure 5.3. Both TAT and DAT modules consist of a directory that maps task descriptor and dependence addresses to task and dependence IDs, respectively, and an additional queue of free IDs. Both modules are implemented using set-associative memories.

Selecting the correct bits to index the DAT is crucial to avoid conflicts. It is common that different tasks access different blocks of the same data structure, so the lower bits of the addresses of different dependences share the same values. For example, if tasks access different 4KB blocks of a vector, the lower 12 bits of all the dependences are equal. If these bits are used to index the DAT, only one set is used and many conflicts happen. To avoid conflicts, the size of the dependence is used to select the address bits employed as index, which start at the $log_2 size$ lower bit.

The alias tables allow the rest of DMU modules to work with internal IDs, which offers two important advantages. First, the Task and Dependence Tables employ RAM memories, indexed with the internal task and dependence IDs, avoiding costly associative lookups of 64-bit task descriptor and dependence addresses keys. Therefore, using TAT and DAT a single lookup is required per DMU instruction, followed by many subsequent direct accesses to the Task and Dependence Tables, as explained in Section 5.2.3. Second, the storage requirements of the list arrays can be reduced significantly, as the size of the internal IDs is much smaller than the 64-bit identifiers used in the runtime system. Our experiments in Section 5.3.2 show that DAT and TAT with 2048 entries suffice for any application, so 11-bit IDs can be used and the size of the list arrays is reduced by a factor of $5.8\times$.

5.2.2.2 Task and Dependence Tracking

The Task and Dependence Tables are used to keep the information of the tasks and the dependences. The Task Table is an SRAM indexed by the Task ID. Figure 5.3 shows each entry of the Task Table containing the relevant information of a task: its descriptor address, the number of successors and predecessors, and pointers to the lists of successors and dependences. The Dependence Table follows the same scheme to track dependences, storing the task ID of the last task that writes the dependence and a pointer to the list of readers.

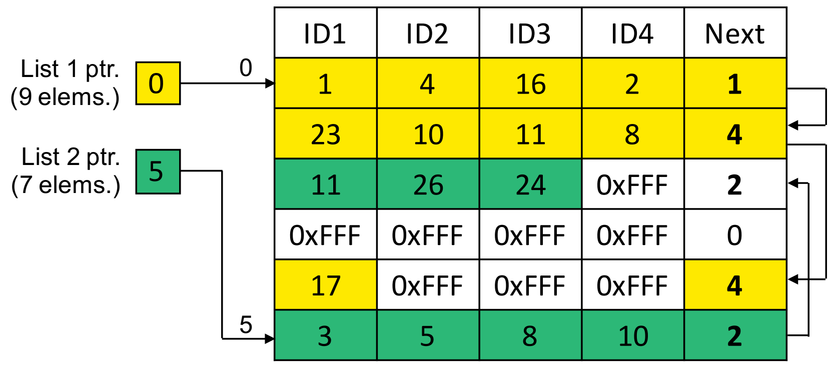


Figure 5.4: Overview of a generic list array.

The lists of successors, dependences and readers are implemented in three list array structures. As shown in Figure 5.4, each list array is an SRAM that can store multiple lists. To accommodate a variable number of elements in each list we use a storage layout inspired by UNIX filesystem inodes. The maximum number of elements in each entry is fixed by design (4 in the example), but the list can continue in another entry. The *Next* control field of every entry points to the entry in the list array where the list continues. The *Next* field is set to the current entry number if the list finishes in this entry. Invalid elements are set to all ones.

The Successor List Array uses this organization to store the lists of successors of each in-flight task, identified by their task IDs. Task IDs are also stored in the lists of the Readers List Array, which track the reader tasks of all the in-flight dependences. The Dependence List Array keeps the lists of dependences of the in-flight tasks, so dependence IDs are stored in the lists. Note that OpenMP 4.0 uses the input/output dependences provided by the programmer to build the TDG when tasks are created in program order. The DMU preserves this model by decoupling the dependences, that are tracked in the dependence and readers lists, from the edges of the TDG, that are tracked in the successors lists.

5.2.3 Operational Model

The runtime system triggers DMU operations using the ISA instructions in the task creation and finalization phases.

5.2.3.1 Task Creation

The runtime system uses the *create_task* instruction to send the task descriptor address to the DMU. Then, for every dependence of the task, it uses the *add_dependence* instruction

Algorithm 2: Algorithm for *add_dependence* instruction.

```
Data: taskID, depID, dir
Insert depID in dependence list of taskID;
if lastWriterID of depID is valid then
    Insert taskID in successor list of lastWriterID;
    Increment #succ of lastWriterID;
    Increment #pred of taskID;
end
if dir is In then
    Insert taskID in reader list of depID;
end
if dir is Out then
    for readerID in reader list of depID do
        Insert taskID in successor list of readerID;
        Increment #succ of readerID;
        Increment #pred of taskID;
    end
    Flush reader list of depID;
    Set lastWriterID of depID to taskID and mark valid;
end
```

to inform the DMU.

When the *create_task* instruction is executed, the DMU uses the TAT to generate a task ID. The Task Table is indexed with the task ID and the entry is initialized by setting the task descriptor address, setting to 0 the number of successors and predecessors, and reserving a new list of successors and a new list of dependences in the Successor and Dependence List Arrays. If some structure of the DMU has no entries available the instruction blocks until an entry is freed.

After the task is created, for every *add_dependence* instruction an entry is allocated in DAT and Dependence Table. The DMU uses TAT to obtain the task ID and DAT to obtain the dependence ID. Then, the DMU behaves as described in Algorithm 2. First the dependence is inserted in the list of dependences of the task and the task ID is inserted in the successor list of the last writer of the dependence. Then, if the dependence is an input, the task ID is inserted in the readers list of the dependence. Otherwise, if the dependence is an output, all the readers of the dependence insert the task in their successor lists, the reader list is flushed, and the task becomes the last writer of the dependence.

Algorithm 3: Algorithm implemented by DMU for the *finish_task* instruction.

```
Data: taskID
for succID in successor list of taskID do
    Decrement #pred of succID;
    if #pred of succID = 0 then
        Insert succID in the Ready Queue;
    end
end
for depID in dependence list of taskID do
    Remove taskID from reader list of depID;
    if lastWriterID of depID = taskID then
        Mark lastWriterID of depID as invalid;
    end
    if lastWriterID of depID is invalid &&
    reader list of depID is empty then
        Free reader list of depID;
        Free depID entry in DepTable and DAT;
    end
end
Free successor list of taskID;
Free dependence list of taskID;
Free taskID entry in TaskTable and TAT;
```

5.2.3.2 Task Finalization

When a task finishes, the runtime system uses the *finish_task* instruction to communicate the task descriptor address to the DMU, and this carries out the steps described in Algorithm 3. In the first loop the DMU wakes up the successor tasks by traversing the successor list of the task and decrementing the number of predecessors of each successor. If the number of predecessors becomes zero, the successor task is moved to the Ready Queue. In the second loop the task is removed from the reader list and the last writer field of each of its dependences. Finally the DMU frees the entries allocated for the task in the Task Table, the TAT, and the Successor and Dependence List Arrays.

5.2.3.3 Implementing Task Schedulers in Software

After the finalization of a task the runtime system requests ready tasks to the DMU by issuing *get_ready_task* instructions in a loop. For every *get_ready_task* instruction the DMU consults the Ready Queue. If it is empty, a null pointer is returned. Otherwise, the task ID at the head of the queue is retrieved and used to index the Task Table to get the task descriptor address and the number of successors that are returned to the runtime system. Then the runtime system adds the returned task descriptor address to a pool of ready tasks and stores the number of successors in the task descriptor.

The pool of ready tasks can be used by the runtime system to implement any scheduling policy. The scheduling algorithms can traverse the pool of ready tasks in any order, move ready tasks to different data structures, or perform any action required by each particular implementation. By allowing the usage of different task schedulers, TDM provides flexibility, adaptability and composability to the system.

5.2.4 Additional Considerations

The size of the hardware structures of the DMU limit the number of in-flight tasks and dependences. To preserve correctness, the TDM ISA instructions have barrier semantics, so they cannot be re-ordered in the CPUs and younger instructions cannot be executed before the TDM instructions commit. The DMU processes the instructions sequentially and, if there is no room available in some structure, the instruction is blocked until some in-flight task finishes.

TDM manages tasks and dependences inside parallel regions and relies on the runtime system to handle barriers and other global synchronization points. To do so, the master thread executes the code sequentially and creates the tasks while the worker threads request tasks and execute them. The runtime system tracks how many tasks have been created by the master thread and how many have been executed. When the master thread reaches the barrier it adopts the behavior of a worker thread, and when all the tasks have been executed it resumes the sequential execution of the program.

The proposed design of TDM can be easily extended to support context switches and multiprogrammed workloads. A simple and effective solution is to tag TAT and DAT with the operating system process ID, so different processes can use TDM concurrently and the structures of the DMU do not need to be saved and restored at context switch.

The centralized design of the DMU is not a limiting factor for scalability. The DMU executes several instructions per task that, all together, take tens to hundreds nanoseconds, while the average task duration in our experiments is 4771 microseconds, as shown in Section 5.3.1. Given that the task duration is 5 orders of magnitude larger than the latency of the DMU instructions per task, the DMU is able to scale up to thousands of concurrent tasks before becoming a bottleneck.

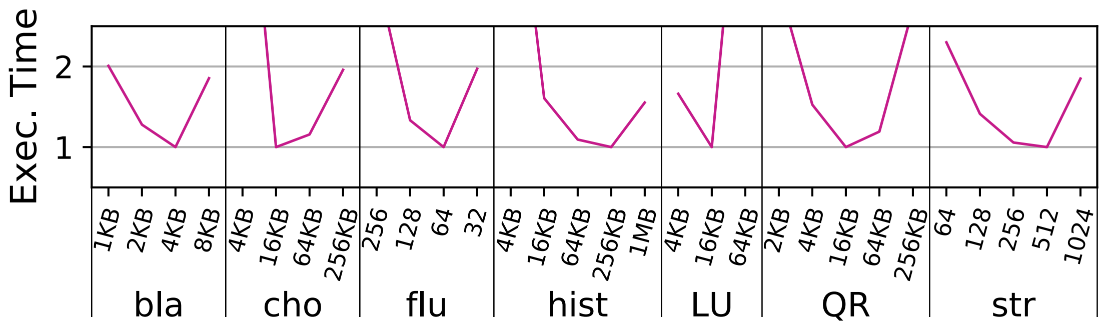


Figure 5.5: Execution time for different task granularities. The X axis shows the size of the blocks processed by each task in Blackscholes, Cholesky, Histogram, LU, and QR; the number of partitions of the 3D volume in Fluidanimate; and the number of points processed by each task in Streamcluster.

5.3 Design Space Exploration

5.3.1 Benchmarks and Task Granularity

To test TDM we use five benchmarks from PARSECSs [28], a task-based OpenMP 4.0 implementation of the PARSEC [14], together with four benchmarks from the high performance computing domain: Cholesky, Histogram, LU and QR. These benchmarks are representative algorithms and use different parallelization strategies: Blackscholes and Streamcluster use fork-join parallelism, Fluidanimate is a 3D stencil, and Dedup and Ferret use pipeline parallelism. Regarding the other four benchmarks, Cholesky performs a Cholesky decomposition of a matrix, Histogram computes a cumulative histogram for all pixels of an image, LU does a LU decomposition of a matrix, and QR calculates a QR factorization of a matrix. Tiling is applied in these algorithms so that tasks process 2D blocks of the matrices.

The benchmarks are compiled with Mercurium 1.99 source-to-source compiler [8] with gcc 4.6.4 as backend compiler. *Simlarge* input sets are used for the PARSEC benchmarks, Cholesky decomposes a dense 2048×2048 matrix, histogram processes a 4096×4096 image and generates a histogram with 10 bins, LU decomposes a sparse 2048×2048 matrix, and QR a dense 1024×1024 matrix.

In all benchmarks we ensure that parallel regions scale well to 32 cores using performance analysis tools to visualize the parallel executions. The optimal task granularity is carefully selected to minimize load imbalance and execution time in the baseline software approach. Figure 5.5 shows the execution time with different task granularities growing

Table 5.1: Benchmark characteristics. Number of tasks and average task duration with the optimal task granularity for the software runtime system and for TDM.

	Software		TDM	
	# tasks	Duration (μs)	# tasks	Duration (μs)
Blackscholes	3,300	1,770	6,500	823
Cholesky	5,984	183	5,984	183
Dedup	244	27,748	244	27,748
Ferret	1,536	7,667	1,536	7,667
Fluidanimate	2,560	1,804	2,560	1,804
Histogram	512	3,824	512	3,824
LU	1,512	424	1,512	424
QR	1,496	997	11,440	96
Streamcluster	42,115	376	42,115	376
Average	**6,584**	**4,976**	**8,056**	**4,771**

along the X axis (i.e., smaller to bigger from left to right). Execution time is normalized to the optimal task granularity. In Dedup and Ferret the task granularity cannot be changed without modifying the application, as each task processes a pipeline stage. In general, shorter task duration increases parallelism, but leads to higher runtime system overheads.

Table 5.1 summarizes the number of tasks and their average duration for each benchmark. The number of tasks ranges from 244 (Dedup) to 42,115 (Streamcluster), and the average duration between 96μs (QR) and 27ms (Dedup). The optimal task granularity is used for the corresponding approach (software or TDM) in all the experiments of the evaluation.

5.3.2 TAT, DAT and List Arrays

We perform a design space exploration to determine the optimal size of the DMU hardware structures. All the experiments are performed in the gem5 simulator using an ARM platform consisting of 32 cores and full system simulation as detailed in Section 3.1. We first study the sizing of TAT and DAT, considering a DMU implementation with N TAT entries, M DAT entries, and unlimited entries in the list arrays. The size of the TAT and the DAT determine the size of the Task and Dependence Table, respectively. Figure 5.6 shows the performance obtained when N and M vary between 512 and 4096. Perfor-

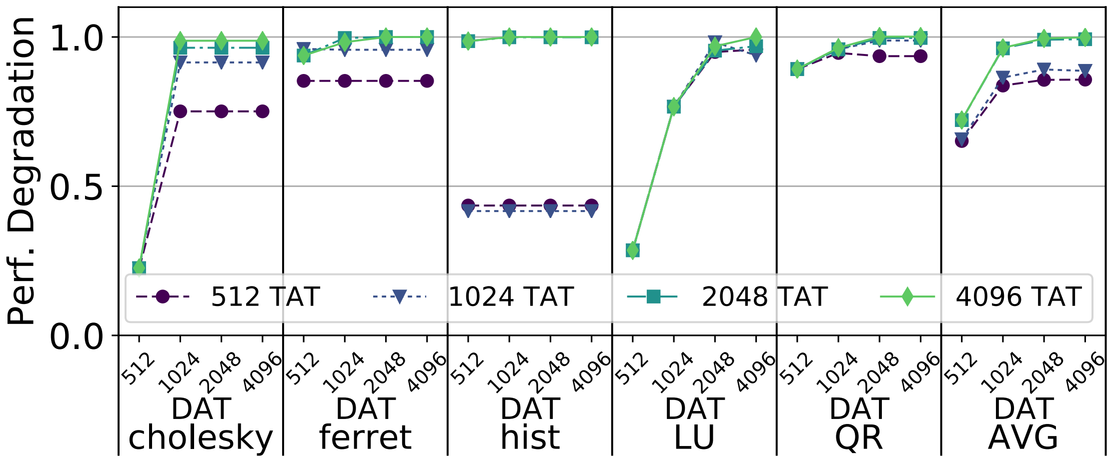


Figure 5.6: Average performance with different sizes of the TAT and DAT. Results are normalized to an ideal DMU with unlimited entries and equal latency.

mance is normalized to an ideal design with an infinite number of entries in all DMU structures and same latency.

Figure 5.6 shows results for 5 benchmarks. The rest of benchmarks already achieve maximum performance with 512 entries in DAT and TAT. The geometric mean considers all the benchmarks. The figure shows LU and QR are sensitive to the DAT size, achieving maximum performance with 2048 entries. The other three benchmarks are sensitive to the TAT size. The most demanding benchmark is Histogram, as its tasks have a significant amount of dependences between them and the distance between independent tasks is high. Thus, it requires at least 2048 TAT entries to achieve maximum performance. On average, with 2048 entries in both DAT and TAT, the DMU only suffers a 0.91% performance degradation with respect to the ideal case with infinite entries and same latency. We also explore the associativities of TAT and DAT, results showing that 8-way associative structures minimize conflicts and offer the best performance.

Next we explore the size of the successor, dependence and reader list arrays. Figure 5.7 shows the average performance when these structures vary from 128 to 2048 entries, normalized to an ideal design with an infinite number of entries in all DMU structures and same latency.

