POLITECNICO DI MILANO
Dipartimento di Elettronica, Informazione e Bioingegneria
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

Harnessing Adaptivity Analysis for the Automatic Design of Efficient Embedded and HPC Systems

Doctoral Dissertation of:
Silvia Lovergine

Advisor:
Prof. Fabrizio Ferrandi
Tutor:
Prof. Donatella Sciuto
Supervisor of the Doctoral Program:
Prof. Carlo Ettore Fiorini

2013 – XXVI

POLITECNICO DI MILANO
Dipartimento di Elettronica, Informazione e Bioingegneria
Piazza Leonardo da Vinci, 32 I-20133 — Milano

Abstract

Embedded Systems (ESs) and High-Performance Computing (HPC) systems belong to two distinct areas of the Information Technology (IT). On one side, embedded systems are usually designed as hardware implementation of a specific functionality, with the aim of either reducing power consumption, or accelerating the most frequently executed portion of an application. On the other side, high-performance computing systems are designed as massively parallel supercomputers with tens of thousands of processors, usually employed to solve complex, highly parallel scientific problems. Due to the distinct nature of the two domains, these systems have historically evolved following different trends.

However, many issues and design challenges are lately arising, which affect both the domains. Among the examples, increasing computational demands and large parallelism degrees, together with the growing capabilities of silicon technology, led to the design of increasingly complex architectures and applications. As a consequence, modern embedded systems exploit the potentialities of hundreds or thousands of processing units, often heterogeneous and physically distributed, which run in parallel on the many-core platform. These same factors led to massively parallel systems in HPC domain. As another example, power constraints, which have always been a critical requirement for embedded systems design, are becoming more and more relevant also in the high-performance domain. As a consequence, communication minimization techniques are required also for modern HPC systems design. This trend suggests that HPC systems design techniques can be exploited, at a different abstraction level, by embedded systems designers to address shared issues, and vice versa. In other words, designing modern supercomputers, as well as modern embedded systems, requires a holistic approach that relies on tightly coupled hardware-software co-design methodologies.

Among the numerous issues which are shared between the ES and the HPC domains, this thesis focuses on unknown, uncertain and unpredictable behaviors. Modern computing systems have to deal with this kind of behaviors, which are often related to incomplete information at design/compiler time. Unknown, uncertain and unpredictable information can be originated by different sources. For example, the interaction with external modules (e.g., sensors, IPs, memories) generates variable and possibly unpredictable communication latency. Moreover, the nature of high-level applications is inherent uncertain (e.g., unknown number of loop iterations, unknown values of the incoming

arguments of a function, unknown outcome of conditional instructions evaluation, or unpredictable memory accesses). Incomplete information at design or compile time often leads to suboptimal designs: to prevent the system from a failure, the designer must either consider all the possible scenarios, with consequent increase in area and complexity of the circuit, or take a conservative approach, with consequent decrease in performance. Moreover, this approach is error prone, since the correctness cannot be guaranteed in all the situations (such as run-time events which the designer cannot predict, or technology parameters which change unpredictably during the life cycle of the device, for example due to component degradation). In this scenario, systems able to dynamically adjust their behavior at run-time appear to be good candidates for the next computing generation in both the domains.

This work aims at defining efficient design methodologies for modern embedded systems and high-performance computing systems, able to deal with unknown, uncertain and unpredictable behaviors. For such purpose, this thesis introduces the concept of *Adaptivity Analysis*, which provides a formal approach to study the adaptivity properties of the applications. Given the different scale of the problem in the ESs and HPCs domain, Adaptivity Analysis is defined at two distinct abstraction levels.

In the embedded systems domain, Adaptivity Analysis addresses the problem of designing efficient adaptive hardware cores. More in detail, this analysis identifies, at design-time, the conditions that each instruction (or group of instructions) must satisfy to execute, according to their dependences. Such conditions, called *Activating Conditions (ACs)*, are logic formulas. Since unknown information may occur at design-time, the ACs provide parametric results, which depend on conditions that will be resolved at run-time (e.g., outcome of the evaluation of a conditional instruction). At run-time, as soon as the unknown information is resolved, the ACs are evaluated. Once an AC is satisfied, the corresponding instruction can safely execute, thus providing an explicit activation mechanism for the instruction, which allows their dynamic auto-scheduling. Such a scheduling technique, called dynamic AC-scheduling, provides support for the High-Level Synthesis (HLS) of adaptive hardware cores. Moreover, this thesis defines a prototype for this kind of system. Furthermore, this work defines a proper Intermediate Representation (IR), called Extended Program Dependence Graph (EPDG), able to provide a parallel execution model for adaptive hardware cores. Finally, the dynamic AC-scheduling has been implemented as part of the PandA framework, thus providing a fully automated design methodology.

In the HPC domain, Adaptivity Analysis is used as automatic compiler-based generation of parallel irregular applications. In HPC design, run-time systems are entitled for managing unknown, uncertain and unpredictable behaviors. Hence, adaptivity analysis is used in this context as a compiler technique for the support of run-time systems. More in detail, this work focuses

on the automatic parallelization of a particular class of applications, known as Irregular Applications. Due to their irregular nature (i.e., irregular data structures, irregular control flow, irregular accesses to the communication network), the parallelization of irregular applications is significantly challenging. This thesis introduces the Yet Another Parallel Programming Approach (YAPPA) compilation framework. YAPPA aims at efficiently parallelizing irregular applications running on commodity clusters. The novel parallel programming approach supported by YAPPA is defined by the underlying run-time library, called Global Threading and Memory (GMT). GMT cross combines different parallel programming approaches, which are usually exploited separately in the literature: it integrates Global Address Space (GAS) across cluster nodes, lightweight multithreading for tolerating memory and network latencies, and support for a fork/join programming model. YAPPA extends the LLVM compiler with a set of transformations and optimizations, which at first instrument the sequential code to run GMT primitives (parallelization phase), and then apply a novel set of transformations to improve the efficiency of the generated parallel code (optimization phase).

Estratto

Embedded Systems (ESs) e High-Performance Computing (HPC) systems appartengono a due aree distinte dell'Information Technology (IT). Da un lato, i sistemi embedded sono di solito progettati come implementazioni hardware di funzionalità specifiche, con lo scopo di ridurre il consumo di potenza, o di accelerare l'esecuzione delle porzioni di codice che vengono eseguite più di frequente. Dall'altro lato, i sistemi high-performance computing sono progettati come supercomputer massicciamente paralleli, con decine di migliaia di processori, e vengono di solito impiegati per risolvere problemi scientifici complessi ed altamente paralleli. Per via della diversa natura dei due domini, l'evoluzione di questi sistemi ha storicamente seguito strade divergenti.

Ultimamente, tuttavia, un numero sempre più consistente di problematiche sta interessando entrambi i domini. Tra gli esempi, la crescente domanda di potenza computazionale, assieme ai miglioramenti nella tecnologia del silicio, hanno portato allo sviluppo di architetture ed applicazioni sempre più complesse. Di conseguenza, i sistemi embedded moderni sfruttano le potenzialità di migliaia di unità di elaborazione, spesso eterogenee e fisicamente distribuite, che lavorano in parallelo sulla piattaforma many-core. Questi stessi fattori hanno portato allo sviluppo di sistemi altamente paralleli nel dominio HPC. I vincoli di potenza sono un altro esempio di problematiche comuni. Da sempre un requisito stretto per la progettazione di sistemi embedded, i vincoli di potenza stanno diventando sempre più rilevanti anche nel dominio HPC. Di conseguenza, la progettazione dei sistemi HPC di oggi richiede tecniche di minimizzazione della comunicazione. Questa tendenza suggerisce che le tecniche di progetto tipiche dei sistemi high-performance computing possano essere sfruttate, ad un diverso livello di astrazione, per il progetto di sistemi embedded, e viceversa. In altre parole, la progettazione dei supercomputer moderni, così come quella dei sistemi embedded moderni, richiede un approccio olistico basato su metodologie di co-design hardware-software strettamente accoppiate.

Tra i numerosi problemi condivisi dai domini ES e HPC, questa tesi si concentra su comportamenti incogniti, incerti e imprevedibili. Le architetture di oggi devono fare i conti con questo tipo di comportamenti, che spesso sono legati alla presenza di informazione incompleta a design/compile time. Informazione incognita, incerta o imprevedibile può essere originata da diverse fonti. Per esempio, l'interazione con moduli esterni (es., sensori, IPs, memorie) genera latenza di comunicazione variabile e potenzialmente imprevedibile.

Inoltre, le applicazioni scritte in linguaggi ad alto livello hanno natura intrinsecamente incerta (es., numero incerto di iterazioni di un loop, valore incognito dei parametri di ingresso di una funzione, risultato incognito della valutazione di una istruzione condizionale, o accessi a memoria impredicibili). La presenza di informazioni incomplete a design o compile time comporta risultati subottimi: per prevenire errori, il progettista deve considerare tutti i possibili scenari, con conseguente aumento in area e complessità del circuito. Alternativamente, è possibile considerare l'approccio conservativo, con conseguente perdita in performance. Oltretutto, questo approccio è soggetto ad errori, in quanto la correttezza del design prodotto non può essere garantita in tutte le situazioni (es., eventi dinamici che il progettista non può prevedere, o cambiamenti nei valori dei parametri tecnologici durante il ciclo di vita del dispositivo, ad esempio a causa di degradazione dei componenti). In questo scenario, i sistemi che sono in grado di adattare dinamicamente il proprio comportamento a run time sembrano essere buoni candidati per la prossima generazione computazionale in entrambi i domini.

Questo lavoro mira a definire metodologie di progetto efficienti per sistemi embedded e high-performance computing moderni, capaci di gestire comportamenti incogniti, incerti e imprevedibili. A tal proposito, questa tesi introduce il concetto di Adaptivity Analysis, la quale fornisce un approccio formale per lo studio delle proprietà adattative delle applicazioni. Data la diversa scala del problema nei due domini, l'Adaptivity Analysis è definita a due diversi livelli di astrazione.