These results clearly show that a design with 128 entries in any of the list arrays leads to suboptimal performance. In contrast, with 1024 entries in all the list arrays, performance already saturates. On average, with 1024 entries in all list arrays, the DMU only suffers a 1.1% performance degradation with respect to the ideal case with an infinite number of entries and same latency. Doubling the size of all list arrays leads to an average

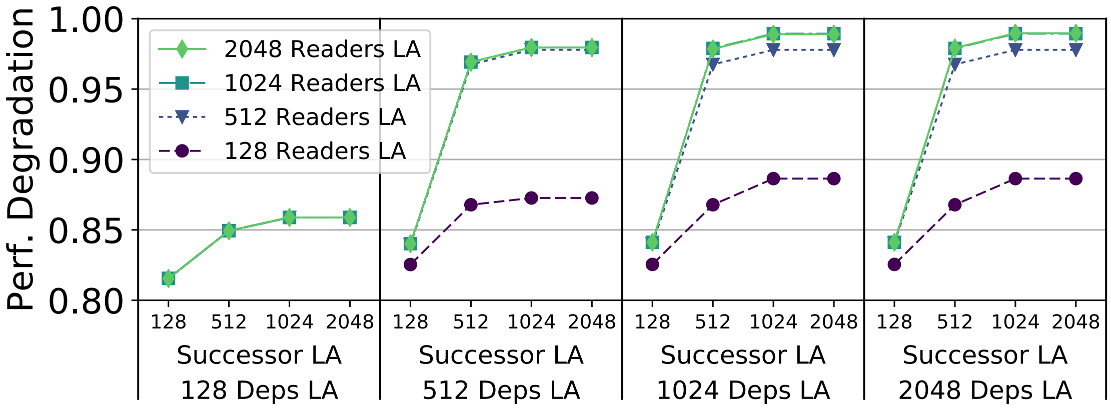


Figure 5.7: Average performance with different sizes of the list array (LA) structures. Results are normalized to an ideal DMU with unlimited entries and equal latency.

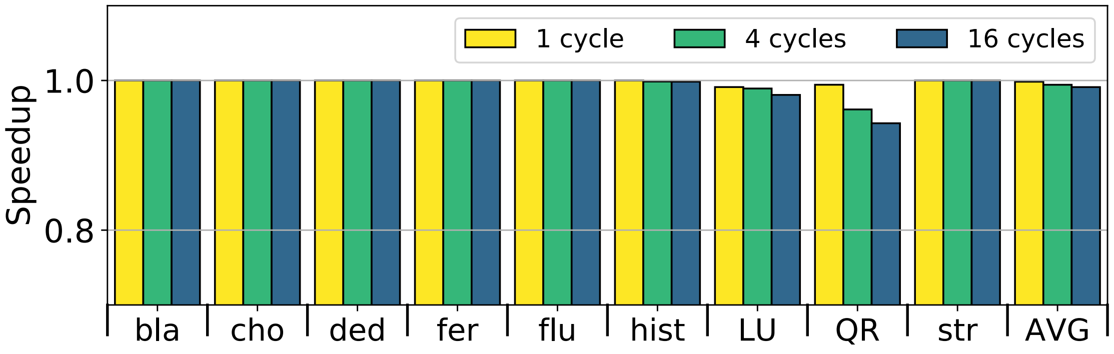


Figure 5.8: Performance degradation when varying the access time of all DMU structures from 1 to 16 cycles. Results are normalized to DMU structures with zero latency.

1.0% performance degradation, but requires a significant increase in area. For this reason, we size all list arrays in the DMU with 1024 entries.

5.3.3 DMU Access Latency

As explained in Section 5.2, the algorithms that implement TDM instructions require accessing different hardware structures. Also, the lists stored in the list arrays may spread over multiple entries, which requires multiple accesses to traverse the complete lists. Consequently, DMU operations require multiple cycles to finalize. Next, we evaluate the performance of the DMU when varying the latencies of its hardware structures. In these experiments we use the sizes of the DMU structures determined in the previous section.

Figure 5.8 shows the performance degradation when increasing the access time of all DMU structures from 1 to 16 cycles. Most benchmarks do not suffer any performance degradation due to higher latencies, as with the optimal task granularity DMU operations

Table 5.2: DMU storage (KB) and area (mm^2) requirements.

	Storage	Area		Storage	Area
Task Table	23.00	0.026	SLA	12.25	0.019
Dep Table	5.25	0.013	DLA	12.25	0.019
TAT	18.75	0.031	RLA	12.25	0.019
DAT	18.75	0.031	ReadyQ	2.75	0.012
Total	**105.25 KB**	**0.17 mm^2**			

happen infrequently. Only LU and QR are slightly affected by this parameter. On average, performance degrades only 0.2% with a 1-cycle access time and 0.9% with a 16-cycle access time.

5.3.4 DMU Area and Power Overhead

Table 5.2 shows the storage and area requirements of the DMU for the sizes selected in Section 5.3.2. Storage values consider the number of bits of the task and dependence IDs, which depend on the size of the tables they point to. The structures are modeled in CACTI 6.0 [87] to obtain the area values with a process technology of 22 nm.

The components of the DMU have a negligible effect on the power consumption, less than 0.01% of the total power. The low power requirements of the DMU combined with the small sizes of the hardware structures allow to design the DMU with a 1-cycle access time to each data structure.

As a conclusion of this design space exploration, we select a design with a DAT and TAT of 2048 entries and all the list arrays of 1024 entries. The storage and area requirements for this configuration, 105.25KB and 0.17mm^2, are very affordable with current design technology. The rest of this chapter makes use of this configuration in all the experiments.

5.3.5 Runtime Overhead Reduction

Next, we measure the impact of TDM in the task creation time. Figure 5.9 shows the average time spent by the master creating tasks and managing their dependences, which corresponds to the DEPS category in Figure 5.1. Task creation time is not completely eliminated with TDM because of the latency of the DMU structures and because some operations are still performed in the runtime system, such as creating task descriptors, issuing TDM instructions, etc. All benchmarks benefit from the hardware support provided by the DMU, achieving up to a $5.2\times$ reduction in task creation time in Blackscholes.

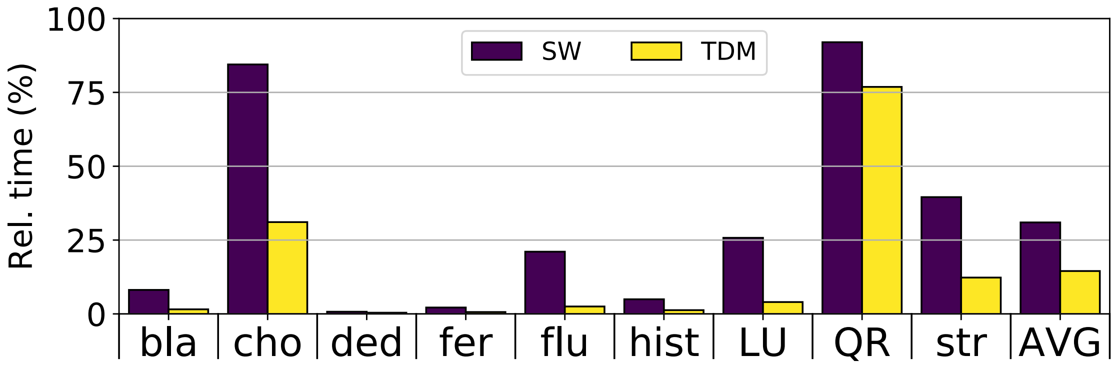


Figure 5.9: Percentage of time spent in task creation with a pure software approach (SW) and with TDM.

On average, task creation is reduced from from 31.0% to 14.5% of the total CPU time, proving the effectiveness of TDM. This reduction of task creation time has a big impact on the idle time, that is reduced from 32% to 22% on average, and translates into overall application speedups as will be shown in Section 5.4.

5.3.6 Index Bit Selection for DAT

We show the importance of selecting the appropriate bits of the dependence addresses to index the DAT. As described in Section 5.2.2.1, when different blocks of the same data structure are specified as dependences, many dependence addresses have the same values in the lower bits, causing conflicts if these bits are selected to index the DAT. To avoid this problem, the DMU uses the size of the dependence to select the bits of the dependence addresses to index the DAT.

Figure 5.10 shows the average number of occupied sets in the DAT for the six benchmarks that are sensitive to this issue. The X axis shows 5 numerical values that correspond to different options to statically select the index bits (e.g., 4 means the index bits start at the 4th lower bit of the dependence address), and the proposed dynamic mechanism (DYN) that uses the size of the dependence. Results show that each fixed value drastically changes the occupancy of the DAT, from 1% to 88%. More importantly, every benchmark requires selecting different index bits. This happens because the benchmarks use different block sizes, so the number of lower bits that are equal in the dependence addresses changes in every benchmark. By using the size of the dependences provided by the runtime system to dynamically select the index bits, the DMU avoids conflicts in the DAT and maximizes its occupancy in all benchmarks.

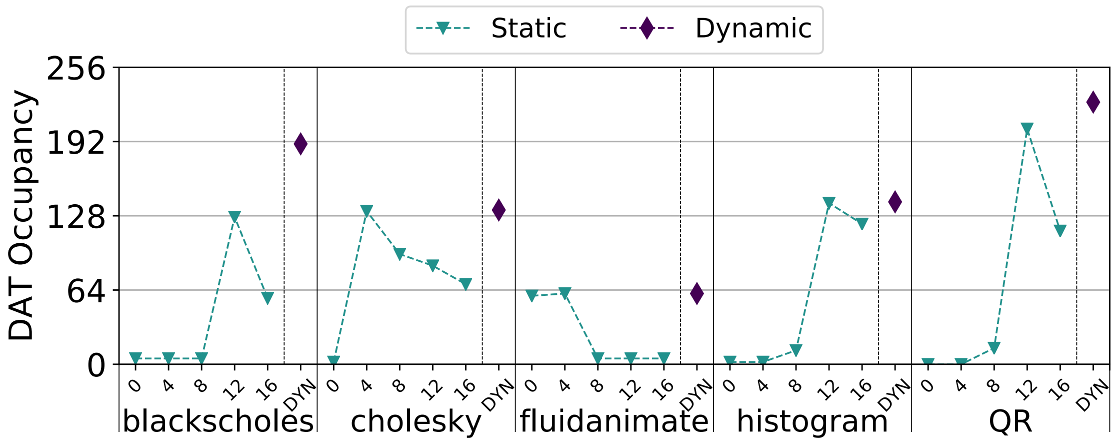


Figure 5.10: Occupancy of sets in DAT with static index bit selection and with dynamic index bit selection.

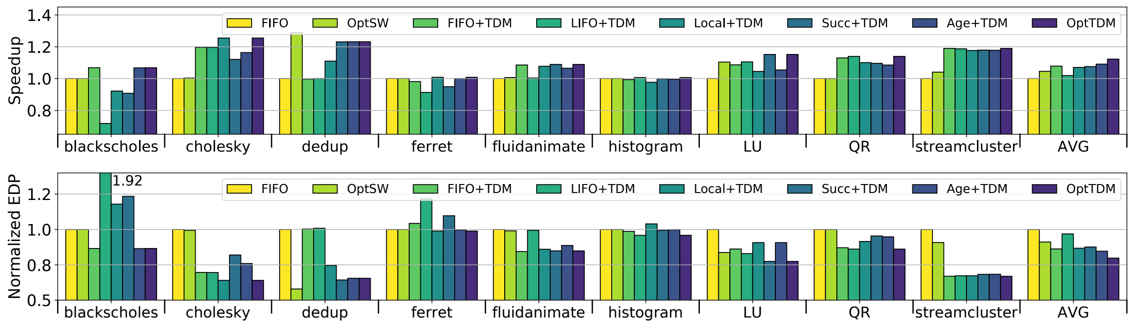


Figure 5.11: Speedup (top) and EDP reduction (bottom) with FIFO, LIFO, Locality-aware and Criticality-aware schedulers using software runtime system and TDM. Results are normalized to the software runtime system with a FIFO scheduler.

5.4 Flexible Scheduling with TDM

This section illustrates the synergy of TDM with different software schedulers that exploit the characteristics of the applications to improve performance and power consumption.

Five schedulers are used in the experiments: *First-In First-Out (FIFO)* schedules tasks in the same order as they become ready. *Last-In First-Out (LIFO)* schedules first the last task that has become ready. *Locality* scheduler exploits data locality and assigns tasks to cores aiming to minimize data movements. When a task finishes executing on a core and some of its successor tasks is ready, a successor is executed on the core. If no successors are ready the first task in the ready queue is scheduled. *Successor* scheduler counts the number of successors of a task. If this number is above a threshold it is placed in a high priority ready queue, otherwise it is placed in a low priority ready queue. Threads first check the high priority ready queue and, if it is empty, they look for tasks in the low priority ready queue. *Age* scheduler sorts tasks in the ready queue by their creation time, so older tasks have higher priority than younger ones.

These schedulers can be used with TDM without any modification. The runtime system communicates with the DMU at task creation and finalization phases, and requests all the tasks that have become ready after a task finishes. The schedulers organize the ready tasks in software data structures and ready queues that implement the different policies. TDM reduces task creation overheads and, consequently, all schedulers benefit from this architectural support.

5.4.1 Performance Evaluation

We evaluate the performance of the different software schedulers when they are deployed by an entirely software-based runtime system and when they are combined with TDM. For each application we select the best scheduler with and without TDM, denoted OptTDM and OptSW, respectively. Figure 5.11 shows the speedups of OptSW, FIFO+TDM, LIFO+TDM, Locality+TDM, Successor+TDM, Age+TDM and OptTDM policies over a FIFO scheduler without hardware support. The geometric mean of the speedups is also reported (AVG).

In general, FIFO and LIFO schedulers show similar performance except for Blackscholes, which is parallelized with 64 independent chains of dependent tasks. With FIFO, all independent chains progress at the same pace, while with LIFO, 32 chains (one per core) progress much faster than the others, leading to a significant load imbalance and 29.3% performance degradation. A similar situation happens with Locality+TDM and Successor+TDM, although performance only degrades 7.8% and 9.2%, respectively.

TDM significantly reduces the task dependence management overheads in Cholesky, as reported in Figure 5.9. The locality scheduler further improves performance, as this is a memory intensive application that reads blocks of a dense matrix from memory. Thus, Cholesky is sensitive to data locality, and Local+TDM outperforms FIFO+TDM by 4.2%.

Priority schedulers (Successor and Age) achieve important improvements in benchmarks with a clear critical path in the TDG. Dedup has many compute-intensive tasks and each one of them is followed by a long I/O-intensive task. I/O tasks cannot be executed in parallel, which is enforced by means of control dependences between them, so overlapping I/O with compute tasks maximizes parallelism. Successor+TDM achieves this overlap, as I/O and compute tasks have the same priority (all tasks have 1 successor), and yields a 23.2% performance improvement. FIFO prioritizes compute tasks because they become ready before their I/O counterparts, so it fails in overlapping I/O and computation. However, the successor scheduler harms performance in Cholesky, as it delays the execution of tasks that process the borders of the matrix, limiting the available parallelism.

Overall, OptSW performs worse than TDM with any scheduler, while the best scheduler (Age+TDM) achieves an average 9.1% speedup. More importantly, the best performance is achieved with FIFO+TDM, LIFO+TDM, Locality+TDM, Successor+TDM, and Age+TDM for 2, 2, 2, 2, and 1 different benchmarks, respectively.

When the best scheduler per application is used, average 4.5% and 12.2% performance improvements are obtained with OptSW and Opt+TDM, respectively. The benefits of TDM are demonstrated by two facts: first, TDM provides enhanced results for all the schedulers and, second, TDM exposes the scheduler policy to the software, which yields large performance benefits due to the flexibility it provides.

5.4.2 Energy Efficiency

This section evaluates the energy efficiency of TDM combined with different schedulers. The bottom chart of Figure 5.11 shows the EDP of FIFO, LIFO, Locality, Successor and Age schedulers when combined with TDM. This figure considers the power introduced by the DMU hardware structures. Results are normalized to a pure software runtime system with a FIFO scheduler, and a geometric mean (AVG) of the results is shown.

Figure 5.11 shows that TDM provides significant EDP reductions in seven benchmarks, and minimal reductions are obtained in Ferret and Histogram. On average, EDP is reduced up to 8.9% with the best software solution (OptSW), while EDP is reduced between 3.1% and 15.4% when combining different schedulers with TDM. Combining TDM with the best scheduler per application (OptTDM) yields the best results, achieving average reductions in EDP of 20.3%.

In terms of power consumption, the DMU consumes a negligible fraction of the total power, less than 0.01%. All benchmarks consume very similar power with the considered schedulers on a software runtime system and when they combine the schedulers with TDM (less than 1.0% difference). In addition, average power results show less than 1.0% variation between different schedulers. Since the average power consumption does not significantly change, the improvements in total energy to solution follow the same trends as Figure 5.11.

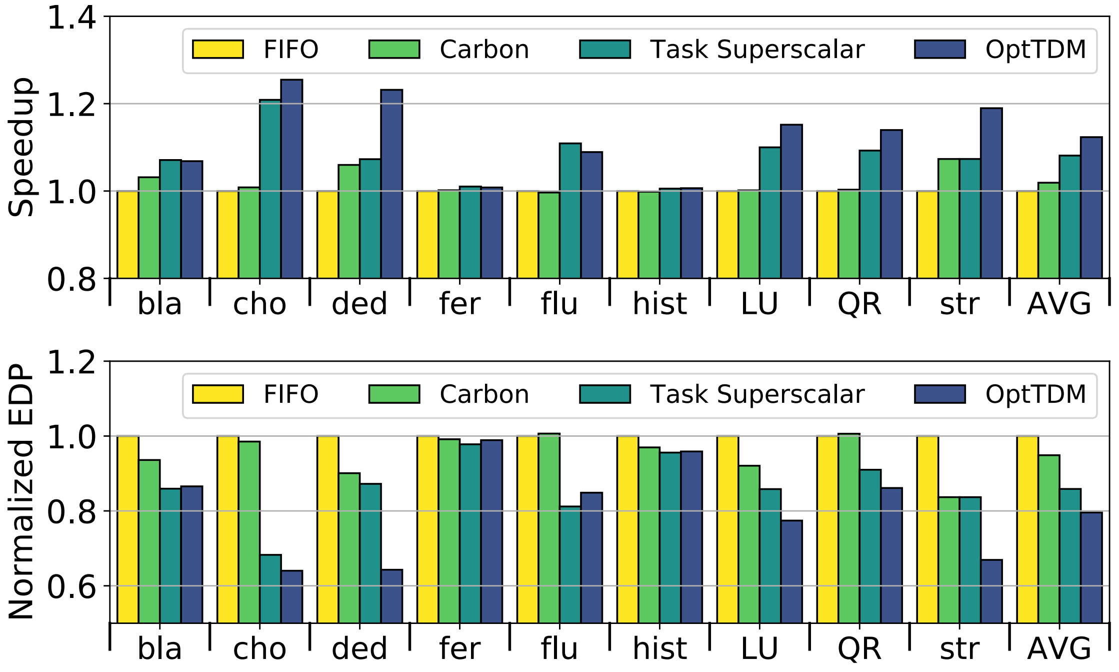


Figure 5.12: Speedup (top) and EDP reduction (bottom) of Carbon, Task Superscalar and TDM over a software runtime system with FIFO scheduler.