Nel dominio dei sistemi embedded, la tecnica di Adaptivity Analysis affronta il problema della progettazione efficiente di hardware core adattativi. Più in dettaglio, questa analisi identifica, a design time, le condizioni che ciascuna istruzione (o gruppo di istruzioni) deve soddisfare per poter essere eseguita, a seconda delle sue dipendenze. Queste condizioni, chiamate Activating Conditions (ACs), sono formule logiche. Dato che alcune informazioni a design time possono essere incognite, le AC forniscono risultati parametrici, dipendenti da condizioni che saranno note a run time (es., risultato della valutazione di una istruzione condizionale). A run time, non appena le informazioni incognite diventano note, le AC vengono valutate. Quando una AC è soddisfatta, l'istruzione ad essa associata può essere eseguita in modo sicuro, fornendo quindi un meccanismo di attivazione esplicito delle istruzioni, che permette il loro auto-scheduling dinamico. Questa tecnica di scheduling, chiamata dynamic AC-scheduling, rappresenta il supporto per la sintesi ad alto livello di hardware core adattativi. Inoltre, questa tesi definisce un prototipo per questo tipo di sistemi. Oltretutto, questo lavoro definisce una rappresentazione intermedia opportuna, chiamata Extended Program Dependence Graph (EPDG), capace di fornire un modello di esecuzione parallelo per hardware core adattativi. Infine, la tecnica di dynamic AC-scheduling è stata implementata come parte del framework PandA, quindi fornendo una metodologia di progetto interamente automatizzata.

Nel dominio HPC, la tecnica di Adaptivity Analysis viene usata per la generazione automatica di codice parallelo basata su compiletore, per applicazioni irregolari. Nell'ambito HPC, i sistemi run time sono intitolati per la gestione di comportamenti incogniti, incerti e imprevedibili. Quindi, l'Adaptivity Analysis è usata in questo contesto come tecnica di compilazione per il supporto di sistemi run time. In particolare, questo lavoro si concentra sulla parallelizzazione automatica di una specifica classe di applicazioni, note come applicazioni irregolari. Per via della loro natura irregolare (es., strutture dati irregolari, flusso di controllo irregolare, accessi irregoalri alla rete di comunicazione), la parallelizzazione di applicazioni irregoari risulta particolarmente complessa. Questa tesi introduce il framework di compilazione Yet Another Parallel Programming Approach (YAPPA). YAPPA mira a parallelizzare efficientemente applicazioni irregolari su commodity cluster. L'innovativo approccio di programmazione parallela supportato da YAPPA è definito dalla sottostante libreria di run time, chiamata Global Threading and Memory (GMT). GMT combina differenti approcci di programmazione parallela, che sono di solito utilizzati separatamente in letteratura: integra Global Address Space (GAS) sui nodi del cluster, lightweight multithreading per tollerare latenze di memoria e di rete, e supporto per un modello di programmazione fork/join. YAPPA estende il compilatore LLVM con una serie di trasformazioni e ottimizzazioni, che innanzitutto instrumentano il codice sequenziale in modo da supportare primitive GMT (fase di parallelizzazione), e poi applicano un insieme di nuove trasformazioni per migliorare l'efficienza del codice parallelo generato (fase di ottimizzazione).

Contents

List of Figures

List of Tables

1 Introduction

In the past decades, design methodologies of Embedded Systems (ES) and High Performance Computing (HPC) systems have evolved following different trends. On one hand, Embedded Systems have been usually designed either as custom standalone devices dedicated to the execution of specific tasks (e.g., guidance computers in aerospace industry), or as custom integrated devices with a dedicated function within a larger mechanical or electrical system (e.g., microwave, anti-lock breaking systems in automotive, biomedical instrumentation). Embedded Systems have been largely employed either in real-time systems, to meet performance constraints for safety and usability reasons, and in systems with low performance requirements, to reduce the costs by simplifying the hardware. On the other hand, since the 1960s, HPC systems have been designed as machines at the front-line of contemporary processing capacity. Originally designed with only few processors, by the end of the 20th century, massively parallel supercomputers with tens of thousands of "off-the-shelf" processors became the norm. HPC systems have been extensively employed for a wide range of computationally intensive tasks in various fields (e.g., quantum mechanics, weather forecasting, molecular modeling, physical simulations, machine learning).

However, these two domains are lately experiencing common issues, whose solutions converge to similar approaches. Indeed, the literature shows numerous examples of issues and design challenges which are common to modern embedded and HPC systems [115]. For example, increasing computational demands and large parallelism degrees play a dominant role, especially in real-time applications. These factors led to distributed many-core platforms in embedded systems design, and to massively parallel systems in HPC domain. Another example is the introduction of heterogeneous processors, specialized for different workloads, which often lead to the integration of embedded systems in complex HPC systems. Power constraints are becoming more and more relevant for both classes of systems. Communication minimization, which is fundamental for reducing power consumption on embedded processors, is now a stringent requirement for supercomputer design at the chip, node, and machine level. Furthermore, the scaling down of technology, which resulted in process variation as well as component degradation, introduced the need of designing more flexible systems, able to deal with unpredictable behaviors (adaptivity) [40]. Finally, the design process of such increasingly complex architectures requires great skills and significant programming effort, being time-consuming and error-prone. Modern computing systems contain a heterogeneous mixture of processing cores (e.g., FPGAs, GP-GPUs, DSPs) combined with a single or multi/many core CPUs. Achieving the desired performance on such increasingly complex heterogeneous systems is not trivial. Thus, improvements in the design automation process is a challenge for both ESs and HPCs. Most of the approaches used for the design of embedded systems have been also applied, at a different scale, to HPC systems [54]. Recent research in supercomputing believes that, to design next generation supercomputers, we have to know embedded systems and to take inspiration from their design methodologies [90]. Thanks to the introduction of more compact and more powerful embedded processors, embedded systems are becoming HPC capable, and the two domains are converging to a new generation of computing platforms, called *embedded supercomputing systems* (e.g., MontBlanc Project [25], [66]). Designing modern supercomputers, as well as modern ESs, requires a holistic approach that relies on tightly coupled hardware-software co-design methodologies [70].

Modern embedded supercomputers necessarily have to deal with several sources of unknown, uncertain and unpredictable information, due, for example, to the interaction with external modules, such as sensors (e.g., real-time data processing for home security systems, body imaging scanners in medical diagnostic, or anti-collision systems in automotive) or memories, which present variable latency, especially considering the increasing complexity of their hierarchies. Moreover, applications show inherent unknown or uncertain information (e.g., unknown number of loop iterations, unknown values of the incoming arguments of a function, unknown outcome of conditional instructions evaluation, or unpredictable memory accesses), which complicates many design choices, such as instructions/tasks scheduling, partitioning of the applications into tasks and consequent mapping on the available computational resources. In this scenario, systems able to dynamically adjust their behavior at run-time appear to be good candidates for the next computing generation, and will most probably condemn non-adaptable systems to rapid extinction. Adaptive systems are able to deal with (and dynamically adapt to) uncertain and unpredictable conditions, due, for example, to reliability issues, inherent uncertainty of an application or unpredictable communication latency due to external modules.

Our society is experiencing a new era of data explosion in all the domains of computing systems. This explosion of data, known as *Data Deluge* [64], together with the increase in the use of unstructured data, is leading to growing computational requirements and demanding of new computing approaches. The demand is growing faster than the technology can keep up. On one side, the increasing computational requirements mirror another important limitation, i.e. power efficiency. The number of devices, packed on a chip, that can be simultaneously used is restricted by the power used by each device, which does no longer drop accordingly. This issue is contributing to the increasing popularity of complex systems, which take advantage of many computational resources, often heterogeneous. On the other side, Data Deluge led to the introduction of a particular class of applications, known as *irregular applications*. Applications can be irregular because of data structures (e.g., unbalanced trees, graphs, unstructured grids), control flow (i.e., conditional statements) or communication patterns. They present unpredictable memory and network accesses, which lead to very poor spatial and temporal locality and make ineffective big and complex caches. They have very big datasets, difficult to partition without generating load unbalance, and require fine-grained synchronization. Unfortunately, current HPC systems and embedded supercomputing systems are optimized for data locality, easily partitionable datasets, and bulk synchronization. Developing irregular applications on these systems demands a substantial effort, often not leading to the desired performance.

Adaptivity is a fundamental property of a system, used to face different problems in different domains. Such problems range from ensuring correct execution in presence of unknown and possibly unpredictable information, to guaranteeing high performance, or even to balance the computational workload among the nodes of multi or many core architectures. Adaptivity is a different concept with respect to self-adaptation. In the literature we can find different reference to the term self-adaptation, which may refer to different kinds of run-time adjustments, such as run-time (possibly partial) reconfigurability of a hardware core [93], or dynamic adjustment of supply voltage and clock frequency to minimize power consumption (i.e. Dynamic Voltage/Frequency Scaling - DVFS [123]), as well as Dynamic Power Management (DPM) [38], or even dynamic sizing of memories [55]. In all these cases, self-adaption is used to describe adaptive hardware, i.e. systems where the hardware can be modified at run-time in a (semi) automatic fashion.

This thesis work introduces adaptivity analysis and its use to address several design challenges in both embedded and HPC domains [129] [95]. It defines adaptivity as a property of an application to adapt to unpredictable events. In the embedded systems domain, adaptivity analysis identifies, at compile-time, the conditions that each instruction/group of instructions (according to the desired granularity level) must satisfy to execute. It provides parametric results, which depend on conditions that are resolved at run-time, since unknown information may occur at compile-time [131]. Despite adaptivity analysis could be theoretically used to modify the hardware, in this thesis work it is used as a software analysis for the design of flexible architectures, which remain fixed during their life time, while embedding support for unknown, uncertain or unpredictable events which may occur at run-time. Thus, in the ES domain, adaptivity analysis enables run-time instructions auto-scheduling [131] [132], thus providing the support for the automatic design of adaptive hardware cores [40]. These cores are able to deal with modules with variable and possibly unknown latency. The proposed approach relies on the computation of a set of conditions, namely the instructions Activating Conditions (ACs), under which each instruction can be executed. The ACs are evaluated at run-time, enabling the auto-scheduling of the instructions. Furthermore, this thesis work discusses how to use adaptivity analysis to address new challenges in HPC systems design. In the HPC domain, adaptivity is usually managed by run-time systems. Guided by the idea that run-times can benefit by compiler support, this thesis work defines a set of transformations and optimizations aiming to generate optimized parallel code for a proper run-time system. It focuses on irregular applications, by studying compiler transformation and optimizations to efficiently deal with the many sources of uncertainty and unpredictability which are typical of this class of applications. The choice of the IR significantly affects the performance of the system. It employs the LLVM intermediate representation to study the adaptivity level of irregular applications [130].

The main contribution of this thesis work can be summarized as follows:

- **Definition of Adaptivity Analysis in the ES Domain**: Adaptivity Analysis relies on the concept of *Activating Conditions* (ACs) to handle uncertain/unpredictable information, such as, for example, number of loop iterations, outcome of conditional instructions evaluation, value of the incoming arguments of a function or unknown latency of an external module. The ACs are logic formulas that indicate which conditions must be satisfied at run-time to correctly enable the execution of each instruction. They provide an explicit activation mechanism for the instruction, thus also supporting unknown latency of modules. Due to the sequential nature of the high level language describing the specification, the first step to identify the ACs consists in the definition of a proper IR, able to expose the inherent parallelism. The Program Dependence Graph (PDG) [62] can serve as a starting point for the purpose, since it does not include control flow edges causing potentially unnecessary sequencing (e.g., an edge between the two conditional instructions of two independent loops in the CFG). However, it does not contain control flow edges which are instead necessary to provide a parallel execution model and to guarantee correct execution. For example, if a data dependence occurs between an instruction i belonging to a loop $loop_i$ and an instruction j outside the loop, the PDG does not show that j must wait for $loop_i$ termination before to execute. The proposed methodology defines the *Extended Program Dependence Graph (EPDG)*, which extends the PDG by adding the minimum control flow edges. In the previous example, a minimum control flow edge between the loop exit instruction(s) of $loop_i$ is added, i.e. the branch instruction(s), and j, with label equal to the loop exit condition(s), to force j to wait for the termination of $loop_i$. This thesis work provides a formal definition of EPDG, a definition of minimum control flow edges, and a methodology for the EPDG construction. The EPDG is analyzed to compute the ACs, according to a set of formal rules. The ACs indicate which conditions must be satisfied at run-time to correctly enable the execution of each instruction. Such conditions depend not only on minimum data and control dependencies, but also on minimum control-flow dependences. This thesis work defines the formal rules for ACs computation as composed of three parts. The first part handles minimum data dependences; it also takes into account data dependences among instructions belonging to different control regions. The second part manages minimum control dependences; it takes into account that at each computation only the execution of a single control instruction c_j will cause the execution of instruction i, even if i has multiple control dependences c_j, with $j = 1, ..., n$. The third part handles minimum control flow dependences, due, for example, to loop back-edges, data dependences among instruction of different nested loops, or even loop-carried data dependences. Properly defined operators are responsible to combine together these three parts.

- **Design Methodology for Efficient Adaptive Embedded Systems**: This thesis work provides a methodology to automate the design of adaptive hardware cores, starting from a C-like behavioral specification. Supporting instructions auto-scheduling allows overcoming restrictions due to unknown or uncertain information at design time, with a limited area overhead, while simultaneously increasing parallelism exploitation. Firstly, the proposed approach defined a proper architectural model for self-adaptive hardware cores [40]. Such model is composed of a datapath and a lightweight token-based controller, which does not introduce communication overhead. Then, it automated the ACs computation process, and integrated it into a synthesis flow.

- **Compiler Support for Run-Time Systems in HPC Design**: Adaptivity analysis is applied to the HPC domain as a support to run-time systems, which are entitled to manage unknown information. More in detail, the proposed technique provides automatic generation of parallel code on modern HPC systems with heterogeneous, multi- and many-core architectures, targeting irregular applications. The YAPPA compilation framework [130] has been developed for supporting irregular applications on commodity clusters which integrates global address space across cluster nodes, lightweight multithreading for tolerating memory and network latencies, and support for a fork/join programming model. Adaptivity analysis is used at the compiler level for implementing a set of transformations and optimizations. The LLVM compiler has been extended for such purpose. Since irregular applications often have large amounts of parallelism in form of loops (e.g., graph visiting algorithms), they are strong candidates for thread partitioning. However, techniques to establish the partitioning sizes are required to avoid resource saturation. Such methodology should be dynamic, to best adapt to changes in computational demands and memory requirements. The compiler can provide hints to the run-time system for such purpose. The adaptivity level of an application is studied to generate and optimize efficient parallel code for irregular applications, which have to deal with many sources of unpredictability.

The main contributions in terms of publications are:

- C. Pilato, V. G. Castellana, **S. Lovergine**, and F. Ferrandi. "A Runtime Adaptive Controller for Supporting Hardware Components with Variable Latency". In in Proceedings of International NASA/ESA Conference on Adaptive Hardware and Systems, pages 153-160, 2011.

- **S. Lovergine** and F. Ferrandi. "Instructions Activating Conditions for Hardware-Based Auto-Scheduling". In Proceedings of International Conference on Computing Frontiers, pages 253-256, 2012.

- **S. Lovergine** and F. Ferrandi. "Dynamic AC-Scheduling for Hardware Cores with Unknown and Uncertain Information". In in Proceedings of 31st IEEE International Conference on Computer Design, pages 475-478, 2013.

- **S. Lovergine** and F. Ferrandi. "Harnessing Adaptivity Analysis for the Automatic Design of Efficient Embedded and HPC Systems". In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW '13, pages 2298-2301, Boston, Massachusetts, USA, 2013. IEEE Computer Society.

- **S. Lovergine**, A. Tumeo, O. Villa, and F. Ferrandi. "Parallelizing Irregular Applications through the YAPPA Compilation Framework". Poster at SuperComputing (SC13), 2013.

- **S. Lovergine**, A. Tumeo, O. Villa, and F. Ferrandi. "YAPPA: a Compiler-Based Parallelization Framework for Irregular Applications on MPSoCs". In in Proceedings of International Symposium on Rapid System Prototyping, pages 123-129, 2013.

The remainder of this thesis is organized as follows. Chapter 2 discusses some related works for the design methodologies of embedded systems and HPC systems. Chapter 3 outlines the current trend in the design of modern computing systems and describes the common design challenges for Embedded and HPC systems. Chapter 4 presents the proposed methodology for the design of adaptive embedded systems, based on the instructions Activating Conditions. Moreover, it examines the capabilities of the hardware cores resulting from the proposed methodology for ES automatic design, compared to classical ones, i.e. finite state machine. The results show an average gain in performance of more than 34%. Chapter 5 outlines how to apply adaptivity analysis to the HPC domain. It also evaluates the benefits of using adaptivity analysis for the design of modern HPC systems, by means of a case study: the Breadth First Search (BFS) algorithm. Finally, Chapter 6 concludes outlining possible future directions for the work.

2 State of the Art

This Chapter summarizes some methodologies proposed in the literature for the design of modern computing systems. Particular attention is devoted to various forms of unknown, uncertain and unpredictable information and to how these issues are addressed by conventional design approaches. More in detail, Section 2.1 introduces common design methodologies of modern embedded systems, while Section 2.2 describes traditional design approaches for modern HPC systems. Finally, Section 2.3 discusses how unknown, uncertain and unpredictable information affects the design of both efficient embedded systems and supercomputers.

2.1 Design Methodologies of Modern Embedded Systems

Traditionally, the design of embedded systems followed two main approaches, which differ from each other for the type of input language used to describe the specification. In the first case, the hardware designer exploits the functionality of some Hardware Description Language (HDL), such as Verilog and VHDL. The HDL code represents the input of a logic synthesis tool, which produces the Register-Transfer Level (RTL) code. In the past, designers used to hand-write directly the RTL code. However, such approach was soon abandoned due to its complexity and thanks to the improvements in logic synthesis tools. The second approach uses High-Level Languages, such as C, C++, or SystemC, to describe the desired functionality. Such high-level application is then provided as input for a High-Level Synthesis (HLS) tool, which produces the HDL code, which in turn is used as input for a logic synthesis tool to generate the RTL. Over the years, HLS has received more and more attention by the designers, due to the growing capabilities of silicon technology and to the increasing complexity of applications and architectures. Among the reasons for the success of HLS, the possibility to reduce the design time, thus increasing the chance for companies of hitting the market window. HLS automates the design process, thus significantly decreasing the development cost. The design error rate is reduced by the presence of a proper verification phase. Working at a higher level of the design hierarchy using high-level synthesis reduces the amount of code that must be developed by as much as two thirds [101]. Indeed, separating the design intent from the physical implementation avoids the tedious process of rewriting and retesting code to make architectural changes. This fact also facilitates the design space exploration process since a good synthesis system produces several designs for the same specification in a reasonable amount of time, allowing the developers to consider different solutions and trade-offs. Generally, it is reported an overall reduction of the design effort, with respect to lower level methodologies, of 50% or more [101]. Despite the main advantages of high-level synthesis, there are still many aspects requiring improvements. Indeed, the quality of the results produced by HLS tools is usually not comparable with those obtained by hand-writing the RTL, especially for those class of applications which show irregular control flow. In general, this gap increases as the specification shows unpredictable behaviors, and as the designer has to deal with incomplete information at design time. The following of this Section briefly illustrates the typical phases of the high-level synthesis process.

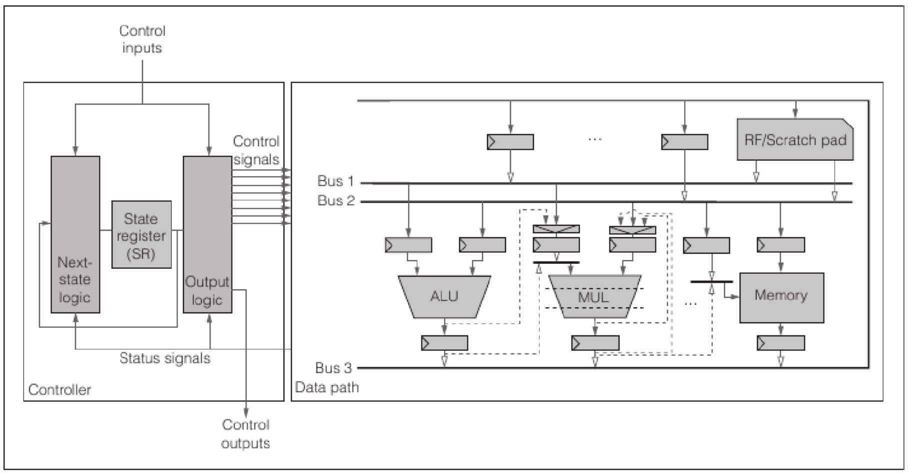


Figure 2.1: Typical architecture of a hardware core produced by a HLS tool.

High-Level Synthesis The term High-Level Synthesis (HLS) indicates a design process which, given an abstract (unclocked, or partially clocked) behavioral specification of a digital system (e.g., C, C++, SystemC description) and a set of constraints (e.g., timing, area, power), automatically generates a Register-Transfer Level (RTL) structure that implements the desired behavior. Typically the output of the HLS process consists in two parts: a *datapath*, which consists of the modules, registers and multiplexers, properly interconnected to each other, needed to perform the actual computation; and a *controller* (typically a centralized Finite State Machine - FSM), which provides the logic to issue the operations in the right order. Figure 2.1 shows the typical architecture of an hardware core produced by high-level synthesis.