5.4.3 Comparison with Other Proposals

This section compares TDM with two alternative hardware support proposals for the runtime system. Carbon [71] implements the task scheduler at the hardware level and task dependence management is done in software by the runtime system so, conceptually, it is the opposite to TDM. Carbon provides ISA instructions that allow threads to add and request ready tasks, and the hardware support consists of a set of distributed hardware queues to keep ready tasks and a fixed FIFO scheduling policy with work stealing. Task Superscalar [39] offloads all the runtime system activities to the architecture, including task dependence management and task scheduling with a fixed FIFO policy. It uses an interface similar to Carbon, and its hardware support consists of a gateway, a ready queue, and distributed tables to track tasks and dependences. As explained in Section 5.1, thread synchronizations overheads are negligible in our experiments, so proposals that accelerate thread synchronization such as ADM [107] are not included in the study.

The top chart of Figure 5.12 presents the speedup of Carbon, Task Superscalar and TDM over a software runtime system with a FIFO scheduler. TDM makes use of the best scheduling policy per benchmark found in the previous section, and the geometric mean of the results is also presented.

Carbon improves performance in Blackscholes, Dedup and Streamcluster, reaching speedups of up to 7.3%. In the rest of benchmarks its impact is negligible because, as shown in Figure 5.1, the time spent in scheduling phases is very low, while tracking task dependences is much more costly. As a result, Carbon obtains a modest average speedup of 1.9%.

Task Superscalar performs both task scheduling and dependence management in hardware. This approach provides significant speedups in several benchmarks, reaching an average 8.1% speedup. TDM achieves similar reductions in runtime system overheads and further improves performance by allowing flexible software schedulers, achieving an average speedup of 12.3% and clearly qualifying as the best option. The advantage of TDM is particularly significant in cases where using the appropriate scheduling policy is fundamental to increase the parallelism, as in Dedup, where TDM improves performance by 23.1% while Carbon and Task Superscalar just reach 5.9% and 7.2%, respectively.

The bottom chart of Figure 5.12 shows the EDP of Carbon, Task Superscalar and TDM normalized to the baseline software solution with a FIFO scheduler. Results consider the extra power consumption added by the hardware structures of TDM, Carbon and Task Superscalar. Important EDP reductions are obtained in seven of the benchmarks, while more modest EDP reductions are obtained in the remaining two benchmarks. On average, TDM reduces EDP by 20.4% while Carbon and Task Superscalar only achieve reductions of 5.1% and 14.1%, respectively.

Regarding hardware complexity, TDM lays between Carbon (simple hardware queues) and Task Superscalar. Table 5.2 shows that the DMU requires 105.25KB for the selected configuration. For the same configuration in terms of number of in-flight tasks and dependences, Task Superscalar requires 769KB: a 1KB Gateway, a 256KB TRS (2048 entries$\times$128B), a 256KB ORT (2048 entries$\times$128B), and a 256KB Ready Queue (2048 entries$\times$128B). The cost of the OVT is not taken into account, as the DMU does not perform dependence renaming. In addition, Task Superscalar requires more power-hungry CAM look-ups than the DMU. Alltogether, the DMU requires 7.3$\times$ lower hardware complexity than Task Superscalar.

Finally, another solution is to add an extra core devoted to the runtime system. We observe that the performance of a 33-core system with a pure software runtime system improves marginally, 0.8% on average. In the 32-core baseline task creation is already executed by one thread running on a core, so the extra core just adds one more worker thread and has no impact on dependence tracking overheads.

5.5 Remarks

ATaP models are very appealing for large-scale multi-cores. A key aspect of task-parallel programs is the task granularity, which determines the potential to exploit the available parallelism and to ensure load balancing, but also dictates the runtime system overheads.

This thesis proposes TDM, a hardware/software co-designed mechanism to accelerate task dependence management operations while allowing flexible task scheduling in software. Unlike previous works with schedulers fixed in the architecture, the separation of concerns in TDM provides high degrees of flexibility, adaptability and composability, which are key in modern computing infrastructures with multiple sockets and off-chip accelerators, and also allows to capitalize on the benefits that different scheduling policies provide for certain applications and environments. In addition, the architectural support of TDM includes novel techniques that maximize efficiency, such as renaming IDs to reduce the storage requirements or leveraging the size of the dependences to avoid conflicts in the hardware structures when the lower bits of the dependence addresses are equal.

As a result, TDM outperforms software runtime systems by an average 12.3% while reducing EDP by 20.4%. Compared to pure hardware solutions, TDM achieves an average speedup of 4.2% with 7.3$\times$ lower hardware complexity.

Chapter 6

Easing Communication Bottlenecks Through The Runtime System

We have observed that inefficient interactions between ATaP models and MPI limit the achievable computation-communication overlap and negatively impact the performance of large parallel programs.

To address these issues, we present two mechanisms, similar to the MPI tools interface (MPI_T) [46], for exchanging information between MPI and an ATaP runtime system and analyze their trade-offs: 1) A fast mechanism to poll events when idle using a lock-free queue, and 2) a delivery solution based on callbacks that can benefit from a hardware implementation, shown in the bottom row of Figure 6.1. These mechanisms allow ATaP runtimes to seamlessly interoperate with MPI by reducing or completely eliminating the need to rely on explicit polling or waiting on specific requests, and instead deliberately invoking the progress engine only when needed, driven by runtime events.

- We present a novel approach to optimize interactions between an ATaP runtime system and MPI by exploiting knowledge of MPI internal events.

- We expose new opportunities to overlap MPI collectives with tasks that depend on partially received collective data.

- We present a detailed evaluation of the proposed ideas using MPI and OmpSs [38], an ATaP model that follows the semantics of OpenMP 4.0 tasks. When compared to state-of-the-art solutions with task-based communications and dedicated communication threads, we show improvements of up to 16.3% and 34.5% for proxy applications with point-to-point and collective communication, respectively.

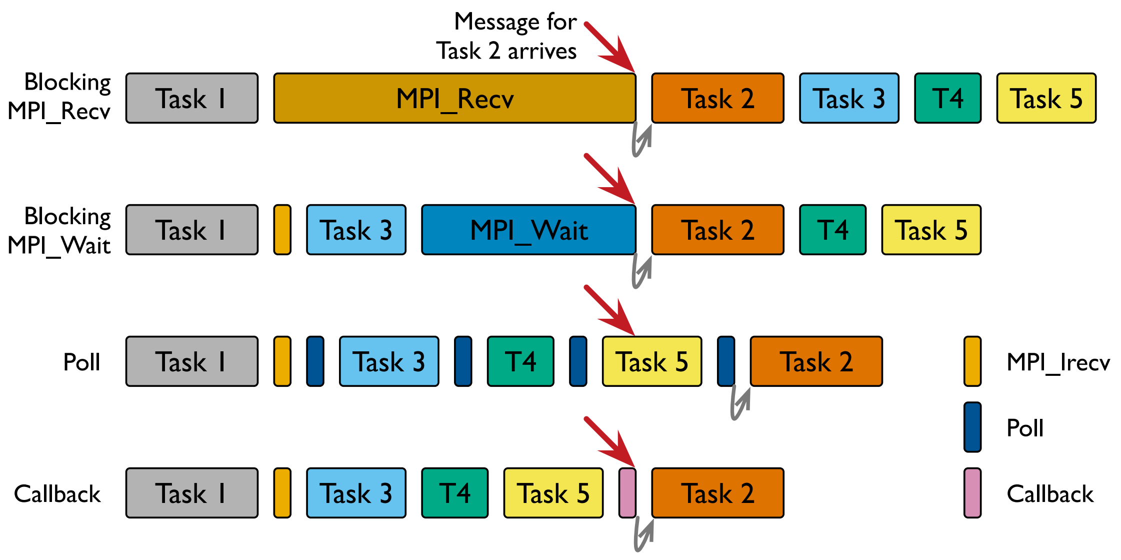


Figure 6.1: Early invocation of a blocking `MPI_Recv` by one task can prevent other tasks from making progress (top row). The remaining three rows represent alternatives. In each case, the red arrow marks the arrival of an MPI message in time for a specific task (Task 2).

6.1 Bottlenecks on taskified communication

Let us consider several common mechanisms used in different ATaP models for making MPI calls from within tasks. A first mechanism is for tasks to make a blocking MPI call, such as `MPI_Recv`. This prevents other tasks from using the idle core while the task in question is blocked waiting for messages and is clearly inefficient as shown in the top row of Figure 6.1. The second mechanism, as a potential solution to the above problem, is to use a non-blocking `MPI_Irecv` and `MPI_Wait`. However, this still has the problem of being blocked at the `MPI_Wait` if it is called too early. A third mechanism is to periodically poll for message arrivals; this avoids blocking but can require multiple trials. Thus, none of these mechanisms is perfect and they all waste valuable CPU resources. Despite these issues, the use of MPI as the underlying communication mechanism for ATaP models is attractive, since it represents a convenient portability layer that is available on virtually every high performance computing platform.

If the runtime system of an ATaP model is aware of the progress or state of communication in MPI, it can make better scheduling and task-creation decisions, and efficiently overlap computation and communication. Our approach tracks certain events in the MPI layer and exposes them to the ATaP runtime system in order to efficiently schedule block-

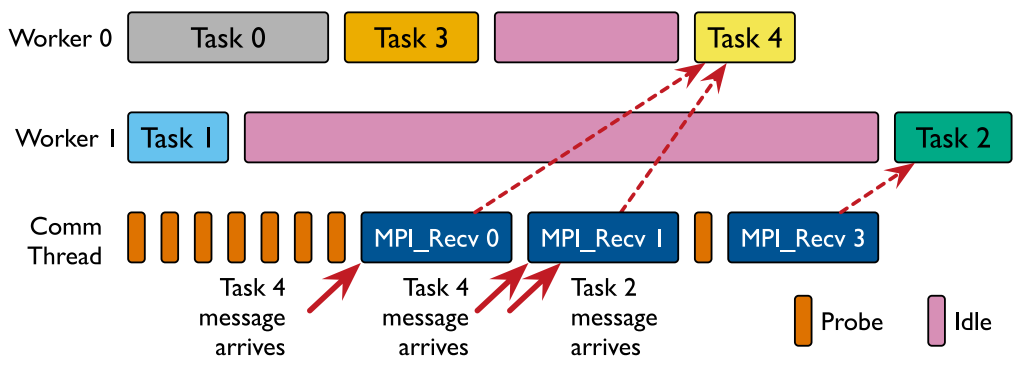


Figure 6.2: Communication thread can become a serial bottleneck if one thread is responsible for many workers.

ing MPI primitives or initiate a specific request to poll. Further, using our approach, the runtime system can execute tasks that utilize partially received data of an on-going MPI collective operation, thus providing opportunities for computation-communication overlap that were not exposed previously.

In both explicit and implicit styles of communication, some inefficiencies prevent the interaction between MPI and ATaP models from reaching its full potential. ATaP models seek to exploit asynchrony and to overlap communication with computation across tasks to improve performance. In contrast, several MPI features have traditionally been influenced by a bulk synchronous programming model, in which communication and computation occur in phases. This results in performance inefficiencies such as those shown in the top two rows of Figure 6.1 – resources can remain idle if blocking calls are made early before the messages have arrived.

ATaP models typically deploy communication threads to improve computation-communication overlap. A dedicated thread is made responsible for data transfers in order to avoid blocking the worker threads. However, communication threads do not execute computation tasks, which results in resource under-utilization if the thread is assigned a dedicated core. If communication threads are not assigned dedicated cores, they can perform poorly. They can also become a serial bottleneck, as shown in Figure 6.2, in which the communication thread is responsible for sending, probing, and receiving messages for all workers. In this example, worker 1 is idle for a long time because the communication thread is busy processing messages for task 4 of worker 0.

Another potential source of inefficiency is the implicit global synchronization that MPI collectives impose. While it is possible to overlap computation and collective communication to a certain extent by using non-blocking collectives [51] as standardized with

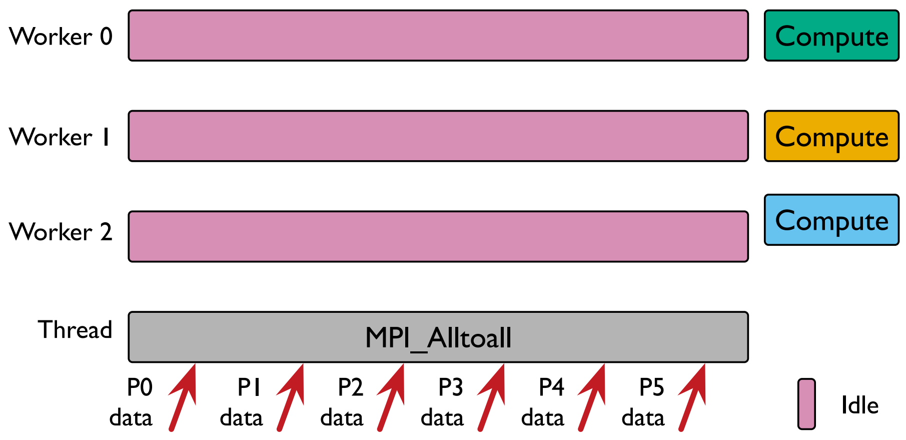


Figure 6.3: Tasks that can begin computation with partial data (P0, P1 -> worker 0; P2, P3 -> worker 1; P4, P5 -> worker 2) need to wait for the collective to complete.

MPI 3.0, it is currently not possible to use partially received data while the collective is in progress. For example, in the `MPI_Alltoall` operation, although data from some tasks/processes is received earlier than others, execution of tasks that only need that partial data cannot begin until the collective completes as shown in Figure 6.3. In this example, although each worker depends only on data from two processes (P0, P1 -> worker 0; P2, P3 -> worker 1; P4, P5 -> worker 2), all workers have to wait until data from all processes is received and the `MPI_Alltoall` call completes.

The proposals described in this chapter aim at removing the inefficiencies that undermine the cooperation between ATaP runtime systems and distributed memory messaging libraries such as MPI. By exposing information about events happening at the communication layer (e.g., incoming messages, data transfer completions or collective operations progress) to an ATaP runtime, we believe the runtime system can schedule tasks more efficiently and reduce idle time, maximizing computation-communication overlap as a result. Section 6.2 describes the interface we propose to achieve these goals.

6.2 Exposing MPI Activity to ATaP Runtimes

We propose to make ATaP runtime systems aware of MPI activity to overcome the performance issues highlighted in Section 6.1. Further, we propose to drive this information exchange by triggering callbacks in hardware or via lower-level system software. This section describes the proposed interactions between MPI and ATaP runtimes, and a mechanism to enable computation-communication overlap in the case of collective communications.

6.2.1 Extending MPI to Support Event Handling

Our approach exposes a set of events that can be triggered by a MPI implementation to ATaP runtime systems. In order to stay consistent with the MPI standard, and to enable future possible standardization efforts, we build on existing concepts, in particular MPI_T, the MPI Tool Information interface introduced in MPI 3.0 [42], as well as the currently proposed MPI_T_Events extensions [46]. The latter provides the necessary infrastructure for callbacks in MPI, intended for the support of tracing tools, but does not define any concrete events, matching the philosophy of MPI_T. In particular, we propose adding the following events to MPI:

- `MPI_INCOMING_PTP` signals the arrival of a point-to-point message. It saves the tag and source of the message, and the associated `MPI_Request` handle, if any. For a message expected to use the rendezvous protocol, this event may indicate the arrival of the control message.
- `MPI_OUTGOING_PTP` signals the completion of a non-blocking point-to-point send operation. It saves the `MPI_Request` handle.
- `MPI_COLLECTIVE_PARTIAL_INCOMING` signals the arrival of some data in the context of a collective communication. It saves the MPI source rank relative to the communicator being used.
- `MPI_COLLECTIVE_PARTIAL_OUTGOING` signals the sending of some data in the context of a collective communication. It saves the MPI rank of the receiver. When this event is triggered, it is safe to overwrite the corresponding portion of the outgoing buffer.

We implement these events in the context of MVAPICH 2.2 [121] implementation on Intel platforms, where PSM2 [55] is primarily responsible for conducting point-to-point communication. Thus, in our implementation, events such as the detection of an incoming point-to-point message originate at the PSM2 layer, which in turn notifies MPI of the associated point-to-point event. PSM2 uses lightweight helper threads to handle communication, which efficiently share resources with other threads in the system. Event notification to MPI is triggered by these helper threads. On the other hand, the creation of events associated with the progress of collective communication is still handled by MPI. In both cases, MPI is responsible for the delivery of the events to the ATaP runtime system.

6.2.2 Mechanisms for Event Delivery

We consider two mechanisms to deliver the events described in Section 6.2.1 to the ATaP runtime: a polling-based mechanism and a callback-based approach.