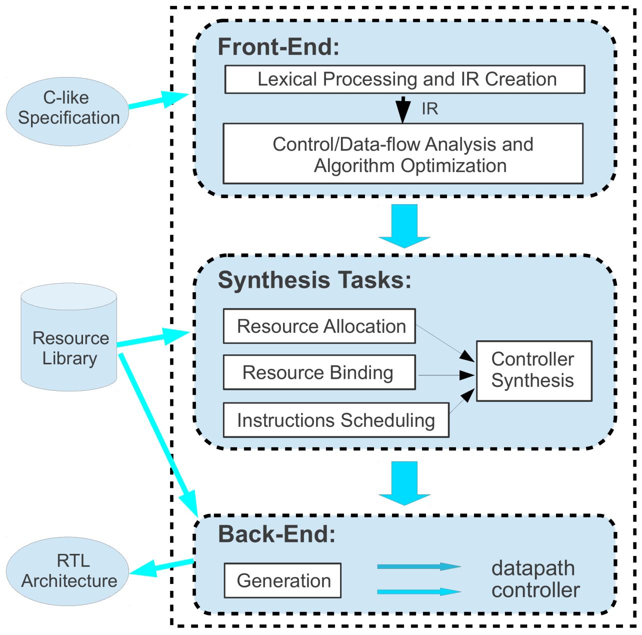


Figure 2.2: Typical High-Level Synthesis Flow.

High-level synthesis is typically composed of different tasks, as shown in Figure 2.2. There exist many approaches in literature that perform these activities in different orders, using different algorithms. In some cases, several tasks can be combined together or performed iteratively to reach the desired performance. In general, the HLS flow steps can be grouped in three main macro-tasks:

- **Front-End**: this macro-step includes tasks such as lexical processing, algorithm optimization and control/dataflow analysis. It takes as input a C-like specification and produce as output an Intermediate Representation (IR), which will be used by the subsequent steps. The IR represents the original specification in a different form (usually graphs), which better highlights specific properties of the program, with respect to the original version. For example, IRs are employed to show the inherent parallelism of an application, which is hided by high-level constructs in C-like specifications.

- **Synthesis**: this phase includes the main synthesis tasks characterizing HLS, which take into account the specified design constraints, while exploiting the IRs produced by the previous steps for the analysis. The main tasks in HLS are: instructions scheduling, resources allocation and resources binding. The scheduling task assigns a control step to each instruction. The resources allocation task establishes number and type of functional units, registers and multiplexers to allocate in the circuit, according to the available resources, described in a resource library. Finally, the resources binding task maps the instructions and variables to the hardware components, multiplexers, registers and wires of the datapath. All these information are used to define the states of the FSM controller.

- **Back-End**: this last step takes as input the information computed by the previous two steps to implement the original functionality into hardware, according to the description of the resources provided by the resource library. The output is a RTL description of the circuit.

The remainder of this Section describes some traditional techniques and approaches related to the design of modern embedded systems. The main goal of this thesis work in the embedded systems domain is to propose a novel High-Level Synthesis (HLS) approach, which exploits *adaptivity analysis* to implement *dynamic scheduling* of flexible and *adaptive hardware cores*. Moreover, it is worth notice that the quality of the results provided by HLS is significantly affected by the choice of the Intermediate Representation (IR). For these reasons, this Section concentrates on the following topics:

- *high level languages and Intermediate Representation (IR)*: the most common IRs adopted in HLS are reviewed in Section 2.1.1, illustrating their features from the adaptivity and performance perspectives. IRs are often employed to show some features of the program which high-level languages are not able to highlight

- *scheduling methodologies*: some scheduling approaches in high-level synthesis are examined in Section 2.1.2, highlighting their pros and cons in terms of uncertainty support by means of a motivating example

- *adaptivity*: Section 2.1.3 discusses how other forms of adaptivity analysis have been employed in the literature for the design of embedded systems

- *HLS Architectures*: handling unpredictable events requires the definition of flexible HLS architectures, due to limitations in standard architectures, as shown in Section 2.1.4

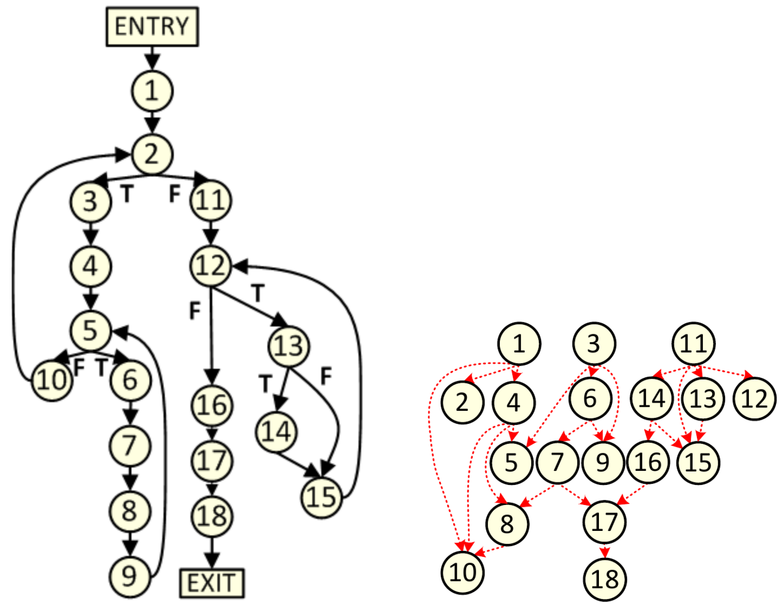
- *HLS tools*: Section 2.1.5 describes the features of some industrial and academic HLS tools

2.1.1 High Level Languages and Intermediate Representations (IRs)

High-level synthesis is usually based on the analysis of one or more Intermediate Representations (IRs), which represent the original program in an equivalent form, more suitable for exposing applications features, such as parallelism level. It is worth notice that the parallelism level of an application depends on information which is potentially unknown at design time. The sequential nature of C-like languages, used for describing the specification, mismatches with the inherent parallel nature of hardware cores. For this reason hardware designers use to represent the original specification in an equivalent form, called Intermediate Representation (IR). Since this thesis work proposes a novel dynamic scheduling technique to manage unknown information, the following of this Section focuses on the choice of a proper IR for the *scheduling* of embedded systems in HLS. The choice of the IR may significant affect the performance of the scheduling algorithm. The most common approach for the scheduling of embedded systems employs the Control-Data Flow Graph (CDFG) [32] as IR [65] [52] [51] [69]. The CDFG is a Basic Block graph, built by considering the data dependences in the Data Dependence Graph (DDG) and the control dependences in the Control Dependences Graph (CDG). Usually, the use of the CDFG is coupled with a list scheduling algorithm [65], which statically assigns, at design time, a control step to each instruction, from which the centralized FSM is obtained. This approach cannot deal with unpredictable components or behaviors. Indeed, while assigning a control step to each instruction and building the FSM, it is necessary to conservatively consider the worst case latency for instructions with variable latency, provided that it is known or predictable. Otherwise, as in the case of process variation, an error can occur during the execution. The CDFG exposes the parallelism at the basic block level, i.e. groups of instructions belonging to different basic blocks are sequentialized, while instructions belonging to the same basic block can be simultaneously executed, if they are data independent. As a consequence, such approach well fits the synthesis of data flow intensive specifications, while limiting parallelism exploitation for control flow intensive applications. Indeed, the CDFG is based on the Control Flow Graph (CFG). The CFG contains some control flow edges, which represent neither data nor control dependencies. These edges are superfluous for scheduling purposes and may cause useless sequencing. As an example, consider the code fragment in Figure 2.3, containing three loops: $loop_1$, $loop_2$ nested into $loop_1$, and $loop_3$. The loops $loop_1$ and $loop_2$ are independent from $loop_3$, since neither data nor control dependencies occur among the instructions in the corresponding bodies. Figure 2.4a shows the associate CFG, which highlights how such independent loops are sequentialized through the edge between instructions 2 and 11. Figure 2.4b shows the associated DDG. The CDFG, built from DDG

```
        void motiv(int a, int b, int c, int * out){
            int i, j, n, t, k, z, res;
 1:         i = 0;
 2:         while ( i<a ){         // loop1
 3:             j = 0;
 4:             n = i + 1;
 5:             while ( j<n ){     // loop2
 6:                 t = j + 1;
 7:                 c = a + t;
 8:                 i = i + c;
 9:                 j = j +1;
                }
10:             i = i + 1;
            }
11:         k = 0;
12:         while (k < 10){        // loop3
13:             if (a > k)
14:                 b = k + a;
15:             k = k + 1;
            }
16:         z = a * b;
17:         res = z + c;
18:         *out = res;
```

Figure 2.3: Motivating example C code.



(a) CFG (b) DDG

Figure 2.4: CFG (a), and DDG (b), for the specification in Figure 2.3.