6.2.2.1 Polling-based Notification

Our first approach is to add a polling interface to MPI, or more specifically to the MPI_T Events proposal, which an ATaP thread can use to query events at its convenience. When invoked, the polling call checks if an event has occurred and, if so, returns the data associated with the event. This is significantly different from the polling capabilities currently available in MPI through the set of `MPI_Test` calls. These only return the state of a specific request, which is inefficient as it requires users to individually poll on all outstanding requests until one of them is in a desired state. In contrast, our approach only returns completed events across all event sources, and therefore prevents unnecessary queries. Conceptually, our polling interface can be viewed as an extension of `MPI_Probe`: in addition to information about the incoming message that `MPI_Probe` returns, our extension can also return information regarding events related to non-blocking send/receive requests and collectives.

We propose one additional function that implements the actual polling: `MPI_T_Event_poll (MPI_T_event* event)`. This function indicates on return whether one of our defined events has occurred since its last invocation and, if yes, returns an opaque event object containing information regarding the event. The type of this object is identical to the event object type used in the MPI_T_Events proposal, and hence can be decoded with a matching call to `MPI_T_Event_read`. In our implementation, a lock-free event queue, from the C++ Boost library [19], is used to store the events until they are consumed by the ATaP runtime system. We also modify the specific runtime used in this chapter, Nanos++, to use its worker threads to invoke the polling interface. These invocations are done either between consecutive task executions or when worker threads are idle. Figure 6.4 summarizes the polling-based delivery mechanism described in this section.

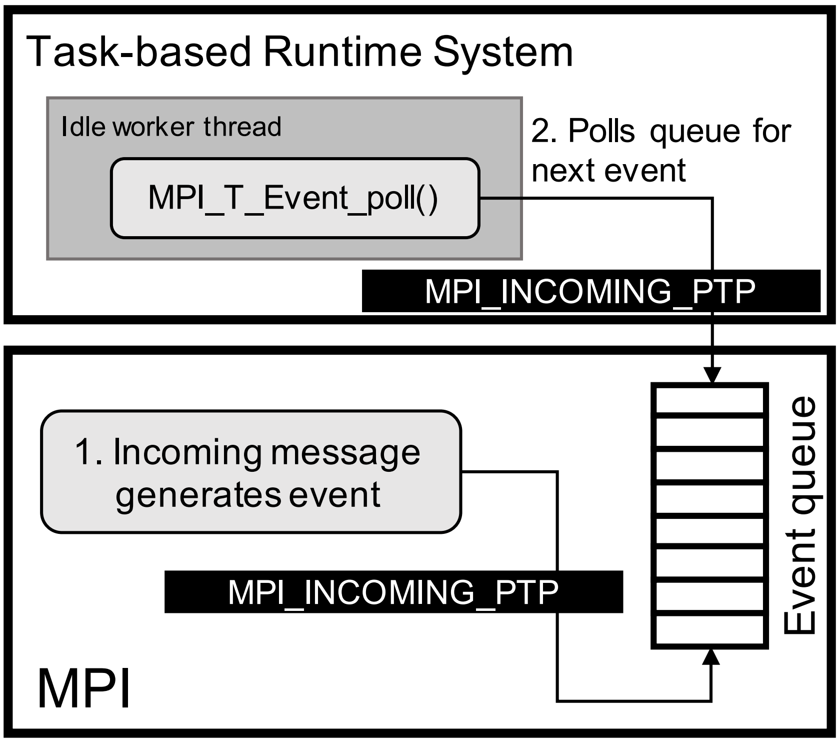

Task-based Runtime System
Idle worker thread
MPI_T_Event_poll()
2. Polls queue for next event
MPI_INCOMING_PTP
1. Incoming message generates event
MPI_INCOMING_PTP
Event queue
MPI

Figure 6.4: MPI_T event creation, addition to the event queue, and consumption by a worker thread in polling mode.

6.2.2.2 Callback-based Notification

Callback-based notification works by associating handler functions with specific events in a way that the corresponding handler function is invoked to perform some action once the event occurs. In particular, by having the events described in Section 6.2.1 handled by callbacks, we release the ATaP runtime system from the need for polling the event queue. For this functionality, we directly rely on the MPI_T_Events proposal [46], which provides generic callbacks mainly intended to implement tracing tools. For our purpose, we use it to support the events described above and offer them to the user, in our case the ATaP runtime, which can then associate a handler function by invoking the *MPI_T_Event_handle_alloc* call, as described by Hermanns et al. [46]. Once invoked, the runtime receives an MPI events object, which can be decoded with *MPI_T_Event_read*.

The primary concern with the use of callbacks is the impossibility of knowing beforehand when an event will occur, which makes it impossible to know which thread will handle the event and execute the callback. Thus, to ensure correctness, the implementation of any callback handler must respect some restrictions:

- Callbacks should not take any locks that may already be taken by the thread executing the callback.
- Callbacks have restricted MPI capabilities as described in [46].
- Callbacks should not be nested.

These restrictions are not a problem in our context. The primary purpose of event notification is to satisfy an event dependence for a task in the system and, once all dependences for the task are met, push it to the scheduler. These actions just require locks that maintain the state of runtime system metadata as well as locks that control the interaction with the scheduler queue. Neither of these locks can be taken by a worker thread if it executes the callback while invoking MPI inside a task. Similarly, a worker thread does not hold any lock if it executes the callback while invoking MPI when idle to progress communication. If callbacks are invoked by MPI helper threads, no runtime system locks will be taken by them. These actions also do not require any calls to MPI and thus cannot invoke other callbacks. Thus, inside the ATaP runtime, we use callbacks to identify, unlock and push ready tasks to the scheduler.

Hardware-induced callbacks: Callbacks can also be triggered as user-level interrupts by the Network Interface Card (NIC) when it detects associated MPI events. Having the NIC detecting events and triggering callbacks can improve both flexibility and performance by providing a more accurate and early notification of an incoming message, message delivery completion, and RDMA operations. For reference, Keppitiyagama et al. [69] showed that Myrinet NIC hardware, which included a programmable network processor, could drive the MPI progress engine, and networks like SCI (The Scalable Coherent Interface) included remote doorbell capabilities. In addition, some of the current Intel Xeon processors already integrate the OmniPath NIC in the processor package. We hope that this work will lead to the addition of such capabilities in hardware. In this chapter, we emulate this capability by using a hidden thread running on a dedicated core to monitor MPI state.

6.2.3 Changes to the OmpSs runtime

Typical implementations of ATaP models such as OmpSs may indicate to the underlying runtime that a task performs/depends on communication, but do not require exposing more details. For the ATaP runtime system to match tasks with notifications, more information is required. To this end, we extend our ATaP model, OmpSs, to notify its

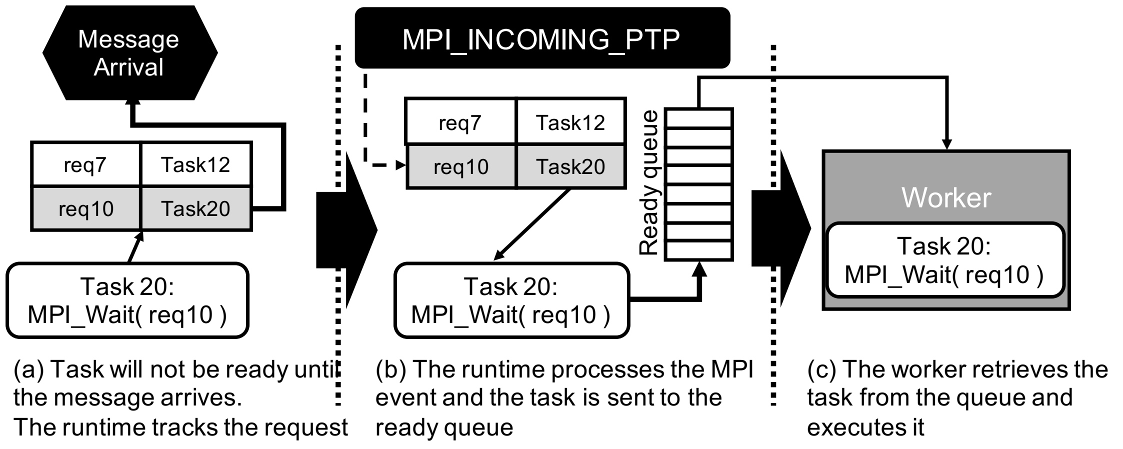

Message Arrival
req7
Task12
req10
Task20
Task 20: MPI_Wait(req10)
(a) Task will not be ready until the message arrives. The runtime tracks the request
MPI_INCOMING_PTP
req7
Task12
req10
Task20
Ready queue
Task 20: MPI_Wait(req10)
(b) The runtime processes the MPI event and the task is sent to the ready queue
Worker
Task 20: MPI_Wait(req10)
(c) The worker retrieves the task from the queue and executes it

Figure 6.5: The runtime creates dependences between tasks and corresponding MPI_T events and the notification of an MPI_T event unlocks a task waiting for it.

underlying runtime, Nanos++, of messages being sent or received, as well as of MPI requests that are accessed in a task. This information is used to create a task dependence on the corresponding event. In our implementation, MPI calls inside tasks are identified by the OmpSs compiler, which introduces code to inform Nanos++ of the MPI call and its arguments such as source/destination rank and `MPI_Request` object.

Enabled by this communication-related information, the Nanos++ runtime system creates dependences between tasks and their corresponding MPI_T events. This implies, for example, that a task performing a blocking MPI call is not allowed to run until the corresponding MPI_T event, `MPI_INCO- MING_PTP`, related to the task has taken place and received by Nanos++. Similarly, a task invoking a blocking `MPI_Wait` is not allowed to execute until the completion event associated with the incoming or outgoing message request takes place (Figure 6.5 (a)).

When an event is delivered to Nanos++, either via polling or execution of a callback, it is used to unlock the associated task for execution as depicted in Figure 6.5 (b). For every task with an event dependence, Nanos++ contains an entry in a reverse look-up table based on the identifiers (message tag, source, or the `MPI_Request` object). This table is used to identify the task, which is then scheduled for execution if all its dependences are met. In this way, by waiting for communication events to occur before the tasks are scheduled, we are able to avoid unnecessary blocking of worker threads.

Even under this scheme, the use of blocking MPI calls for sending or receiving can still block threads unnecessarily when a rendezvous protocol is deployed in MPI. For example, a receiving task is unlocked upon the arrival of the control message of the senderinitiated rendezvous message. Following this, the task will be active and the thread will be blocked for the time the message data is transferred over the network from the source. We recommend that, in such situations, non-blocking send/receive should be used in the task, and another task with an associated `MPI_Wait` should be marked for execution when the actual data arrives.

6.2.4 Overlapping Computation with Collectives

So far, we have discussed exposing point-to-point communication related MPI events to an ATaP runtime system to potentially increase its responsiveness and reduce the time for which threads remain blocked. We now switch our attention to exploiting computation-communication overlap for MPI collectives. MPI 3.0 introduced non-blocking collectives, akin to non-blocking point-to-point operations, to allow programmers to perform computation while the collective progresses in the background. However, like point-to-point operations, this can be restrictive due to the need for a wait or frequent test calls.

Traditionally, an MPI collective call is viewed as a monolithic operation whose intermediate progress is not exposed to the programmer. We propose to notify ATaP runtime systems of partially received data so that they can trigger dependent computations as early as possible and thus maximize computation-communication overlap. In particular, we focus on many-to-one or many-to-many style collectives such as `MPI_Alltoall`, `MPI_Gather`, and `MPI_Allgather`, in which tasks can be unlocked even when partial data has been received.

Figure 6.6 shows an example in which an all-to-all operation is used to share data among tasks on different nodes. Tasks are created in this example such that some tasks need only part of the data received during the collective to start execution. For example, the left task depends only on data received from the MPI process 0, while the right task depends on data from MPI process 1. With the existing MPI semantics, all these tasks will be unlocked and ready for execution when the entire collective completes and data from all processes is received. However, as several collectives in MPI are typically implemented using point-to-point communication, it is likely that data required to unlock one of these tasks is received earlier than the remaining data. Thus it is potentially possible to start the execution of one task earlier, as shown in the figure.

In order to enable computation overlap with collectives, we add two events:

- `MPI_COLLECTIVE_PARTIAL_INCOMING`

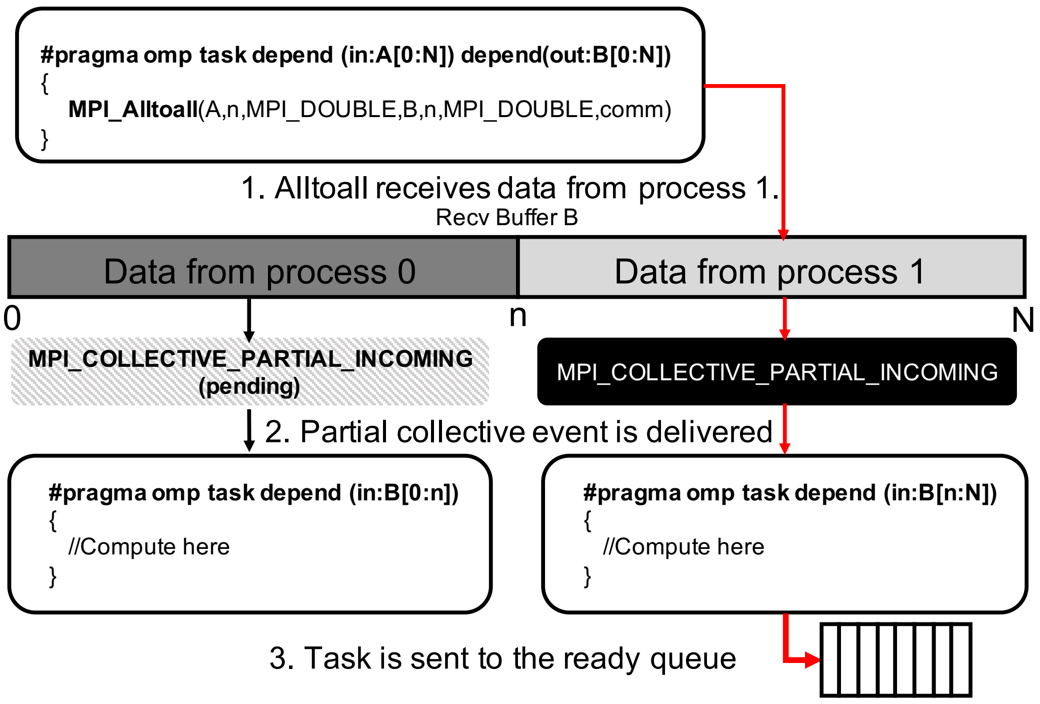

#pragma omp task depend (in:A[0:N]) depend(out:B[0:N])
{
MPI_Alltoall(A,n,MPI_DOUBLE,B,n,MPI_DOUBLE,comm)
}
1. Alltoall receives data from process 1.
Recv Buffer B
Data from process 0
Data from process 1
0
n
N
MPI_COLLECTIVE_PARTIAL_INCOMING (pending)
MPI_COLLECTIVE_PARTIAL_INCOMING
2. Partial collective event is delivered
#pragma omp task depend (in:B[0:n])
{
//Compute here
}
#pragma omp task depend (in:B[n:N])
{
//Compute here
}
3. Task is sent to the ready queue

Figure 6.6: Unlocking of the right task when partial data from process 1 is received leading to a collective callback.

- `MPI_COLLECTIVE_PARTIAL_OUTGOING`.

MPI can use these events to notify the runtime when part of the data expected in the collective has arrived or has been sent, respectively.

The runtime system is already aware of the memory locations that a task reads or writes, as they are specified in the task creation pragma (see examples in Section 2.3.1 and Figure 6.6). With our extensions of Section 6.2.3, the runtime system also knows the send/receive locations and the volume of the data sent/received in the collectives. Hence, when the `MPI_COLLECTIVE_PARTIAL_*` event arrives, the runtime system matches the partial data received with the task that depends on it, and if all its dependences are satisfied, executes it without waiting for the collective to finish. Non-blocking collectives can also benefit from this approach, since even for them, there is no existing mechanism to signal when it is safe to use partial data.

6.3 Performance Evaluation

This section evaluates the performance impact of using our proposed MPI events to drive the task execution in the OmpSs programming model, as described above.

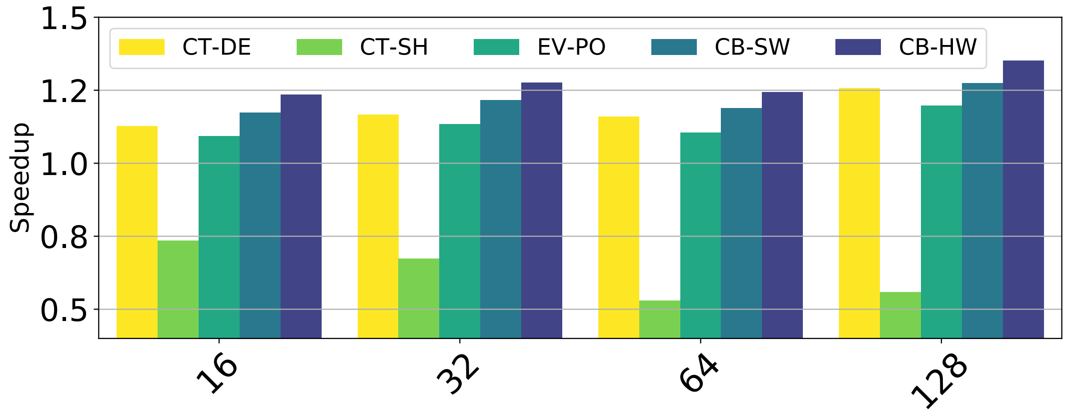
6.3.1 Results for Point-to-point Benchmarks

We compare the performance of baseline task-based executions of HPCG and MiniFE with executions in five other resource-equivalent scenarios. These executions are performed on 16, 32, 64 and 128 nodes; every node contains four MPI processes, each of which can use 8 cores which is the configuration that minimizes the execution time for the baseline implementation. In the baseline scenario, eight worker threads per MPI process are responsible for executing the computation as well as the communication tasks and for invoking the MPI progress engine. It is worth mentioning that this is the only available OmpSs+MPI or OpenMP 4.0+MPI out-of-the-box configuration with task-based communication. We present two scenarios with communication threads: one in which we add a communication thread shares hardware with worker threads on available cores, i.e., 8 worker threads and 1 communication thread share 8 cores (CT-SH), and the other in which a dedicated core is assigned to the communication thread and the computation tasks are executed on the remaining 7 cores by 7 worker threads per MPI process (CT-DE). Such solutions represent the state-of-the-art communication model of most common ATaP models. However, as it was mentioned before, OmpSs and OpenMP does not integrate a communication thread in their available releases so we hope this work can motivate the default inclusion of such configuration for hybrid applications.

We then compare three scenarios covering variants of our proposals against these baselines: i) polling-based notification (EV-PO), where worker threads poll for MPI_T events when idle (Section 6.2.2.1); ii) callback-based event delivery in software (CB-SW); and iii) a hardware-induced callback-based event delivery (Section 6.2.2.2). The hardware support is emulated by using an additional hidden core that monitors the internal status of MPI and PSM2 to trigger the callbacks; this core never executes a task.

Figure 6.7 (a) presents the speedups obtained with the scenarios described above normalized to the baseline for HPCG. Use of a dedicated core for the communication thread (CT-DE) provides a speedup ranging from 12.7% to 25.7% with respect to the baseline approach for the 16 to 128 node configurations respectively. This improvements are due to the early execution of communication tasks enabled by CT-DE as well as due to the avoidance of blocking in worker threads. On the other hand, when the communication thread is not assigned a dedicated core (CT-SH), performance degrades by 26.5%.

The polling-based event notification (EV-PO) mechanism yields slightly lower performance improvements than CT-DE (9.25%, 13.5%, 10.5% and 19.7%). This is caused



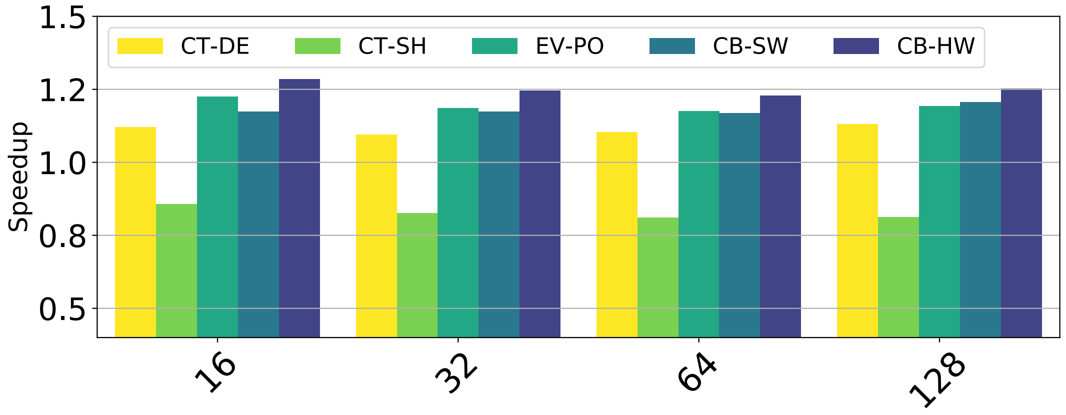


Figure 6.7: Speedup for (a) HPCG and (b) MiniFE over a baseline task-based implementation on 16, 32, 64 and 128 nodes with 64, 128, 256 and 512 MPI processes, each with 8 threads (cores).

by computation tasks in HPCG delaying the polling for MPI events, and thus delaying communication. Hence, benefits of event notifications are sometimes neutralized by the lack of progress. With software callbacks we are able to unlock the tasks as soon as the events arrive and performance improvements rise to 17.4%, 21.7%, 19.0% and 27.4% respectively. In addition, hardware-based support for event detection and triggering of callbacks (CB-HW) is able to overcome the delays due to long running computation tasks and improves performance by 23.5% 27.6%, 24.3% and 35.2% (up to 9.5% over CT-DE). In CB-HW, as soon as an MPI_T event occurs, the associated task becomes ready for execution. Finally, the small granularity of the tasks doing the pre-conditioning steps of the matrix require communication to be done as early as possible, improving the performance over the baseline of the proposed solutions as the node count increases. The time spent in communication for HPCG is approximately a 10.7% of the total execution time executing MPI calls without event notification. This time is reduced to a 3.6% when using callbacks as the delivery mechanism leading. This is due to MPI calls being only invoked when the associated event has arrived, and effectively minimizing waiting time. Moreover, as only ready MPI calls are only executed, the time that was spend on checking the status of the MPI requests is now devoted to computation leading to the aforementioned speedups.

Figure 6.7 (b) shows that we observe similar trends for MiniFE as for HPCG. The key difference is that due to the smaller granularity of computation tasks in MiniFE, the polling-based event notification (EV-PO) is able to outperform the scenario with a dedicated communication thread (CT-DE). Aided by the relatively shorter delays for polling, the improvements for EV-PO are 22.5%, 18.6%, 17.5% and 19.2% in comparison to 12.2%, 9.5%, 10.3% and 13.0%. As was the case for HPCG, the presence of hardware support for callbacks further improves the performance and achieves speedups of 28.4%, 24.6%, 22.8% and 25.2% over the baseline and up to 16.3% over CT-DE. The lack of a pre-conditioner in MiniFE yields a lower communication frequency than HPCG which does up to 11 neighbor communications per iteration while MiniFE only does one. This lower communication/computation ratio translates into a constant scalability over the baseline for all the node counts in contrast to HPCG where the baseline performance degrades as the communication needs increase with the node counts. Finally, similarly to HPCG the time spent in communication is a 11.8% of the total execution time with a reduction up to a 3.3%.

Regarding the overheads of the polling and the callbacks based approaches, the average time spent polling for events is 9x and 15x that of callback for MiniFE and HPCG respectively, with polling happening 100x more times than callbacks in both benchmarks.

6.3.2 Results for Collective Benchmarks

This section evaluates the performance impact of the mechanism described in Section 6.2.4 for exposing the potential overlap of computation tasks with MPI collectives.

6.3.2.1 Fast Fourier Transform

Similar to the point-to-point benchmarks, we compare the performance of 2D FFT and 3D FFT benchmarks for executions in the same five resource-equivalent scenarios and present speedup against the baseline execution on 128 nodes. For 2D FFT, we use square matrices of input sizes ranging from 16384×16384 to 262144×262144, and cubic volumes of sizes 1024^3, 2048^3, and 4096^3 for 3D FFT. Since our experimental results did not show significant performance differences between the three event-based scenarios

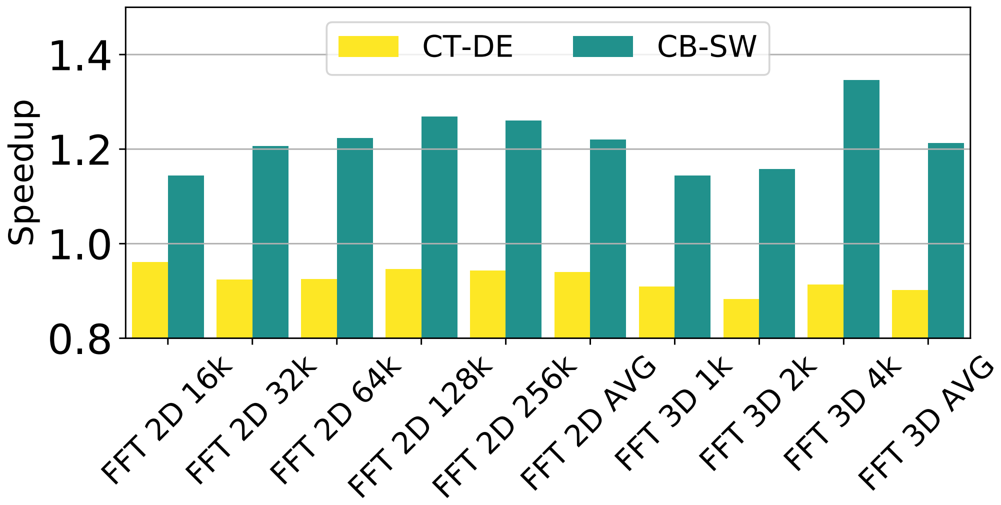
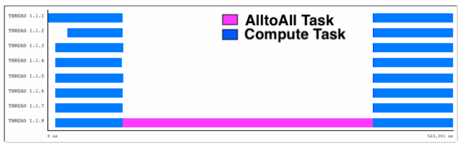


Figure 6.8: Speedup for 2D FFT and 3D FFT over a baseline implementation on 128 nodes.

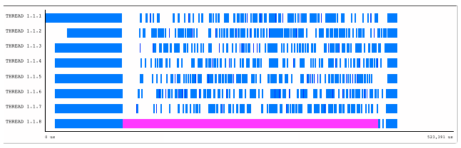
(EV-PO, CB-SW and CB-HW), we only present representative results for CB-SW. These equivalent performance results, as we will show in this section, are caused by the fact that collective calls only block a single worker thread per process. Hence, other worker threads are readily available for either polling MPI for events or executing callbacks in software. Thus, all event-based scenarios are able to promptly handle events and therefore provide equivalent performance. Further, since the performance for the scenario in which the communication thread shares cores with worker threads (CT-SH) never outperforms the scenario in which the communication thread is assigned a dedicated core (CT-DE), we do not show results for CT-SH.

Figure 6.8 (a) shows that CT-DE consistently performs slightly worse (∼4.0%) than the baseline approach. This is because, with CT-DE, the communication thread does not execute computation tasks once the collective communication finishes, and thus negatively impacts performance. In contrast, CB-SW consistently outperforms the baseline, and provides a maximum 26.8% performance gain for the matrix size 65536 × 65536. On an average, CB-SW results in 21.9% performance improvements.

To better understand the reason for the significant performance improvement due to CB-SW, Figure 6.9 presents the execution traces of 2D FFT for the baseline and event-based executions. Figure 6.9 (a) shows that all computation tasks need to wait for the `MPI_Alltoall` to finish before they can be executed. In contrast, Figure 6.9 (b) shows that event-based notification results in some computation tasks executing as soon as the necessary input data is received for them. As a result, we are able to overlap computation tasks which can be executed with the `MPI_Alltoall` that is in progress.



(a) Baseline with no communication-computation overlap.



(b) Event-based communication-computation overlap.

Figure 6.9: Parallel execution traces showing the effect of collective-computation overlapping 2D FFT. Same time range is shown for both figures. A single MPI process and its threads are shown for the sake of clarity.

Results for 3D FFT are displayed in Figure 6.8 (b). Since the 3D FFT benchmark invokes two `MPI_Alltoall` operations instead of one in 2D FFT, it exposes more opportunities for overlapping computation with the collective calls. Therefor the CB-SW approach achieves higher performance enhancements for 3D FFT. Overall, CB-SW provides 21.2% average improvement, with maximum improvement of 34.5% for the 4096^3 sized volume. In contrast, dedicating a core for the communication thread (CT-DE) degrades performance by 9.8% on average in comparison to the baseline approach.

In summary, the event-based exposure of collectives' progress effectively enables an overlap between computation and collective operations in both 2D FFT and 3D FFT. Consequently, we achieve substantial performance gains for both benchmarks.

6.3.2.2 MapReduce

Next, we evaluate the effect of overlapping computation with MPI collectives for the MapReduce framework. We experiments with two applications: Word Count (WC) and a dense Matrix Vector product (MV). Figure 6.10 shows the speedup results for both applications over a baseline implementation.

For the WC application, CB-SW provides 7.2% improvement on an average, and a maximum gain of 10.7% for a dataset consisting of 262 million words. In this application, reduce operations are extremely small as they only increase the counter associated with the key. Consequently, as the size of the dataset grows, the map tasks consume a higher proportion of the runtime. As a result, the impact of computation-communication overlap

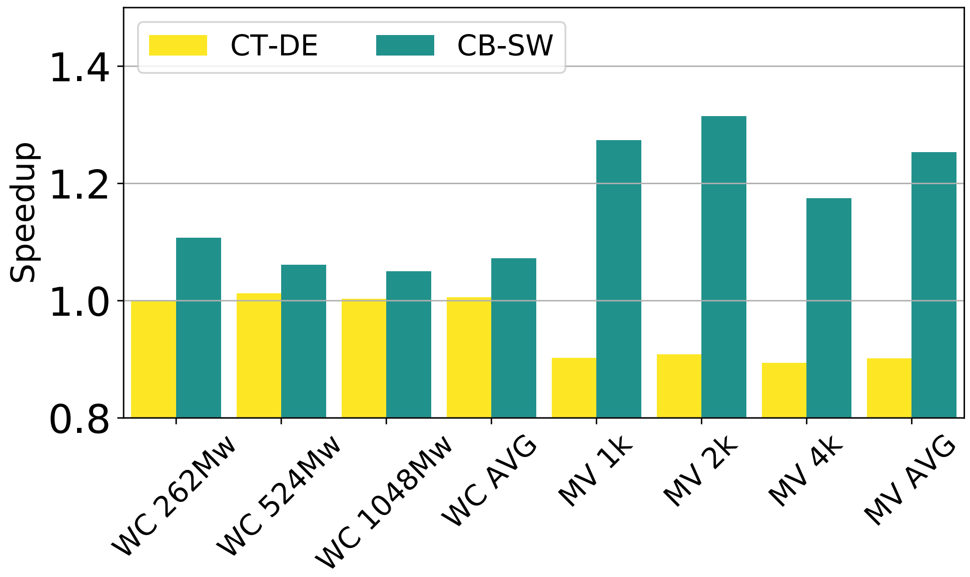


Figure 6.10: Speedup for the MapReduce-based WordCount (WC) and dense Matrix Vector product (MV) benchmarks with different problem sizes.

decreases, and the performance gains reduce to 4.9% for a dataset of 1048 million words.

Unlike the WC application, in the MV application, a similar amount of time is spent in the map and the reduce tasks. This translates to higher impact due to dedicating a thread to communication as well as enabling computation-communication overlap for the reduce tasks. For CT-DE, the inability to use the communication thread to execute tasks degrades performance up to 10.7%. In contrast, performance improvements, ranging from 17.4% to 31.4% are obtained as CB-SW enables overlap of reduce tasks with the `MPI_Alltoallv` collective executed for aggregating keys across processes.

6.3.3 Scalability of the collectives benchmarks

The results presented in this section are obtained using 128 nodes. We have performed weak-scaling experiments on 16, 32, and 64 nodes and have verified that the speedup trends among the different input-sets correlate in all scenarios with performance differences of at most 4.0% in the case of the FFT 3D application. This allows us to conclude that the collective overlapping benefits hold regardless the node count.

6.3.4 Comparison with Task-Aware MPI Library

The Task Aware MPI library (TAMPI) [73] provides MPI with a new level of threading support `MPI_TASK_MULTIPLE`. TAMPI works by intercepting blocking calls to MPI inside tasks and converting them to the non-blocking versions. The task execution is

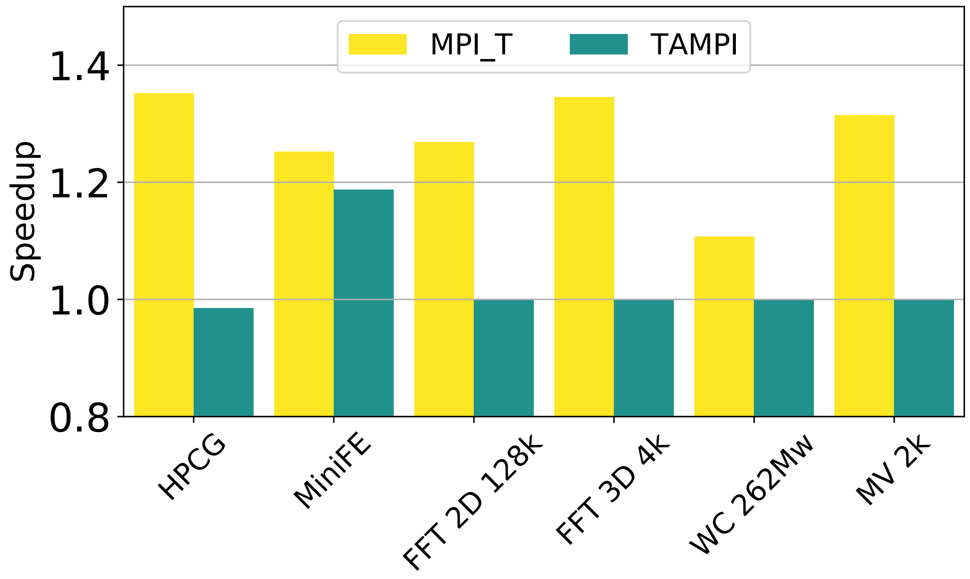


Figure 6.11: Performance comparison of our best performing proposal with TAMPI for every benchmark using 128 nodes

suspended and the MPI_Request object is added to a waiting list. This list is iterated by the workers in between task executions polling every request with the `MPI_Test` call and tasks whose requests have completed are rescheduled to keep executing. The key difference is that TAMPI polls every active request while our proposal only reacts to requests where the MPI layer notifies progression.