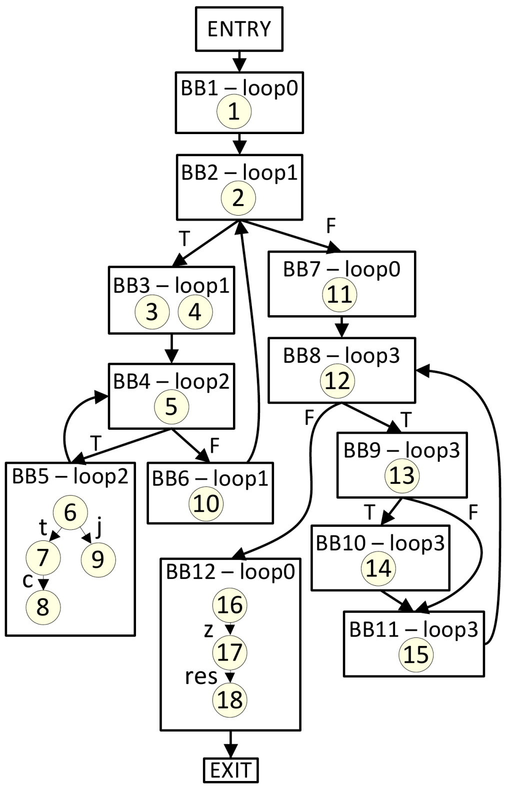


Figure 2.5: CDFG for the specification in Figure 2.3.

and CFG, is shown in Figure 2.5. Alternatives to the CDFG have been proposed, such as the Program Dependence Graph (PDG) [62]. Such graph contains the minimum data and control dependencies, thus not causing potential unnecessary sequencing. The PDG for the specification in Figure 2.3 is shown in Figure 2.6, where red dotted edges represent data dependencies, while black solid ones represent control dependencies. Unfortunately, the PDG alone does not generally contain all the information needed to build the controller. Indeed, since it does not contain control flow information at all, it lacks of the subset of control flow edges that ensure correct execution. This information, associated to control constructs, is needed to create correct designs. For example, consider the instruction 16 in the PDG in Figure 2.6. It must be activated after instruction 14 due to a data dependence. However, since instruction 14 belongs to a loop, instruction 16 must wait for the termination of $loop_3$ to guarantee correct execution. Such information, represented by the edge from 12 and 16 in the CFG, is instead missing in the PDG, which thus allows the instruction 16 to read the wrong value of the variable b. As another example,

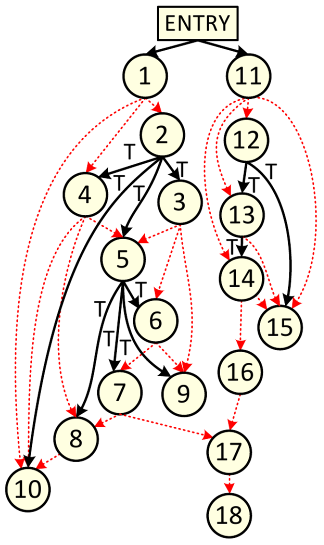


Figure 2.6: PDG for the specification in Figure 4.1.

consider instruction 12 in Figure 2.3, which is data dependent on instruction 11, hence it can be executed after instruction 11 has been executed. However, once the first iteration of $loop_3$ is terminated, instruction 12 must be reactivated. Again, this information, that is represented in the CFG through the edge between 15 and 12 is not present in the PDG.

2.1.2 Scheduling methodologies

In the context of high-level synthesis, the scheduling task [103] has the role to determine a propagation delay for every operation of the input behavioral description and then to assign each operation to a specific control step, which is usually equivalent to a single state of a centralized FSM. The List Scheduling (LS) [65] algorithm represents the most common approach for high-level synthesis of traditional hardware cores. The list scheduling orders the instructions in the specification by statically assigning, at design time, some priorities to them. Many strategies have been proposed to assign priorities to the instructions, such as Highest Level First algorithm (HLF), Longest Path (LP) algorithm, longest processing time, critical path method, or Heterogeneous Earliest Finish Time (HEFT). Once a priority has been assigned to every instruction, for every clock cycle, the algorithm (i) selects one instruction from the ordered list, (ii) it identifies a resource among those able to execute this instruction, and (iii) if no resource can be found, then it selects the next instruction in the list. These three steps are executed iteratively until a valid schedule is obtained. The algorithm attempts to minimize the total execution time by using a local priority function to defer operations when resource conflicts occur. The list scheduling algorithm, together with ASAP (As Soon As Possible) and ALAP (As Late As Possible), belongs to the category of the *constructive scheduling* techniques. In such approaches one operation is assigned to one control step at a time and this process is iterated from control step to control step. Alternatively, *global scheduling* techniques represent another category of scheduling algorithms for HLS. In this case, all control step and all operations are considered simultaneously when operations are assigned to control steps. As an example of global scheduling, the work in [119] presented the Force-Directed Scheduling (FDS) technique. Such algorithm attempts to minimize the resources required to meet a specified global time constraint. The main strength of this algorithm is the use of a global measure of concurrency to guide the scheduling. The work [120] represents an evolution such approach. This work introduced the Force-Directed List Scheduling (FDLS) algorithm, which uses a global measure of concurrency throughout the scheduling process, while being compatible with resources constraints. Other examples of global scheduling approaches are the *neural net scheduling* [111] and the *Integer Linear Programming* algorithms [85]. Finally, *transformational scheduling* algorithms start from an initial schedule, to obtain a final schedule by successive transformations. The main issue common to all these scheduling techniques is that they are static approaches, which completely specify the schedule at design time, without offering support for run-time adaptation. They assume that all the operations have fixed or bounded delays, expressed in numbers of cycles of a single one-phase clock. Consequently, the resulting system incurs in the previously discussed limitations when an unexpected event occurs at run-time.

The work in [88] introduced the relative scheduling technique for the management of unbounded operations, targeting centralized FSM based architectures. The authors define vertices with unbounded delays, which are associated to a subset of vertices called anchors. The anchors are used as reference points to compute the starting time of the operations. Then, they extend the scheduling problem in presence of unbounded operations by introducing the concept of offset with respect to anchors in the problem formulation. The delays are modeled as weights on the constraints graph of the application. Finding the offsets corresponds to finding the scheduling for the specification. The offsets are computed at run-time, since the delays of the anchors are unknown at compile time. Such approach efficiently deals with unbounded modules. However, it requires the definition of a partial ordering relation among the instruction, which restricts the available parallelism. Moreover, it does not cope with inherent uncertainty of programs.

On the other side, some works have been proposed to support both cyclic and acyclic dynamic scheduling, thus making the resulting architecture able to exploit inherent uncertainty due to uncertain parallelism level. Such works are based on the concept that *"there is an inadequate Instruction-Level Parallelism*

(ILP) between the operations in a single basic block, while higher levels of parallelism can only result from exploiting the ILP across basic blocks" [125]. The basic block barrier is induced by the unknown information at design time associated to branch constructs. Global acyclic scheduling techniques, such as trace scheduling [63] [116] and superblock scheduling [138] exploit this idea by moving the operations from their original basic block to preceding or succeeding basic blocks. Among the examples of *cyclic scheduling*, the "unroll-before-scheduling" technique consists in unrolling the loop a certain number of times and then applying a global acyclic scheduling algorithm to the unrolled loop body [63] [116] [138]. In this way, it is possible to obtain overlapping between the iterations in the unrolled loop body, while maintaining a scheduling barrier at the back edge. This limitation can be reduced by increasing the extent of the unrolling. However, this comes at the cost of increased code size and scheduling effort.

Software pipelining [47] refers to a class of global cyclic scheduling algorithms which reorders the operations by overlapping the execution of subsequent loop iterations. Such technique does not impose a barrier at the back edge. Software pipelining aims at removing the dependences among different iterations of a loop, so that seemingly sequential instructions may be executed in parallel. down and around the back edge N times, the first N iterations of the loop should have been peeled off. Similar rules apply if an operation is moved speculatively up and around the back edge to a previous iteration. Although such code motion can yield improvements in the schedule, it is not always clear which operations should be moved around the back edge, in which direction and how many times to get the best results, thus avoiding to significantly increase register pressure. the beginning of the repetitive portion. Recognition of this situation requires that one maintain the state of the scheduling process, which includes at least the following information: knowledge of how many iterations are in execution and, for each one, which operations have been scheduled, when their results will be available, what machine resources have been committed to their execution into the future and are, hence, unavailable, and which register has been allocated to each result. All of this has to be identical if one is to be able to branch back to a previously generated portion of the schedule. Computing, recording and comparing this state presents certain engineering challenges that have not yet been addressed by a serious implementation although the Petri net approach [124] [136] *Modulo scheduling* [126] represents another well-known scheduling technique. It specifies a set of constraints that must be met in order to achieve a legal modulo schedule. The objective of modulo scheduling is to engineer a schedule for one iteration of the loop such that when this same schedule is repeated at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or distinct iterations. This constant interval between the start of successive iterations is called initiation interval. In contrast to unrolling approaches, the code expansion is quite limited. In fact, with the appropriate hardware support, there is no need of code expansion whatsoever [127]. Once the modulo schedule has been created, all the implied code motions and the complete structure of the code, including the placement and the target of the loop-closing branch, can be determined. Despite modulo scheduling is at today the most qualified option for instructions scheduling in HLS, it does not completely manages uncertain information. Indeed it only copes with inherent uncertainty of applications (e.g., conditional instructions, incoming arguments of a function), while it is not able to manage other sources of uncertainty (e.g., modules with variable latency).

Moreover, Kountouris at al. [86] proposed the Hierarchical Conditional Dependence Graph(HCDG)-based list scheduling technique. Such approach defines the HCDG as a proper intermediate representation able to show conditional behaviors of the specification. Then it applies a list scheduling on such graph. Since every node of the HCDG is a basic block, as in the CDFG, such approach restricts fine-grained parallelism exploitation.

Finally, the System of Difference Constraint (SDC)-based scheduling [79] transforms a set of scheduling constraints in a systems of difference constraints, i.e., a set of constraint in the form $x - y \leq b$, where x and y are integer variables and b is a constant. The authors provide a mechanism for the conversion of many scheduling constraints into system of differences constrains. The performance objective is expressed as a linear function. In this way, the global optimization consists in solving a linear programming problem. However, the SDC-based scheduling operates over the CDFG, thus incurring in the previously discussed limitations related to this kind of intermediate representation.

2.1.3 Adaptivity

The main goal of this thesis work is to propose a dynamic scheduling methodology for automating the design of adaptive hardware cores. The proposed methodology is based on the explicit activation of the instructions at run-time according to some information computed at design time. For this reason, this Section discusses some methodologies which have been proposed in the literature to manage activation mechanisms for the instructions, and which can be potentially extended to support adaptivity in the embedded systems domain. The notion of Activating Conditions (ACs) has been introduced in [96] for the generation of parallel source code through parallelizing compilers. This work provides a methodology for coarse grained task scheduling on multiprocessor architectures. Such technique seems to fit well the design of hardware cores with support for uncertain and unpredictable information. However, it should be extensively revised to provide correct and efficient designs, when employed for fine-grained instruction scheduling of flexible embedded systems. More in detail, such technique aims at exploiting parallelism, by means of the Hierarchical Task Graph (HTG) [122] [71], which is used as IR for auto-scheduling. The HTG, at each hierarchical level, is a Task Graph enhanced with control and data dependencies, thus encompassing parallelism at all levels. The ACs for each task node in the HTG are computed according to its incoming data and control dependencies. Independent task nodes at the same hierarchical level can be simultaneously executed. Thus, while this approach well fits software-based auto-scheduling techniques, the use of the HTG as IR restricts parallelism exploitation when implementing hardware-based auto-scheduling, where it is possible to take advantage of a finer parallelism. As an example, consider the code fragment in Figure 4.3a. As shown in the corresponding Data Dependence Graph (DDG) in Figure 4.3d, instruction 5 is data dependent on instruction 2. Since instructions from 3 to 6 belong to a loop, they will be represented as a single composite node $3'$ in the corresponding HTG, shown in Figure 4.3c. Moreover, in the HTG the execution of the node $3'$ will start after instruction 2 has been executed, due to the data dependence between 2 and 5. This means that, despite the instructions 3, 4 and 6 are totally independent from instruction 2, their execution will be prevented until instruction 2 will be executed. The more loop bodies are large, the more this problem affects the performance. A different IR can overcome such problem. Furthermore, the formalization provided in [96] for the notion of Activating Conditions must be extended and revised when targeting hardware-based auto-scheduling and when using IRs different from the HTG, as will be deepened in Chapter 4.

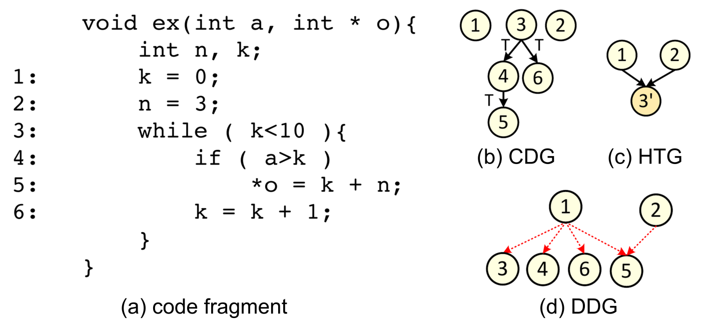

```
        void ex(int a, int * o){
            int n, k;
1:          k = 0;
2:          n = 3;
3:          while ( k<10 ){
4:              if ( a>k )
5:                  *o = k + n;
6:              k = k + 1;
            }
        }
```

Figure 2.7: Example code (a), and corresponding CDG (b), HTG (c) and DDG (d).

2.1.4 Architectures for HLS cores

Centralized FSMs represent the predominant model for designing controllers of hardware cores. However, they present serious difficulties in interacting with the environment and in adjusting their behavior according to external conditions. Indeed, instruction scheduling, resource allocation and resource binding are established and fixed at design-time. As a consequence, every time an unpredictable event occurs at run-time, the system either fail or underperform. For example, let us consider the case of the instructions scheduling. The designer, or the HLS tool, statically assigns a control step to every instruction. However, the latency of an operation can vary dynamically and unpredictably, due for example to process variation, to component degradation, or to interaction with an external module. On one side, if the latency of an operation decreases, the operations which depend on it can potentially execute sooner than expected. However, traditional hardware cores would provide suboptimal performance, since they would not be able to adapt the scheduling accordingly. On the other side, if the latency of an operation increases, then the operations which depend on it must be delayed to ensure correct execution. Thus, in this case conventional hardware cores would fail.

A partial solution to overcome the limitations related to standard hardware cores architectures consists in explicitly defining the different possible behaviors of the system directly during the creation of the FSM, as in [67]. However, this approach requires to know all the possible behaviors of the system at design time. Unfortunately, this may not be always possible, as in the case of component degradation or unpredictable latency of an external module. Moreover, this approach exponentially increases the area of the controller, and consequently the area of the resulting circuit. As an alternative to the centralized FSM, several architectural solutions have been proposed in literature, such as distributed FSMs [134], parallel FSMs [82] and hierarchical FSMs [81]. Despite such models have shown higher flexibility with respect to the centralized one, they still present several limitations. In the following such alternative architectural models for hardware cores controllers will be deepened.

Distributed Controllers Distributed controllers are obtained through FSM decomposition. Such technique is a top-down approach which starts from creating a centralized finite state machine. Then, the centralized FSM is divided into sub-machines [114, 134]. Such architectural model was introduced for reducing the delay of the critical path [49]. Distributed controllers allow simultaneous execution of statically identified parallel code portions, thus providing support for increasing the performance of the system, according to its parallelism level, which is potentially unknown at design-time. However, it results often impossible to identify non-overlapping decomposition. Hence, in such cases, portions of circuit must be duplicated, leading to an increase in the overall area, not only for the controller, but also from the datapath side. Moreover, this model does not support the interaction with components having variable latency, and it does not support unpredictable behaviors of the system. It only partially deals with inherent uncertainty of the application.

Parallel and Hierarchical FSMs A parallel FSM is a controller structure composed of communicating sub-controllers. Such architectures are event-driven. Sub-controllers communicate through message passing, handled by proper communication and synchronization protocols. Parallel controllers have been often implemented adopting a specification description capable to express the parallelism in the application, such as Petri nets [87]. A formal controller decomposition methodology for a Petri net specification was presented in [82], explicitly implementing VHDL simulation cycle implications into a Petri net. Finally a hierarchical PN-based approach was proposed in [81]. The main advantage in using PNs is that they allow easy specification of cooperating subsystems, and the use of formal validation methods. The model represents a token-based architecture, implementing the communication into the controller structure. However, it requires priority and synchronization schemes to share variables and to implement sub-controllers communication, thus increasing the design costs. Being event-driven, these machines are able to deal with unpredictable behaviors, due, for example, to process variation. The support for process variation is often based on statistical models, determining the best combination of resources and scheduling decisions to improve yield and total execution time [139]. However, statistical analysis does not guarantee execution correctness in all situations; this can lead to errors that sometimes cannot be tolerated (e.g., components in critical systems). One of the techniques is to replace common flip-flops with specific latches [48] to limit execution errors. Another technique is to couple critical components with sensors and analyze the status to provide information to the controller. This information can be then exploited to create adaptive systems able to automatically support such variation at run-time.

Controller Synthesis Techniques for Adaptivity Support The work in in [34] explores the synthesis of controllers directly from the behavioral specification. However, this work does not consider run-time adaptivity. On the contrary, the support for operations with variable latency was proposed in [56] by means of request and acknowledge signals, despite limitations in exploiting the inherent parallelism (inherent uncertainty in general). The work in [68] recently proposed a library of components for adaptive computing systems that can support synchronization between them. However, it is not described how to synthesize the controller and the communication protocol requires different signals to be exchanged. The work in [33] proposed a methodology to identify multiple modes of execution for the behavioral specification, merging them in the final architecture. This requires to design such implementations and an architectural support to switch from one configuration to the other. At a higher level of abstraction, Moreira [104] proposed an approach to dynamically identify the enabling conditions for the execution of concurrent tasks.

2.1.5 HLS Tools

At today, there exist many HLS tools, coming from both academic and industrial world. Among the examples, Xilinx's AccelDSP [1] accepts as inputs untimed Matlab descriptions. This tool allows several optimizations, such as loop unrolling, pipelining and memory mapping. However, it only works on streaming functions, thus significantly reducing the application domain. As another example, Agility Compiler [2] is a tool by Agility Design Solutions (ADS), acquired by Mentor Graphics in 2009. Agility Compiler takes as input SystemC specifications, providing automatic code generation for Actel, Altera and Xilinx FPGAs. However, it requires the designer to manually write hardware processes, to define sensitivity lists, and so on. AutoPilot [3] was developed by AutoESL, which has recently been acquired by Xilinx. It accepts as input C, C++ and SystemC specifications, with only minimal modifications. Xilinx has recently released the new version for this tool, which is called Vivado HLS [141]. BlueSpec [27] is another well known HLS tool. BlueSpec works at a lower abstraction level with respect to other tools. Indeed, it accepts as input BlueSpec System Verilog (BSV) applications. BSV is a Verilog based language in which design modules are implemented as a set of rules. For this reason, the designer cannot reuse pre-existing code, since the algorithm must be fully re-implemented in BSV. Catapult C [7], owned by Calypto Design Systems (acquired from Mentor Graphics in August 2011), accepts a large subset of C, C++ and SystemC. Memory accesses are not optimized by the tool, array elements that are reused in subsequent iterations are fetched from memory on every use. To enable local buffering, the designer must manually modify the source code. Compaan [8] is a relatively new HLS tool. It is designed for Xilinx FPGAs and works together with the Xilinx toolchain. Compaan is not a full HLS tool, since it only generates the communication infrastructure for C or Matlab applications, but not the processing elements themselves. It converts streaming algorithms in Kahn Process Networks (KPN), which can be easily mapped on parallel heterogeneous multicore platforms. C-to-Silicon (CtoS) [5] is a recent HLS tool from Cadence, taking as input SystemC applications. It includes a SystemC wrapper for designs written in C, C++ or TLM 1.0. However, unfortunately, this wrappers uses SystemC extensions that do not comply with the OSCI SystemC standard. The Impulse CoDeveloper [14] tool generates a RTL design starting from C. It supports a wide variety of FPGA platforms, such as Altera, Nallatech, Pico and Xilinx. It can be used either to generate modules, that can be combined with other IP to build a complete systems, and to create hardware accelerators, that can be connected an embedded FPGA processor. The main drawback of this tool is that it only supports data flow applications. ROCCC [22] is an open source HLS tool from the University of California at Riverside. The tool uses a very restrictive subset of C as its design language. This often complicates the design of the desired behavior. For example, if-then-else constructs can only be used to assign two possible values to the same variable. There is also a rather artificial distinction between modules that are sub-blocks of a design, and systems, representing the top level of a design. Synphony C Compiler [23] is Synopsys' HLS tool, which generates RTL code starting from wide range of C and C++ constructs. The architecture of the generated design is a Pipeline of Processing Arrays (PPA) that may contain individually compiled Tightly Coupled Accelerator Blocks (TCAB). Furthermore, Legup is an open-source HLS framework [12]. Despite the advantage of providing access to the source code, this tool provides lower quality of the results with respect to commercial tools.

Finally, the PandA framework, which is used in this work to evaluate the HLS results, is an open-source framework (see Figure 2.8) covering different aspects of the hardware/software co-design of embedded systems, including methodologies to support:

- high-level synthesis of hardware systems
- parallelism extraction for software and hardware/software co-design
- definition of metrics for the analysis and mapping of sequential code into multiprocessor architectures
- design flow for dynamic reconfigurable systems.

PandA uses the GCC compiler [11] as front-end. The representation taken as input for the construction of the EPDG is the *GIMPLE* produced by gcc. As a consequence, the dependencies among the instructions are those computed by gcc, including dependencies due to memory accesses [112]. The designs produced by the high-level synthesis can be simulated through a series of simulators, such as Modelsim [16], Icarus Verilog [13], or Xilinx ISim [28], and then synthesized on hardware, targeting both Xilinx [29] or Altera [10] technologies.