Figure 6.11 shows a performance comparison of the proposed approach with TAMPI. For point-to-point benchmarks with a high communication/computation ratio such as HPCG, the worker threads iterate the list of requests several times delaying the execution of useful computation by polling the status of requests with no changes. This effect can be seen on the HPCG benchmark where TAMPI behaves 1.5% below the baseline. However, for benchmarks with a lower communication/computation ratio such as MiniFE, TAMPI is able to perform in the same range of the MPI_T events based alternatives yielding a speedup of 18.7% compared to the 25.2% obtained by using MPI_T events. The relatively high computation time compared to communication allows a better overlap of the request polling with the computation task execution.

In the collective benchmarks, TAMPI performs exactly as the baseline as shown in Figure 6.11. TAMPI uses MPI calls to test for requests completion but it has no means of accessing internal information such as the partial completion of collectives. This makes TAMPI unable to do the collective overlapping proposed in this work.

6.4 Remarks

In this chapter, we explored mechanisms to optimize overlap between computation and communication in asynchronous task-based programs executing on distributed memory systems. By exposing information about MPI communication events through the MPI_T events interface to a task-based runtime, we significantly reduce idle time caused by waiting on specific MPI requests for point-to-point operations. We also proposed a novel scheme to overlap communication with computation on partially received data from collective operations. Our detailed evaluation on a production system provided performance improvements of up to 16.3% in benchmarks with point-to-point communication patterns, and of up to 34.5% in benchmarks with collective communication. Overall, our approach provides a transparent solution that requires no changes to the source code of an application programmed in OmpSs and MPI, yet improves its performance by better exploiting the overlap of computation and communication in such asynchronous task-based programming models.

Chapter 7

The Complete Runtime-Aware Architecture

7.1 Introduction

The previous three chapters of this thesis focus on three independent aspects: energy-efficiency, performance and communication, which are key in modern computing systems. Although they are self-contained proposals, it is possible to combine them in order to achieve a *Runtime Aware Architecture* where these schemes can interact together so that the three main goals are achievable simultaneously.

This chapter aims to describe how a Runtime Support Unit integrating *CATA*, *TDM*, and *MPI_T* events should be designed and how it should interact with the runtime system of an ATaP model. Instead of a bulk design with three components separated, it is possible to combine all of them in order to reuse information and optimize silicon area. TDM provides a TDG representation where the dependences induced by MPI_T events can be also integrated. Moreover, the TDG in TDM can provide the task criticality information needed to drive CATA.

The following sections describe the Runtime Support Unit design, the interface with the runtime system, the required hardware structure models, as well as the operational model. Finally, the conclusions on why such a design could be useful are provided. Unfortunately, the complexity of all the integrated components makes unfeasible the simulation of such system. Full system simulators such as gem5 do not support the Infiniband or the Intel OmniPath architecture required for modelling modern HPC systems. As a consequence, this chapter provides the theoretical foundation for future work to be performed.

7.2 The Runtime-Aware Architecture Hardware Extensions

This section introduces the complete Runtime Support Unit (RSU) and gives a glimpse of how all the three previous proposals interact together at the hardware level. The core of the RSU are the structures proposed in Chapter 5. The main responsibility of the RSU is to build the TDG created during the application execution by matching the task input and output dependence addresses. The TDG representation in hardware can then be used by other mechanisms to implement the CATA functionality by augmenting the TDG data structures to hold the task criticality. It is also possible to devise a hardware module that iterates through the graph to calculate the Bottom-Level (BL) of each task when no criticality annotations are provided.

Moreover, when creating tasks performing communication, the MPI_T events that act as dependences can be tracked as well by the RSU using the same hardware structures. In such scenario, a NIC capable of doing hardware tag-matching is augmented to deliver MPI_T events directly to the RSU through the Network-on-Chip. Thus the depending tasks can be unlocked automatically without the need of the ATaP model runtime system to explicitly track these events.

7.2.1 Interface and Integration with the Runtime System

The ATaP model runtime system relies on the RSU to deal with all the stages of the task-life cycle. The interface between both elements is done through ISA extensions. The ATaP model runtime system executes these instructions in different phases of the execution as explained in Chapters 4 and 5. For the sake of simplicity only the major changes to the instruction behavior are listed:

- *rsu_add_dependence(task_desc, dep_addr, size, direction, tag, src)*: After creating a task, the runtime system traverses its list of dependences and uses this instruction to inform the RSU of the dependences of the task. When dealing with communication dependences, direction is set to a value *COMM_REQ* to explicitly notify the RSU this is a communication dependence. The dep_addr is a *Communication Request* object generated by the runtime. In the case of communication dependences, the tag and the source that allows to identify MPI messages are added as well to match them with the incoming network interface request.

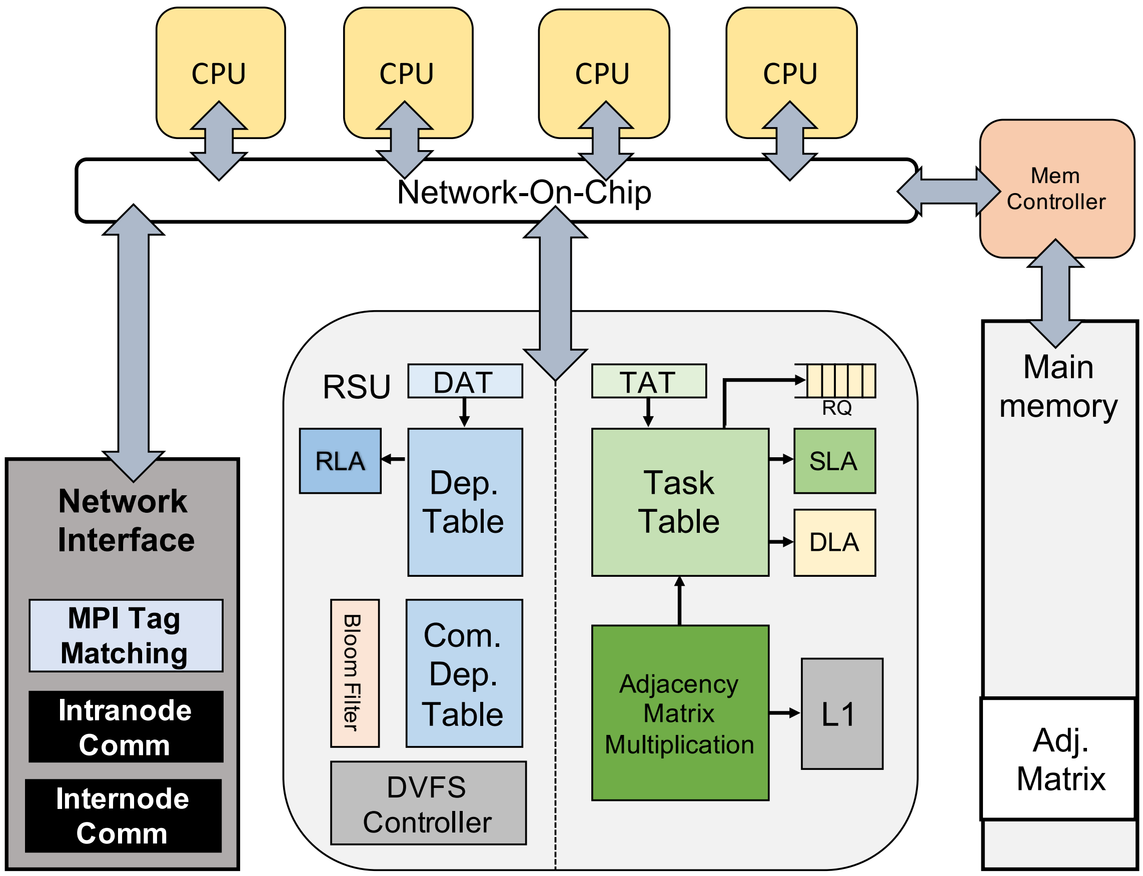


Figure 7.1: The complete RSU

- *rsu_get_ready_task()* → *task_desc, #criticality*: Just after notifying a task has finished, the runtime system uses this instruction to request the successors of the finished tasks and their criticality to the RSU. Then, the runtime system runs the appropiate scheduling algorithm with the retrieved tasks and their criticality. When dealing with communication dependences tasks can be unlocked at any other moment than task finalization. In order to retrieve the unlocked tasks and continue the application execution, the NIC can as well deliver the proposed MPI events to the CPU which will execute the *rsu_get_ready_task()* upon the event reception. An alternative and less complex approach is to poll the RSU executing this instruction during the worker threads idle time.

7.2.2 RSU Hardware Design

The RSU is a centralized hardware module connected to the NoC and accessed through a functional unit in the CPU pipeline issuing messages through a dedicated virtual channel.

The RSU is used to deal with the TDG, task criticality identification, and communication tasks unlocking through messages delivered from the NIC.

The core of the RSU are the DMU structures described in Chapter 5 and an overview can be seen in Figure 7.1. Tasks and dependences are identified through internal IDs that directly index the task and dependence SRAM tables and list arrays. The 64-bit pointers used in the runtime system are translated to the internal IDs by using two set-associative structures, the TAT and the DAT as described in Section 5.2.2.1. Task criticality estimation is done by using a hardware module to iterate the TDG. The BL algorithm obtains the criticality of a task through the exponentiation of the TDG adjacency matrix stored in main memory and accessed by sending requests to the memory controller through the NoC. A small L1 cache is added to speedup the adjacency matrix accesses. Finally, hardware to track which task is executed in each core is also added to identify the cores that should be accelerated using DVFS reconfigurations.

7.2.2.1 Task Criticality Identification

Dynamic task criticality identification is an expensive software operation [31]. It can be performed in hardware to reduce its overhead. To do this a hardware module to iterate the TDG needs to be added to the RSU. There are several alternatives for estimating the criticality of a task and in this section we explore how to provide efficient support for estimating the BL approach described in Chapter 2. The BL can be computed when a task is created using the adjacency matrix of the TDG. The adjacency matrix of a graph is described as a matrix whose elements are of the form $a_{i,j} : a \in \{0, 1\}$ with i, j being two vertices of the graph. If an edge exists between vertices v_i and v_j, then $a_{i,j} = 1$. The exponentiation of the adjacency matrix A^n results in a matrix where an entry $a_{i,j}$ is set to 1 if a path with length n exists from the vertex i to the vertex j or 0 otherwise [15]. The adjacency matrix elements are binary values and the matrix can be stored as a bitmap to save space, in main memory with a small L1 cache in the RSU to access it. The RSU also needs a matrix multiplication unit and temporary registers to save the matrix products. Since elements are binary, the multiplication can be implemented with simple logic gates. As an example, the TPU [64] provides a power efficient systolic-array implementation of a matrix multiplication unit for 8 bit units that can be significantly simplified to be used in the RSU as matrices areonly of 1 bit.

7.2.2.2 Communication Dependences

Tasks depending on an incoming message to be executed are associated to an event that is triggered by the NIC interface once the corresponding message reaches the node. When tasks performing MPI calls are created, the runtime system also creates and assigns an object representing the request with either the MPI tag and source of the message or the MPI_Request object depending on the arguments of the actual MPI call. Calls used to test or complete non-blocking operations rely on the MPI_Request object to identify the actual status inside the MPI runtime. The address of this created object is stored in the RSU dependence table as the task dependence. The dependence table needs to be extended with three 64-bit fields for the message tag, the source and a message timestamp to ensure matching messages preserving the order constraints imposed by MPI. A flag indicating the arrival of a message is also added if the dependence entry is pre-allocated before the task actually writes it.

To avoid iterating the whole dependence table to find incoming messages and save the space related to tag and source fields when not needed, communication dependences can be saved in a separated table. Messages can be matched by iterating the table and looking for the matching entries with the minimum timestamp. However in the case of large tables, this process can be optimized by using a a Bloom filter to prevent complete table walks when a message dependence is not present.

7.2.3 Operational Model

The previous sections give an overview of the design of the RSU and the hardware parts involved in its design. This section aims to describe the algorithms that provide the proposed functionalities. For the sake of simplicity, extended descriptions of the mechanisms presented in Chapters 4 and 5 are omitted.

7.2.3.1 Task Creation, Dependence Analysis, and Task Criticality Detection

The runtime system invokes the *rsu_create_task* instruction to send the task descriptor address to the RSU. Then for every dependence, it uses the *rsu_add_dependence* instruction to inform the RSU.

Dependences on regular memory addresses follow the functional model described in Section 5.2.3.1. However, for dependences on MPI messages the compiler and runtime create an object to encapsulate either the expected tag and source, or the MPI_Request

Algorithm 4: Algorithm for bottom level using the adjacency matrix.

```
Data: Adjacency Matrix $A$
$n = 2$
while $not\ is\_zero(A)$ do
    $A = A * A$
    for each task $i$ in the graph do
        for each leaf-task $j$ in the graph do
            if $A_{i,j}$ then
                $bl_i = max(bl_i, n)$
            end
        end
    end
    $n = n + 1$
end
```

object depending on the MPI call. This object is added as a regular dependence calling *rsu_add_dependence* with the request object as the dependence address and direction set to *COMM_REQ*. The dependence table holds a flag indicating if the message has already arrived and if it is set, the execution can proceed. Then, the task is not associated to that dependence and the dependence entry table is freed as the message already arrived.

Once a task is created, the criticality is estimated by using the BL algorithm. In order to avoid walking the complete TDG every single time a task is added to the graph, a rough estimation of the BL based on the parent BL can be added at first $Task_{BL} = max(0, Parent_{BL} - 1)$, and then refined by executing the BL algorithm at fixed intervals.

Instead of recursively walking through the TDG as in the original algorithm, an efficient way to find the maximum distance of any task to a leaf-task as described is to use the adjacency matrix exponentiation. The adjacency matrix successive exponents are calculated until the result has the complete rows set to zero. The hardware required is extremely small as only 1-bit fused multiply add operations are required and it can be implemented with simple AND and OR gates with the matrix stored in bitmaps. The RSU retrieves the matrix from an area in main memory by issuing coalesced reads to a memory controller through the NoC or using a cache memory, and exponentiates it using the proposed hardware unit. The temporal A^n exponents can be saved in the associated L1 cache or spilled to memory if the matrix size grows too large. This memory table containing the adjacency matrix is completely managed by the RSU and updated everytime a task gets its successors updated. Since the matrix is stored as a bitmap in memory the amount of space it requires for representing large TDGs is greatly reduced. A TDG with

512 nodes requires 32KB of memory storage, fitting in a 64KB L1-cache when storing the A^n temporal exponents.

7.2.3.2 Ready Task Retrieval and Scheduling

The runtime system accesses to the tasks ready to be scheduled as detailed in Section 5.2.3.3 using the *rsu_get_ready_task* instruction. The runtime system obtains the tasks ready to be scheduled and their criticality once a task completes. For tasks depending on MPI events, the NIC also needs to deliver the events to the runtime system to retrieve the tasks without the need for polling.

7.2.3.3 Task Execution

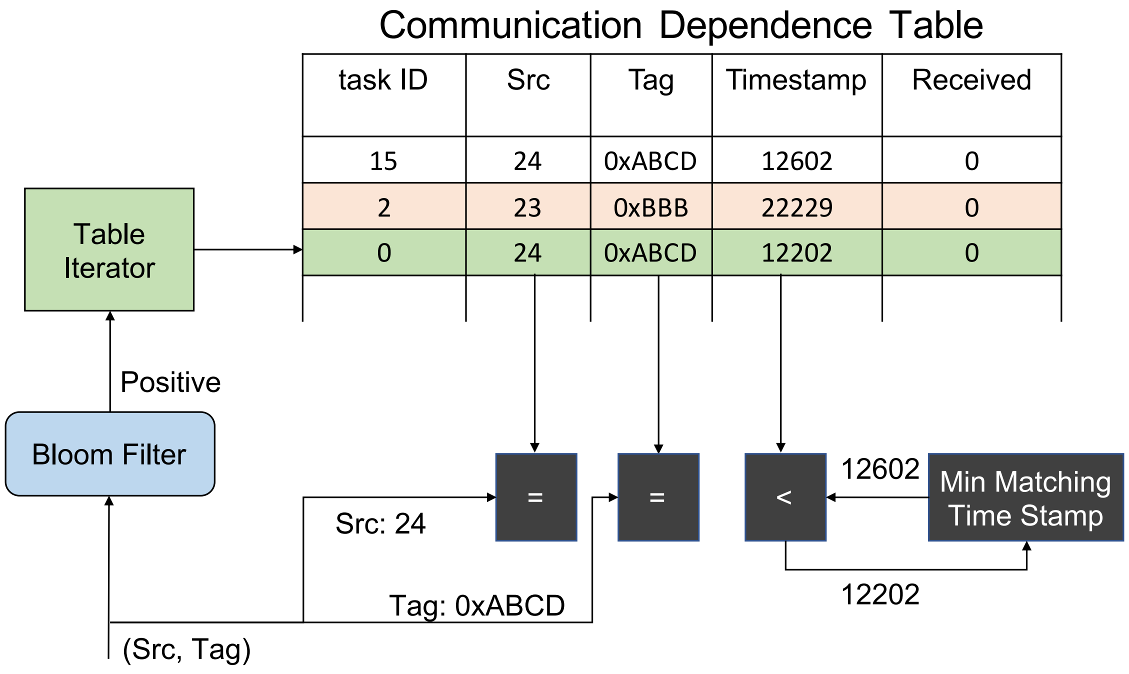
The *rsu_start_task(task_desc)* is used to notify the start of a task execution to the RSU. The task retrieves the criticality level from the task table and reconfigures the DVFS module accordingly. As the BL recomputes criticality only on fixed intervals of time, the DVFS module can be reconfigured with an outdated criticality estimation. However, this situation can be reverted later as a more accurate value is estimated and additional reconfigurations can be triggered when the criticality calculation is over. Thus, we believe this would have a limited effect on performance and DVFS reconfigurations.

7.2.3.4 Task Finalization

To notify task completion to the RSU, the *rsu_finish_task* instruction is used and all the completing task successors are released as described in Section 5.2.3.2. The tasks are stored in the Ready Queue and then consumed by the runtime system that executes the associated scheduling algorithm. This instruction also triggers the DVFS reconfiguration that happens when a task completes as seen in Section 4.2.2.2

7.2.3.5 Task Unlocking on MPI Message Arrival

Upon a message arrival or sending completion, the network interface sends the source and tag arguments of the message, or the associated MPI_Request object if the tag matching engine successfully identified the message. The RSU then matches the tag and source against a Bloom filter to check whether there is a dependence waiting for that message or the message has arrived before the associated receiving task is created. If the message is being expected, the tasks associated to the dependence are retrieved from the dependence readers list and we use the same algorithms described in Chapter 5 to release them. Figure 7.2 shows how the dependence table is extended to hold the expected MPI message tag, source, and the timestamp when the dependence is created in order to preserve MPI ordering semantics. The received tag and source are successfully checked against a Bloom filter and then the complete table is iterated until it finds the minimum matching timestamp request with the same source and tag values. The resulting task is then used to address the task table and unlock its successors.


Communication Dependence Table

task ID	Src	Tag	Timestamp	Received
15	24	0xABCD	12602	0
2	23	0xBBB	22229	0
0	24	0xABCD	12202	0

Figure 7.2: Tag and source matching from an incoming message

On the other hand, if an unexpected message arrives and the receiving task has not been created in this node yet, a new dependence entry is allocated on the table and a flag to notify that the message arrived is set. This flag is used on task creation to let the RSU know that it does not need to add the message as a dependence anymore, because it is ready to be received.

In the case of the tag-matching engine providing back an MPI_Request object instead of a source and tag pair, the MPI_Request object address was set as the actual dependence, so it only accesses the DAT to get the internal address and release the associated task.

One limitation of this approach is that only inter-node messages are detected. In order to also support intra-node MPI communications, this must be done through the network interface or the low level communication layer that needs to trigger the RSU event delivery. Otherwise if two processes are running on the same node, a task doing an MPI_Recv on a message sent by the other process in the same node will never be unlocked as the message is passed through shared memory and never reaches the NIC.

7.3 Remarks

This chapter describes a theoretical Runtime Aware Architecture based on the three previous proposals of this thesis. The conjunction of the three techniques allows designing a Runtime Support Unit that is able to deal with energy, performance and ease some of the communication layer bottlenecks. Furthermore, this RSU provides a seamless interoperability of the ATaP model runtime system and the communication layer.

Current simulation infrastructures prevent us from evaluating the three proposals together as there is no execution driven simulators with adequate support for high performance interconnection networks and the development of such a tool is out of the scope of this dissertation. Instead, this chapter introduces a novel way to perform task criticality estimation using hardware and describes a complete interface for the software runtime system and the RSU.

Chapter 8

Conclusions

This chapter summarizes the main conclusions and contributions of this thesis and presents the future research lines opened by this work. Then it shows the list of publications produced during the realization of this thesis and acknowledges the financial support.

8.1 Goals, Contributions and Main Conclusions

The increasing complexity of multi-core processors has motivated the research on new programming models to efficiently manage the extensive degree of parallelism that these systems offer. Among these programming models, ATaP solutions such as OmpSs, Legion, OpenMP 4.0, Charm++, Habanero or HPX are gaining traction among the research community and started to drive architectural decissions for future Exascale and beyond systems.

This thesis delves into the hardware-software interface to efficiently support ATaP models and lies the foundation for a real runtime-aware architecture while previous proposals just present complete hardware runtime system implementations [39, 116, 118] or provide support for extremely specific bottlenecks of the system [71]. The proposals in this thesis exploit both directions in the architecture-runtime system communication so that the architecture can exploit algorithmic-level information to reconfigure itself, it can provide hardware support to accelerate runtime specific operations, and the runtime system can get access to non-trivial hardware status to do a better scheduling of tasks performing inter-node communication.

The first contribution of this thesis focuses on efficiently driving power management operations of a multi-core system exploiting the information hidden in the Task Dependence Graph (TDG) of an application. Given the criticality of a task, defined as the task being in the critical path of the TDG or not, the runtime system is able to change the frequency of individual cores to either accelerate critical tasks, or decelerate non-critical ones to exploit the slack and manage power more efficiently. This contribution studies a software only approach and points out its limitations as the core count in the chip scales. The main proposal here is to overcome this limitations by using the Runtime Support Unit (RSU), a hardware module that holds the most relevant information for the runtime system and can drive the DVFS reconfigurations accordingly.

The second contribution targets fine-grain tasks and their effects on software-only runtime systems. Fine-grain tasks are necessary to do an efficient load-balancing between all the cores in the chip. As the core count greatly increases each generation, it is necessary to reduce the size of independent work pieces in order to keep all the processing elements busy when applying *weak scaling*. We provide an analysis of all the runtime system execution phases and detect that the bottlenecks are in the task creation and completion operations. We propose Task Dependence Manager (TDM) as a hardware targeted to accelerate dependence management operations that allows keeping non-bottleneck operations such as the scheduler in software to retain flexibility. TDM holds a complete representation of the Task Graph in hardware and can reduce task creation time up to 5.2x in the measured benchmarks.

The third contribution switches to distributed memory systems and analyzes how the information present in the communication layer can be exploited by the runtime system to prevent unnecessary blocking of threads waiting for messages yet to arrive. Mechanisms based in exposing the MPI runtime system activities or the NIC status to the ATaP model runtime system are proposed and evaluated. Moreover, this proposal allows performing previously unfeasible computation/communication overlap in collective operations using partially received data. This approach requires no changes to the code of an application written in OmpSs and MPI and greatly improves its performance by doing a more efficient computation/communication overlap.

These three contributions can be combined together in a complete Runtime System Unit that keeps track of the TDG accelerating task creation, analyzing it to find task-criticality to drive DVFS reconfigurations, and receiving events from the network interface to unlock tasks depending on communication events when necessary. This complete RSU provides support for both shared and distributed memory computing and aids the runtime system with non-mutable operations while keeping flexibility to rely on software approaches when needed.

Besides from the major contributions described above, this thesis has produced several tools and methodologies used by other researchers at the Barcelona Supercomputing Center and the University of Cantabria.

This thesis builds a simulation infrastructure based on gem5 with support for OmpSs workloads that exposes relevant runtime system events to the simulator layer. Significant simulator bugs and issues were fixed in order to provide a stable tool for other researchers.

Based on the simulator-runtime system interface developed in gem5, a module for producing *paraver* traces with micro-architectural and algorithmic states was developed. As the tracing happens only on the simulator layer, the application timing is virtually non affected and there is no memory pollution due to the tracing tools.

The large design space exploration for all the experiments on this thesis, consisting of thousands of combinations motivated the creation of a tool to manage workloads. Tizona [26], an HPC worload manager, which has been released as open source and allows to specify a parameter space to create multiple experiments at once in a single file. Moreover, the same file describes how statistics can be obtained and is able to summarize the results avoiding the need of creating specific scripts for this task. Tizona is also able to abstract the underlying machine queue system, so that no specific SLURM or GridEngine scripts are used to run workloads and the same file can be used across different systems with minimal or no changes.

8.2 Future Work

The proposals presented in this thesis open the door to new research topics that could be explored in the future. Among others, three main research lines can be of great interest.

- *Hardware mechanisms for task criticality detection and execution time estimation.* In this thesis we rely on static annotations by the programmer or the Bottom-Level (BL) analysis of the TDG done in the software layer. However, in certain scenarios it is not possible to obtain these annotations, or the small granularity of tasks prevents applying BL due to its expensive cost. Mechanisms to iterate the TDG or estimate task criticality using approximations of the execution time of a task should be developed to allow fast and on-the-fly criticality estimations. Furthermore, being able to estimate execution time at different clock speeds could open opportunities for more advanced scheduling mechanisms trying to equalize the execution time of different TDG branches.

- *Region-based and non-contiguous memory dependences analysis hardware support.* TDM focuses on a plain dependence analysis model where only the base pointer of the dependences is compared. Minimal extensions can be considered to track regions of contiguous memory as only the length of a dependence is only needed. However in the case of multi-dimensional slices of data structures where the memory is not contiguous, new mechanisms are needed in order to perform the dependence analysis as the algorithms complexity in this scenario is greatly increased.

- *Network interface hardware events delivery mechanisms.* The last contribution of this thesis proposes to deliver events directly from the NIC to the ATaP model runtime system. This scenario makes possible to simplify several aspects of the interoperability between the ATaP model and MPI. Although doorbell capabilities and network processors were part of commercial interconnection technologies, state-of-the-art ones such as OmniPath do not offer those options. Research on NIC-processor integration is needed to develop efficient hardware based event delivery systems.

8.3 Publications

The list of publications derived from this thesis is presented.

- **E. Castillo**, N. Jain, M. Casas, M. Moretó, M. Schulz, R. Beivide, M. Valero, A. Bhatele
  Optimizing Computation-Communication Overlap in Asynchronous Task-Based Programs.
  Accepted as poster at ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, 2019, Washington D.C., USA. 2019.

- **E. Castillo**, Ll. Álvarez, M. Moretó, M. Casas, E. Vallejo, J. L. Bosque, R. Beivide, and M. Valero.
  Architectural support for task dependence management with flexible software scheduling.
  IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018, pp. 283-295, 2018.

- **E. Castillo**, M. Moretó, M. Casas, Ll. Álvarez, E. Vallejo, K. Chronaki, R. M. Badia, J. L. Bosque,R. Beivide, E. Ayguadé, J. Labarta, and M. Valero.
  CATA: criticality aware task acceleration for multicore processors.
  2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016, pp. 413-422, 2016.

- Ll. Álvarez, M. Moretó, M. Casas, **E. Castillo**, X. Martorell, J. Labarta, E. Ayguadé, and M. Valero.
  Runtime-guided management of scratchpad memories in multicore architectures.
  2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA,October 18-21, 2015, pp. 379-391, 2015.

- M. Casas, M. Moretó, Ll. Álvarez, **E. Castillo**, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar,O. S. Unsal, A. Cristal, E. Ayguadé, J. Labarta, and M. Valero.
  Runtime-aware architectures.
  Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings, pp. 16-27, 2015.

Other relevant publications not directly related to the PhD include:

- E. Stafford, J. L. Bosque, C. Martínez, F. Vallejo, R. Beivide, C. Camarero, and **E. Castillo**.
  Assessing the suitability of king topologies for interconnection networks.
  IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 3, pp. 682-694, 2016.

- **E. Castillo**, C. Camarero, A. Borrego, and J. L. Bosque.
  Financial applications on multi-CPU and multi-GPU architectures.
  The Journal of Supercomputing, vol. 71, no. 2, pp. 729-739, 2015.

- **E. Castillo**, C. Camarero, E. Stafford, F. Vallejo, J. L. Bosque, and R. Beivide.
  Advanced switching mechanisms for forthcoming on-chip networks.
  2013 Euromicro Conference on Digital System Design, DSD 2013, Los Alamitos, CA, USA, September 4-6, 2013, pp. 598-605, 2013.

- E. Stafford, **E. Castillo**, F. Vallejo, J. L. Bosque, C. Martínez, C. Camarero, and R. Beivide.

King topologies for fault tolerance.
14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, pp. 608-616.

- **E. Castillo**, J. Castillo, J. Cano, P. Huerta, and J. I. Martínez.
A key size configurable high speed RSA coprocessor.
IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2011, Salt Lake City, Utah, USA, 1-3 May 2011, p. 250, 2011.

- C. Pedraza, **E. Castillo**, J. Castillo, J. L. Bosque, J. I. Martínez, O. D. Robles, J. Cano, and P. Huerta.
Content-based image retrieval algorithm acceleration in a low-cost reconfigurable FPGA cluster.
Journal of Systems Architecture - Embedded Systems Design, vol. 56, no. 11, pp. 633-640, 2010.

- J. Castillo, J. L. Bosque, **E. Castillo**, P. Huerta, and J. I. Martínez.
Hardware accelerated montecarlofinancial simulation over low cost FPGA cluster.
23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009, pp. 1-8, 2009.

- **E. Castillo**, C. Pedraza, J. Castillo, C. Camarero, J. L. Bosque, R. M. de Llano, and J. I. Martínez.
SMILE: scientific parallel multiprocessing based on low-cost reconfigurable hardware.
16th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2008, 14-15April 2008, Stanford, Palo Alto, California, USA, pp. 277-278, 2008.

- C. Pedraza, **E. Castillo**, J. Castillo, C. Camarero, J. L. Bosque, J. I. Martínez, and R. M. de Llano.
Cluster architecture based on low cost reconfigurable hardware.
FPL 2008, International Conference on Field Programmable Logic and Applications, Heidelberg, Germany, 8-10 September2008, pp. 595-598, 2008.

Bibliography

[1] (2018). TOP500 Supercomputer Site. http://www.top500.org.

[2] Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., and Kale, L. (2014). Parallel Programming with Migratable Objects: Charm++ in Practice. SC.

[3] Alvarez, L., Casas, M., Labarta, J., Ayguade, E., Valero, M., and Moreto, M. (2018). Runtime-guided management of stacked dram memories in task parallel programs. In *International Conference on Supercomputing*, (ICS), pages 379–391.

[4] Alvarez, L., Moretó, M., Casas, M., Castillo, E., Martorell, X., Labarta, J., Ayguadé, E., and Valero, M. (2015). Runtime-guided management of scratchpad memories in multicore architectures. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, pages 379–391.

[5] AMD (2011). The new AMD Opteron processor core technology. Technical report.

[6] Arvind, K. and Nikhil, R. S. (1990). Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318.

[7] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *International Conference on Parallel Processing (Euro-Par)*, pages 863–874.

[8] Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., and Labarta, J. (2004). Nanos mercurium: a research compiler for OpenMP. In *European Workshop on OpenMP (EWOMP)*, pages 103–109.

[9] Barcelona Supercomputing Center. Extrae. https://tools.bsc.es/paraver. Online; accessed 2018-09-27.

[10] Barcelona Supercomputing Center. Paraver. https://tools.bsc.es/paraver. Online; accessed 2018-09-27.

[11] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *Proceedings of the 2012 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA. IEEE Computer Society.

[12] Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J. (2006). CellSs: A programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 86:1–86:11, New York, NY, USA. ACM.

[13] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008a). The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81.

[14] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008b). The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81.

[15] Biggs, N., Biggs, N., and Norman, B. (1993). *Algebraic Graph Theory*. Cambridge Mathematical Library. Cambridge University Press.

[16] Binkert, N., Beckmann, B., Black, G., Reinhardt, S., Saidi, A., Basu, A., Hestness, J., Hower, D., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M., and Wood, D. (2011). The gem5 simulator. *Computer Architecture News*, 39(2):1–7.

[17] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. In *International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216.

[18] Boneti, C., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Cher, C.-Y., and Valero, M. (2008). Software-controlled priority characterization of POWER5 processor. In *ISCA*, pages 415–426.

[19] Boost (2018). Boost C++ Libraries.

[20] Brumar, I., Casas, M., Moretó, M., Valero, M., and Sohi, G. S. (2017). ATM: approximate task memoization in the runtime system. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 1140–1150.

[21] Bueno, J., Martorell, X., Badia, R. M., Ayguadé, E., and Labarta, J. (2013). Implementing OmpSs support for regions of data in architectures with multiple address spaces. In *Proceedings of the 27th International Conference on Supercomputing (ICS)*, pages 359–368.

[22] Büttner, D., Acquaviva, J., and Weidendorfer, J. (2013). Real asynchronous mpi communication in hybrid codes through openmp communication tasks. In *International Conference on Parallel and Distributed Systems*, pages 208–215.

[23] Caheny, P., Alvarez, L., Derradji, S., Valero, M., Moretó, M., and Casas, M. (2018a). Reducing cache coherence traffic with a numa-aware runtime approach. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):1174–1187.

[24] Caheny, P., Alvarez, L., Valero, M., Moretó, M., and Casas, M. (2018b). Runtime-assisted cache coherence deactivation in task parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 35:1–35:12.

[25] Caheny, P., Casas, M., Moretó, M., Gloaguen, H., Saintes, M., Ayguadé, E., Labarta, J., and Valero, M. (2016). Reducing cache coherence traffic with hierarchical directory cache and numa-aware runtime scheduling. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, pages 275–286.

[26] Castillo, E. (2017). Tizona: Hpc workloads management tool.

[27] Chasapis, D., Casas, M., Moretó, M., Schulz, M., Ayguadé, E., Labarta, J., and Valero, M. (2016). Runtime-guided mitigation of manufacturing variability in power-constrained multi-socket NUMA nodes. In *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016*, pages 5:1–5:12.

[28] Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., and Valero, M. (2015). PARSECSs: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Transactions on Architecture and Code Optimization*, 12(4):41:1–41:22.

[29] Chatterjee, S., Tasirlar, S., Budimlic, Z., Cavé, V., Chabbi, M., Grossman, M., Sarkar, V., and Yan, Y. (2013). Integrating asynchronous task parallelism with mpi. In *27th IEEE International Symposium on Parallel and Distributed Processing*, pages 712–725.