2.2 Design Methodologies of Modern HPC Systems

Parallel computing is emerging as an important trend in modern computing systems design. Despite parallel computing has been long used to solve large-scale complex problems, at today, efficiently implementing parallel applications remain an ongoing issue. Several programming models have been proposed to address this problem. Among those models, the most frequently used

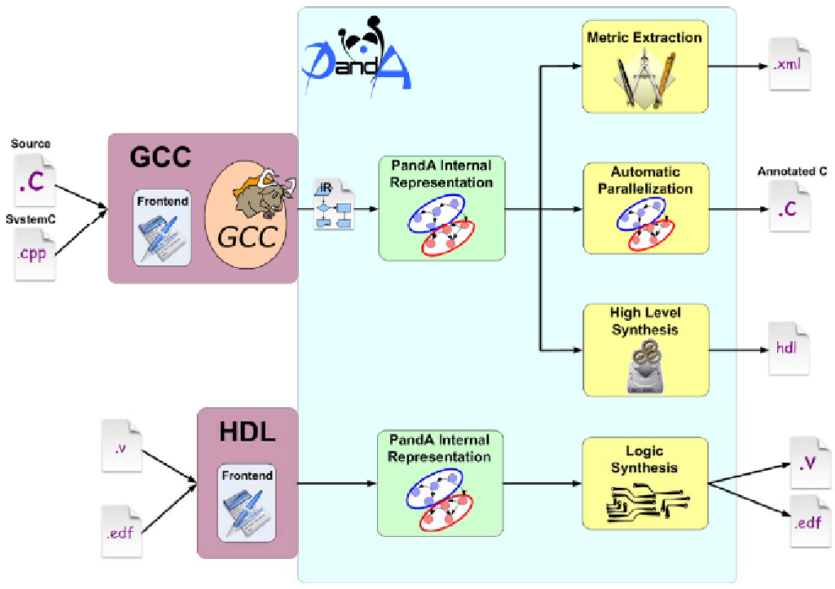


Figure 2.8: Panda framework schematic overview.

are OpenMP [26] [46] for shared memory programming, MPI [24] [17] for distributed memory programming, and Partitioned Global Address Space (PGAS) for distributed memory systems with shared memory abstraction [20]. This Section describes some traditional techniques and approaches related to the design of modern high-performance computing systems. Unknown and unpredictable events are usually managed by run-time systems in the HPC domain. However, such systems can significantly benefit from the support of a parallelizing compilation framework. The main goal of this thesis work in the HPC domain is to provide compiler support to the run-time system for managing unknown, uncertain and unpredictable information. More in detail, the proposed approach aims at defining compiler transformations and optimizations for efficient generation of parallel code for irregular applications. It is worth notice that the quality of the results provided by the compiler is significantly affected by the choice of the parallel programming model, which usually depends in turn on the class of applications. For these reasons, this Section mainly focuses on the following topics:

- *Data Models for Data-Intensive Applications*: Section 2.2.1 overviews the most common data models in HPC, highlighting which can represent a good fit for data-intensive applications
- *Parallel Programming Models*: Section 2.2.2 discusses the most common parallel programming models. More in detail, it describes OpenMP,

usually employed for shared memory systems, Message Passing Interface (MPI), usually adopted in systems with distributed memory, and Partitioned Global Address Space (PGAS), which is used for distributed memory systems which employ a shared memory abstraction

- *Automatic Parallel Code Generation*: Section 2.2.3 summarizes which aspects parallelizing compilers should take into account while attempting to generate efficient parallel code

- *Parallelizing Irregular Applications*: Section 2.2.4 highlights the main issues related to the parallelization of this challenging class of applications, together with the main techniques proposed in the literature to optimize irregular applications on cluster of supercomputers

2.2.1 Data Models for Data-Intensive Applications

Data-intensive programs represent the class of applications which best take advantage of the potentiality offered by supercomputers. Indeed, a main area of this discipline is developing parallel processing algorithms and software programs that can be divided into little pieces so that each piece can be executed simultaneously by separate processors. HPC systems are physically (i.e., at the hardware level) and logically (i.e., at the software level) partitioned into functional units, as shown in Figure 2.9. The partitions cooperate to appear

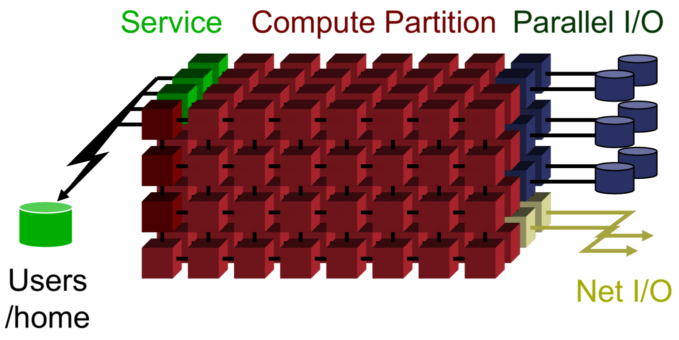


Figure 2.9: Partition Model in HPC Systems

as a single system to the end user. Such systems usually rely on a mixture of programming models. The most common approach exploits message Passing Interface (MPI) to communicate among the nodes, while OpenMP is used at the node level, which is usually a shared memory system, to manage parallelism among the cores. The nodes in the system can be interconnected according to different topologies. Mesh/torus topologies are upgradeable, and they usually show to scale better with the number of nodes. The strategy for the design of parallel applications on such systems changes according to the memory model adopted, which can be shared or distributed. Computer architectures whose memories are physically distributed among the nodes can be in turn modeled as Distributed Shared Memory (DSM) systems, where the nodes see a unique logic shared memory, or distributed memory systems, which are naturally programmed with message passing. Regardless of the logic abstraction used for the physically distributed memory (shared or distributed), these systems are characterized by non uniform access time to the memory (i.e., Non Uniform Memory Access systems, or NUMA). In shared memory systems (i.e., Uniform Memory Access systems, or UMA) all the processes share a unique memory address space, as shown in Figure 2.10. Thus, shared data

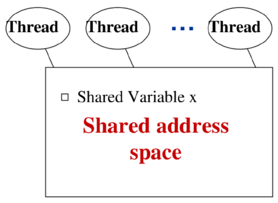


Figure 2.10: The Shared Memory Model.

manipulation requires synchronization among the threads. This approach does not allow locality exploitation and it does not scale neither with the number of processes nor with the number of nodes. Shared memory HPC systems are usually programmed with OpenMP, which abstract all the communication operations. The model works well for architectures with few cores, but does not provide adequate scalability for modern HPC systems, which rely on a multitude of cores, tightly interconnected to private, non-coherent cache memories to exploit locality. Moreover, OpenMP requires detailed knowledge about the underlying architecture to properly set the many technological parameter provided. For these reasons, distributed memory systems are usually employed when operating on massive amounts of data. Non-Uniform Memory Access (NUMA) systems are characterized by variable access time to memory locations, depending on the position of the requested memory portion with respect to the processor which makes the request, either if the memory is logically shared or not. In other words, every processor can access quickly to its local memory, while the latency to remote memory accesses will be higher (i.e., to access other memories, which are local to other processors). Indeed, such systems are characterized by a set of nodes distributed among the architecture, as shown in Figure 2.11. Non-uniform architectures which do not rely on a shared memory abstraction are naturally programmed with message passing programming models. The Message Passing Interface (MPI), is the de-facto standard for message passing for high performance systems. MPI provides routines to move the data from the address space of one process to that of another process through cooperative operations on each process (send/receive routines).

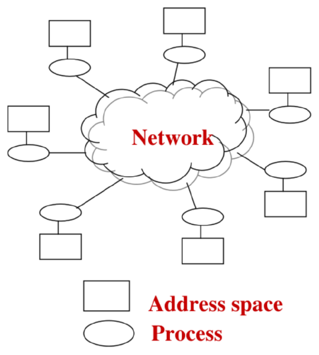


Figure 2.11: The Message Passing Interface (MPI) Model.

Barriers are used to synchronize the threads. Usually a Single Program, Multiple Data (SPMD) control model is coupled with the use of MPI. The tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster. SPMD is the most common style of parallel programming. The main issue in the design of applications for HPC systems resides in the need to restructure the code to exploit the parallelism provided by parallel architectures. Writing parallel code for these machines is not trivial and either it requires detailed knowledge about the underlying architecture, as for OpenMP, or it stands in need to provide a significant programming effort, as for MPI, where the communication must be managed explicitly by the programmer. Moreover, MPI suffers of significant communication overhead for small transactions, thus requiring aggregation of messages. Data aggregation often causes excessive buffering. On the other side, all the processors of

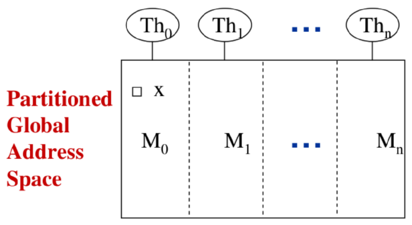


Figure 2.12: The Distributed Shared Memory Model.

distributed memory systems with shared memory abstraction share the logic memory uniformly, which is physically partitioned among the processes, as shown in Figure 2.12. The access time to a memory location is dependent of which processor makes the request or which memory chip contains the transferred data. For regular applications it is possible to partition the memory in such a way that processor M_0 has affinity with thread Th_0 in terms of data, in order to exploit locality. The memory address space resulting from the logical partitioning of the memory is known in the literature as Partitioned Global Address Space (PGAS).

2.2.2 Parallel Programming Models

It is an increasingly common belief that the programmability of parallel machines is lacking, and that the High-End Computing (HEC) community is suffering as a result of it. The population of users who can effectively program parallel machines comprises only a small fraction of those who can effectively program traditional sequential computers, and this gap seems only to be widening as time passes. Current parallel programming languages are significantly different from those that a modern sequential programmer is accustomed to, and this makes parallel machines difficult to use and unattractive for many traditional programmers. To this end, developers of new parallel languages should ask what features from modern sequential languages they might effectively incorporate in their language design. For example, the parallel programming model must be able to express parallelism, data distribution, and typically synchronization and communication.

The value of a programming model is usually judged on its generality: how well a range of different problems can be expressed and how well they execute on a range of different architectures. The implementation of a programming model can take several forms such as libraries invoked from traditional sequential languages, language extensions, or complete new execution models. Generally, parallel programming models or frameworks are based on either language enhancement or run-time libraries. For this reason, the following paragraphs will deepen different parallel programming approaches together with the run-time libraries they are usually associated to.

across multiple frameworks with little modification. Separate chunks of source data that can be worked on concurrently by multiple threads or tasks are encapsulated within an object-oriented design representation. A good practice in single-threaded programming, this approach easily lends itself to parallel execution. Since the range of letters of the alphabet. For example, the first collect object may look at words starting with the letters 'a' through 'e', the second from 'f' through 'j', and so on.