[30] Chronaki, K., Casas, M., Moretó, M., Bosch, J., and Badia, R. M. (2018). Taskgenx: A hardware-software proposal for accelerating task parallelism. In *High Performance Computing - 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings*, pages 389–409.

[31] Chronaki, K., Rico, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. (2015a). Criticality-aware dynamic task scheduling for heterogeneous architectures. In *International Conference on Supercomputing (ICS)*, pages 329–338.

[32] Chronaki, K., Rico, A., Badia, R. M., Ayguade, E., Labarta, J., and Valero, M. (2015b). Criticality-aware dynamic task scheduling on heterogeneous architectures. In *ICS*.

[33] Cook, H., Moretó, M., Bird, S., Dao, K., Patterson, D. A., and Asanovic, K. (2013). A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ISCA*, pages 308–319.

[34] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10.

[35] Dimic, V., Moretó, M., Casas, M., and Valero, M. (2017). Runtime-assisted shared cache insertion policies based on re-reference intervals. In *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, pages 247–259.

[36] Donald, J. and Martonosi, M. (2006). Techniques for multicore thermal management: Classification and new exploration. In *ISCA*, pages 78–88.

[37] Dongarra, J. J., Heroux, M. A., and Luszczek, P. (2015). Hpcg benchmark: a new metric for ranking high performance computing systems.

[38] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193.

[39] Etsion, Y., Cabarcas, F., Rico, A., Ramirez, A., Badia, R. M., Ayguade, E., Labarta, J., and Valero, M. (2010a). Task superscalar: An out-of-order task pipeline. In *International Symposium on Microarchitecture (MICRO)*, pages 89–100.

[40] Etsion, Y., Cabarcas, F., Rico, A., Ramírez, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. (2010b). Task superscalar: An out-of-order task pipeline. In *MICRO*, pages 89–100.

[41] Fang, Z., Zhang, L., Carter, J. B., Ibrahim, A., and Parker, M. A. (2007). Active memory operations. In *International Conference on Supercomputing (ICS)*, pages 232–241.

[42] Forum, M. P. I. (2012). MPI: A Message-Passing Interface Standard Version 3.0.

[43] Frank, S. (1987). Tightly coupled multiprocessor systems speed memory access time. *Electronics*, 57(1).

[44] Grass, T., Allande, C., Armejach, A., Rico, A., Ayguadé, E., Labarta, J., Valero, M., Casas, M., and Moretó, M. (2016). MUSA: a multi-level simulation approach for next-generation HPC machines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 526–537.

[45] Gropp, W. (2002). Mpich2: A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK. Springer-Verlag.

[46] Hemanns, M. A., Hjlem, N. T., Knobloch, M., Mohror, K., and Schulz, M. (2018). Enabling callback-driven runtime introspection via mpi_t. In *Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI)*.

[47] Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

[48] Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38.

[49] Hoefler, T. and Gottlieb, S. (2010). Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 132–141. Springer.

[50] Hoefler, T. and Lumsdaine, A. (2008). Message progression in parallel computing - to thread or not to thread? In *International Conference on Cluster Computing*, pages 213–222.

[51] Hoefler, T., Lumsdaine, A., and Rehm, W. (2007). Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society/ACM.

[52] Hoffmann, H., Eastep, J., Santambrogio, M. D., Miller, J. E., and Agarwal, A. (2010). Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, pages 79–88.

[53] Hum, H. H. J., Maquelin, O., Theobald, K. B., Tian, X., Tang, X., Gao, G. R., Cupryk, P., Elmasri, N., Hendren, L. J., Jimenez, A., Krishnan, S., Marquez, A., Merali, S., Nemawarkar, S. S., Panangaden, P., Xue, X., and Zhu, Y. (1995). A design study of the EARTH multiprocessor. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 59–68.

[54] Intel Corporation (2008). Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors.

[55] Intel Corporation (2015). Intel performance scaled messaging 2 (psm2) programmers guide.

[56] Ipek, E., Kirman, M., Kirman, N., and Martinez, J. F. (2007). Core fusion: Accommodating software diversity in chip multiprocessors. In *ISCA*, pages 186–197.

[57] Jain, N., Bhatele, A., Yeom, J.-S., Adams, M. F., Miniati, F., Mei, C., and Kale, L. V. (2015). Charm++ & MPI: Combining the best of both worlds. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IPDPS '15. IEEE Computer Society. LLNL-CONF-663041.

[58] Jaulmes, L., Casas, M., Moretó, M., Ayguadé, E., Labarta, J., and Valero, M. (2015). Exploiting asynchrony from exact forward recovery for DUE in iterative solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 53:1–53:12.

[59] Jeffrey, M., Subramanian, S., Abeydeera, M., Emer, J., and Sanchez, D. (2016). Data-centric execution of speculative parallel programs. In *International Symposium on Microarchitecture (MICRO)*, pages 1–13.

[60] Jeffrey, M. C., Subramanian, S., Yan, C., Emer, J. S., and Sanchez, D. (2015). A scalable architecture for ordered parallelism. In *International Symposium on Microarchitecture (MICRO)*, pages 228–241.

[61] Jeffrey, M. C., Ying, V. A., Subramanian, S., Lee, H. R., Emer, J., and Sanchez, D. (2018). Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism. In *Proceedings of the 51st annual IEEE/ACM international symposium on Microarchitecture (MICRO-51)*.

[62] Jiménez, V., Gioiosa, R., Cazorla, F. J., Buyuktosunoglu, A., Bose, P., and O'Connell, F. P. (2012). Making data prefetch smarter: Adaptive prefetching on POWER7. In *PACT*, pages 137–146.

[63] Joao, J. A., Suleman, M. A., Mutlu, O., and Patt, Y. N. (2013). Utility-based acceleration of multithreaded applications on asymmetric CMPs. In *ISCA*, pages 154–165.

[64] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami,

R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA. ACM.

[65] Kägi, A., Burger, D., and Goodman, J. R. (1997). Efficient synchronization: Let them eat QOLB. In *International Symposium on Computer Architecture (ISCA)*, pages 170–180.

[66] Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., and Fey, D. (2014). Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA. ACM.

[67] Kamal, H. and Wagner, A. (2014). An integrated fine-grain runtime system for mpi. *Computing*, 96(4):293–309.

[68] Kaxiras, S. and Martonosi, M. (2008). Computer architecture techniques for power-efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207.

[69] Keppitiyagama, C. and Wagner, A. S. (2001). Asynchronous mpi messaging on myrinet. In *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 50:1–50:8.

[70] Khubaib, Suleman, M. A., Hashemi, M., Wilkerson, C., and Patt, Y. N. (2012). Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *MICRO*, pages 305–316.

[71] Kumar, S., Hughes, C. J., and Nguyen, A. (2007a). Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 162–173.

[72] Kumar, S., Hughes, C. J., and Nguyen, A. D. (2007b). Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA*, pages 162–173.

[73] Labarta, J., Beltran, V., PeÃśa, A. J., Perez, J. M., Teruel, X., Bellon, J., Holmes, D., Farre, P., and Sala, K. (2018). Improving the interoperability between mpi and task-based programming models. In *Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI)*.

[74] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. (2009). McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, pages 469–480.

[75] Liu, H., Chen, Y., Liao, X., Jin, H., He, B., Zheng, L., and Guo, R. (2017). Hardware/software cooperative caching for hybrid dram/nvm memory architectures. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 26:1–26:10.

[76] Lo, D. and Kozyrakis, C. (2014). Dynamic management of TurboMode in modern multi-core chips. In *HPCA*, pages 603–613.

[77] Lu, H., Seo, S., and Balaji, P. (2015). Mpi+ult: Overlapping communication and computation with user-level threads. *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 444–454.

[78] Manivannan, M., Negi, A., and Stenström, P. (2013). Efficient forwarding of producer-consumer data in task-based programs. In *Proceedings of the 2013 42nd International Conference on Parallel Processing*, ICPP '13, pages 517–522, Washington, DC, USA. IEEE Computer Society.

[79] Manivannan, M., Papaefstathiou, V., Pericas, M., and Stenstrom, P. (2016). Radar: Runtime-assisted dead region management for last-level caches. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 644–656.

[80] Manivannan, M. and Stenstrom, P. (2014). Runtime-guided cache coherence optimizations in multi-core architectures. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, International Parallel and Distributed Processing Symposium, pages 625–636, Washington, DC, USA. IEEE Computer Society.

[81] Marathe, A., Zhang, Y., Blanks, G., Kumbhare, N., Abdulla, G., and Rountree, B. (2017). An empirical survey of performance and energy efficiency variation on intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, E2SC'17, pages 9:1–9:8, New York, NY, USA. ACM.

[82] Marjanović, V., Labarta, J., Ayguadé, E., and Valero, M. (2010). Overlapping communication and computation by using a hybrid mpi/smpss approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA. ACM.

[83] Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P. (1991). First-class user-level threads. *SIGOPS Oper. Syst. Rev.*, 25(5):110–121.

[84] Miller, T., Thomas, R., and Teodorescu, R. (2012). Mitigating the effects of process variation in ultra-low voltage chip multiprocessors using dual supply voltages and half-speed units. *Computer Architecture Letters*, 11(2):45–48.

[85] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[86] Mueller, F. (1993). A library implementation of posix threads under unix. In *In Proceedings of the USENIX Conference*, pages 29–41.

[87] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. (2009). Cacti 6.0: A tool to model large caches. *HP Laboratories*.

[88] Nikhil, R. S. (1993). The programming language id and its compilation for parallel machines. *International Journal of High Speed Computing*, (2):171–223.

[89] Nikhil, R. S., Papadopoulos, G. M., and Arvind (1992). *T: A Multithreaded Massively Parallel Architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 156–167.

[90] Nvidia Corporation (2017). CUDA C Programming Guide. Version 10.0. October 2018. https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf. Online; accessed 2018-12-05.

[91] OpenMP3 (2018). OpenMP Application Program Interface. Version 3.0. May 2018.

[92] OpenMP4 (2013). OpenMP Application Program Interface. Version 4.0. July 2013.

[93] Ortega, C., Moretó, M., Casas, M., Bertran, R., Buyuktosunoglu, A., Eichenberger, A. E., and Bose, P. (2017). libprism: an intelligent adaptation of prefetch and SMT levels. In *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*, pages 28:1–28:10.

[94] Pallipadi, V. and Starikovskiy, A. (2006). The ondemand governor: past, present and future. In *Proceedings of Linux Symposium, vol. 2, pp. 223-238.*

[95] Pan, A. and Pai, V. S. (2015). Runtime-driven shared last-level cache management for task-parallel programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 11:1–11:12.

[96] Papadopoulos, G. M. and Culler, D. E. (1990). Monsoon: An explicit token-store architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 82–91.

[97] Papaefstathiou, V., Katevenis, M. G., Nikolopoulos, D. S., and Pnevmatikatos, D. (2013). Prefetching and cache management using task lifetimes. In *Proceedings of the 27th International Conference on Supercomputing (ICS)*, pages 325–334.

[98] Pérez, J. M., Beltran, V., Labarta, J., and Ayguadé, E. (2017). Improving the integration of task nesting and dependencies in openmp. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 809–818.

[99] Planas, J., Badia, R. M., Ayguade, E., and Labarta, J. (2013). Self-adaptive OmpSs tasks in heterogeneous environments. In *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 138–149.

[100] Porikli, F. (2005). Integral histogram: A fast way to extract histograms in Cartesian spaces. In *Conference on Computer Vision and Pattern Recognition Workshops (CVPR)*, pages 829–836.

[101] Quinlan, S. and Dorward, S. (2002). Venti: A new approach to archival storage. In *Proceedings of the FAST '02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA*, pages 89–101.

[102] Qureshi, M. K. and Patt, Y. N. (2006). Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432.

[103] Reinders, J. (2007). *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly Media.

[104] Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramirez, A., and Valero, M. (2011). Trace-driven simulation of multithreaded applications. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 87–96.

[105] Rogers, P. (2013). Heterogeneous system architecture overview. In *2013 IEEE Hot Chips 25 Symposium (HCS)*, pages 1–41.

[106] Rountree, B., Ahn, D., de Supinski, B., Lowenthal, D., and Schulz, M. (2012). Beyond DVFS: A first look at performance under a hardware-enforced power bound. pages 947–953.

[107] Sanchez, D., Yoo, R. M., and Kozyrakis, C. (2010a). Flexible architectural support for fine-grain scheduling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 311–322.

[108] Sanchez, D., Yoo, R. M., and Kozyrakis, C. (2010b). Flexible Architectural Support for Fine-Grain Scheduling. In *ASPLOS*, pages 311–322.

[109] Sánchez Barrera, I., Moretó, M., Ayguadé, E., Labarta, J., Valero, M., and Casas, M. (2018). Reducing data movement on large shared memory systems by exploiting computation dependencies. In *Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018*, pages 207–217.

[110] Sandberg, A., Nikoleris, N., Carlson, T. E., Hagersten, E., Kaxiras, S., and Black-Schaffer, D. (2015). Full speed ahead: Detailed architectural simulation at near-native speed. In *2015 IEEE International Symposium on Workload Characterization*, pages 183–192.

[111] Schulz, R. (2008). 3d fft with 2d decomposition. `http://cmb.ornl.gov/members/z8g/csproject-report.pdf`. Online; accessed 2018-09-27.

[112] Shirako, J., Peixotto, D. M., Sarkar, V., and Scherer, W. N. (2008). Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA. ACM.

[113] Shirako, J., Zhao, J. M., Nandivada, V. K., and Sarkar, V. N. (2009). Chunking parallel loops in the presence of synchronization. In *International Conference on Supercomputing (ICS)*, pages 181–192.

[114] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1998). *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition.

[115] Stark, D. T., Barrett, R. F., Grant, R. E., Olivier, S. L., Pedretti, K. T., and Vaughan, C. T. (2014). Early experiences co-scheduling work and communication tasks for hybrid mpi+x applications. In *Workshop on Exascale MPI at Supercomputing Conference*, pages 9–19.

[116] Subramanian, S., Jeffrey, M. C., Abeydeera, M., Lee, H. R., Ying, V. A., Emer, J., and Sanchez, D. (2017). Fractal: An execution model for fine-grain nested speculative parallelism. In *International Symposium on Computer Architecture (ISCA)*, pages 587–599.

[117] Suleman, M. A., Mutlu, O., Qureshi, M. K., and Patt, Y. N. (2009). Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, pages 253–264.

[118] Tan, X., Bosch, J., Vidal, M., Álvarez, C., Jiménez-González, D., Ayguadé, E., and Valero, M. (2017). General purpose task-dependence management hardware for task-based dataflow programming models. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 244–253.

[119] Tendler, J. M., Dodson, J. S., Jr., J. S. F., Le, H. Q., and Sinharoy, B. (2002). Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26.

[120] Thakur, R., Rabenseifner, R., and Gropp, W. (2005). Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66.

[121] The Ohio State University (2018). Mvapich2: Mpi over infiniband, 10gige/iwarp and roce.

[122] Topcuouglu, H., Hariri, S., and Wu, M.-y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

[123] Tullsen, D. M., Eggers, S. J., and Levy, H. M. (1998). Simultaneous multithreading: Maximizing on-chip parallelism. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 533–544, New York, NY, USA. ACM.

[124] Vandierendonck, H., Pratikakis, P., and Nikolopoulos, D. S. (2011). Parallel programming of general-purpose programs using task-based programming models. In *HotPar*, pages 13–13.

[125] Vega, A., Buyuktosunoglu, A., Hanson, H., Bose, P., and Ramani, S. (2013). Crank it up or dial it down: coordinated multiprocessor frequency and folding control. In *MICRO*, pages 210–221.

[126] Wang, Y., Wang, R., Herdrich, A., Tsai, J., and Solihin, Y. (2016). CAF: Core to core communication acceleration framework. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 351–362.

[127] Wong, H., Bracy, A., Schuchman, E., Aamodt, T. M., Collins, J. D., Wang, P. H., Chinya, G., Groen, A. K., Jiang, H., and Wang, H. (2008). Pangaea: A tightly-coupled ia32 heterogeneous chip multiprocessor. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 52–61.

[128] Zuckerman, S., Suetterlein, J., Knauerhase, R., and Gao, G. R. (2011). Using a "Codelet" program execution model for exascale machines. In *International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 64–69.

List of Figures

List of Tables

Glossary

AMC	Asymmetric multi-core.
AMPI	Adaptive message passing interface.
ATaP	Asynchronous task-based programming.
AVG	Average (geometric mean).
BL	Bottom-level.
CATA	Criticality aware task acceleration.
CATS	Criticality-aware task scheduling.
CDG	Codelete graph.
CMP	Chip multiprocessor.
DAT	Dependence alias table.
DMU	Dependence management unit.
DVFS	Dynamic voltage and frequency scaling.
EDP	Energy-delay product.
FFT	Fast Fourier transform.
FIFO	First in, first out.
HJM	Heath Jarrow Morton.
HPC	High performance computing.
HPRQ	High-priority ready queue.
ILP	Instruction-level parallelism.
IPC	Instructions per cycle.
LA	List array.
LIFO	Last in, first out.
LPRQ	Low-priority ready queue.
LSQ	Load-store queue.
MPI	Message passing interface.
NIC	Network interface card.
NUMA	Non-uniform memory access.
RAA	Runtime-aware architecture.
RQ	Ready queue.
RSM	Reconfiguration support module.
RSU	Runtime support unit.
SLTs	System-level threads.
SMT	Simultaneous multithreading.
SPEs	Cell synergistic processing units.
TAMPI	Task-aware message passing interface.
TAT	Task alias table.
TBB	Thread building blocks.
TDG	Task dependence graph.
TDM	Task dependence manager.
TLP	Thread-level parallelism.
ULTs	User-level trheads.