OpenMP

OpenMP is a parallel programming model usually employed in shared memory architectures. A recent innovation is the use of run-time libraries for shared memory architectures. Intel offers the Threading Building Block (TBB) [15], while Microsoft offers the Task Parallel Library (TPL) [92] and Parallel Patterns Library (PPL) [84]. In June 2009 [94], the Oracle Solaris Studio product team presented a new parallel C++ run-time library called MCFX Framework (later renamed to Parallel Framework Project) at the International SuperComputing Conference in Hamburg, Germany. OpenMP [26] [46] is an industry standard API design specification developed through the joint efforts of top computer manufacturers and software developers such as Oracle, Hewlett-Packard, IBM, and Intel. It is supported on the most widely used native programming languages such as Fortran, C and C++. OpenMP offers a common specification that lets software programmers easily design new parallel applications or parallelize existing sequential applications to take advantage of multicore systems configured with shared memory [26]. Portability is an important characteristic of OpenMP. Any compiler that supports OpenMP can be used to compile parallel application source code that is developed using OpenMP. The resulting compiled binary should be run on the target hardware platform to achieve parallel performance. OpenMP constructs are expressed by pragmas, directives, and programming API calls in the source code, and allow programmers to specify parallel regions, synchronization, and data scope attributes. OpenMP also supports the use of run-time environment variables to specify the run-time configuration. For example, the environment variable OMP_NUM_THREADS specifies the number of working threads used at run-time. OpenMP 3.0 enhances OpenMP by adding the concept of task, i.e., work units that can be executed immediately or deferred until later. Tasks are composed of executable code, data environment and internal control variables. A task can be either tied to a thread, or executed by a group of working threads. A thread encountering a parallel directive creates as many tasks as threads in the group. Tasks can be associated with barriers, so that a thread that encounters a barrier is blocked until the set of associated tasks is completed. A thread blocked by a barrier may move onto other pending tasks. Barriers can be of two varieties, taskwait and taskgroup. A taskwait barrier blocks the encountering task until all child tasks are finished.

The biggest benefit of OpenMP is a relatively easy conversion process from sequential programs into a parallel OpenMP programs. The other big benefit is the ability to eliminate tedious thread creation, communication and synchronization details, which allows software developers to focus on core program logic instead of having to manage concurrent threads. However, because OpenMP is specified as a shared memory programming model, its parallel scalability is restricted by the number of processor cores (and thus, threads) available on the target machine. Scalability is also limited by the degree of complexity of the computational logic relative to the internal thread management overhead. OpenMP 3.0 supports the dynamic task queue construct to make the parallel programming model more flexible. Because OpenMP is language-based and every new construct requires work by the compilers, overall OpenMP functionality is not as rich as the run-time library based models. Finally, data race conditions are the most challenging problems for parallel software developers.

Message Passing Interface (MPI)

The Message Passing Interface (MPI) [24] [17] is an industry-standard API specification designed for high-performance computing on multiprocessor machines and clusters. The standard was designed by a broad group of computer vendors together with software developers, with a number of MPI implementations produced by different research institutes and companies. The most popular one is MPICH, which often is used to optimize MPI implementations for a specific platform or interconnect [24]. MPI offers a distributed memory programming model for parallel applications. Although the entire MPI API set contains more than 300 routines, many MPI applications can be programmed with less than a dozen basic routines. In MPICH design, the entire API set is implemented and built on top of a small core of low-level device interconnect routines. This design feature supports portability across different platforms - a developer only need re-work the core set of device interconnect routines to port or optimize MPICH for a new platform or interconnect. Currently, the proposed MPI implementations are still evolving. MPI-1 supports key features such as interconnect topology, and point-to-point and collective message communication with a communicator. A message can contain MPI data of primitive or user-defined (derived) data types with message data content in packed or unpacked format. MPI-2 provides many advanced communication features such as remote memory access and one-side communication. It also supports dynamic process creation and management, and parallel I/O. Given the current state of semiconductor technology and computer architecture, the dominant system performance factor is memory hierarchy rather than CPU clock rate. Obviously, an application runs faster if most of its data memory accesses fall within the cache range. Being a distributed memory programming model, MPI usually can achieve good linear scalability for large-scale applications. When an MPI application is partitioned to run on a large cluster of computing nodes, the memory space of each MPI process is reduced and the memory accesses could fall outside the high performance range of the memory hierarchy. This non-uniform memory performance effect can also apply to other programming models.

MPI offers a good solution for parallel application on computer clusters, but it can be a difficult programming model for many developers. Because MPI has higher communication cost, the program core logic must be properly partitioned to justify the distribution overhead. Analyzing, partitioning, and mapping an application problem to a set of distributed processes is not an intuitive task. Moreover, it can also lead to load unbalance, especially for particular classes of irregular applications. Because of the interactive complexity of many MPI processes, it is also quite challenging to debug and tune a MPI application running on a large number of nodes. The implementation quality of MPI routines can impose additional software development challenges.

MPI performance depends on the underlying hardware platform and interconnect. In some extreme cases, the MPI application behavior may be affected by heavy interconnect traffic. Another big issue is the reliability of large-scale MPI applications. Depending on the implementation, an MPI program could stop working whenever a single computing node fails to respond correctly.

Partitioned Global Address Space (PGAS)

Even though latency and contention problems in large-scale multiprocessors have resulted in a general move away from uniform shared memory toward distributed memory, the shared-memory programming model retains many attractions for users of these systems. In particular, the ability to read and write remote memory with simple assignment statements is considerably more attractive than having to learn all the conventions of a message-passing library, even if the latter is portable. At the same time, the quest for performance often makes it desirable to view program data as distributed among a number of local memories. The Partitioned Global Address Space (PGAS) programming model aims at exploiting the performance and data locality (partitioning) features of MPI, while not loosing the programmability and data referencing simplicity of a shared-memory (global address space) model. PGAS has been gaining rising attention due to its high productivity. It offers ease-of-use due to its global shared address space abstraction, whereas its locality awareness helps exploit higher performance. This can result in reduced development and execution times. The phrase partitioned global address space was first introduced by Katherine Yelick and Tarek El-Ghazawi during the IPDPS 2002 international conference. But the concepts of PGAS have predated its name. The first full day tutorial on distributed shared memory programming model (as it was called back then) was held in Supercomputing 2001 at Denver, CO and covered the concepts, semantics, and syntax of the Unified Parallel C (UPC), the Titanium (a Java dialect), and the Co-Array Fortran (CAF) parallel programming languages [140]. More recently, the DARPA HPCS program has produced new promising PGAS languages, such as the X10 and the Chapel. At today, PGAS languages are in a unique position to address the escalating programming complexities associated with supercomputing architectures. Among the several existing PGAS languages, the following will focus on the key points characterizing UPC, X10 and Chapel.

Unified Parallel C (UPC): UPC is a parallel extension of the C programming language intended for multiprocessors with a common global address space. A descendant of Split-C [53], AC [43], and PCP [39], UPC has two primary objectives: to provide efficient access to the underlying machine, and to establish a common syntax and semantics for explicitly parallel programming in C. Moreover, UPC tries to minimize the overhead involved in communication among cooperating threads. UPC's parallel features can be mapped either onto existing message-passing software or onto physically shared memory to make its programs portable from one parallel architecture to another. As a consequence, vendors who wish to implement an explicitly parallel C could use the syntax and semantics of UPC as a basis for a standard. One of UPC's advantages is that it enables programmers to exploit data locality in a variety of memory architectures. UPC assumes that the programmer must think about issues of memory locality in designing effective data structures and algorithms, by offering a reasonable means of expressing the result of the design effort.

One of the primary principles of UPC is that the presence of parallelism and remote accesses should not obscure the resulting program. Users are able to view the underlying machine model as a collection of threads operating in a common global address space. Details as whether the model is implemented as shared memory or as a collection of physically distributed memories are not visible to the programmer. Within that model the user makes decisions about data locality and memory consistency. It is up to the compiler and runtime to ensure that the programmer's declarations of shared and private data, and strict or relaxed consistency, are implemented correctly. The programmer's job is to understand the programming model and its relation to the algorithm or application. To create a memory model in which local data and remote data are differentiated only by the ways in which they are declared, UPC slightly modifies the C language by focusing on pointers and arrays, which are the two C constructs which are most closely tied to addresses. The addition of keywords gives the programmer the ability to distinguish between data that is strictly private to a given thread and data that is shared among all threads in the parallel program. Arrays can be declared to be shared among the threads in a variety of different ways; the result is a flexibility in data layout comparable to that of High-Performance Fortran (HPF) [128].

A UPC program running with shared data on a parallel system will contain at least a single thread per processor. Each thread has local data on which it can operate with all the efficiency of a traditional process on a sequential computer. At the same time, however, it has easy access to shared data that are local to other threads. Unless a user intervenes, there is no implicit synchronization among the threads of computation running in the system. User intervention in the memory model can happen in several ways. The first involves declarations that tell the compiler how much code motion is acceptable for a given variable; a second involves labels that specify strict or relaxed consistency for a particular statement or sequence of statements. In addition, the user can require synchronization at particular points by specifying barriers, i.e. global operations which consist of waiting for the completion of all operations issued before the barrier, followed by an operation which terminates when all the threads have completed these operations.

By keeping software constructs to a minimum and enabling programmers to use underlying hardware efficiently, UPC is solidly in the C tradition. Programmers can write efficient programs because they can choose how much run-time error checking and synchronization is necessary for their particular codes. At the same time, programmers need to understand the code they are writing. For example, writing a value to a remote memory location does not guarantee that another thread will get the new value when it next reads the variable unless the programmer tells the compiler that such a guarantee is necessary. The compiler is thus free to use the memory consistency mechanisms of the underlying hardware to achieve the best performance it can for the specified program. However, UPC produce efficient parallel programs only when the application shows regularity.

X10: X10 is a modern, statically typed, class-based Object-Oriented programming language, designed for high productivity programming of scalable applications on high-end machines. It introduces concurrency and distribution primitives. The basic move in the X10 programming model is to exploit locality through a notion of place, which hosts multiple data items and activities that operate on them. Aggregate objects (such as arrays) may be distributed across multiple places. New activities can be dynamically spawn in multiple places and then sequenced through a finish operation, which detects termination of activities. Atomicity is obtained through the use of atomic blocks. Activities may repeatedly detect quiescence of a data-dependent collection of (distributed) activities through a notion of clocks, which generalize the concept of barriers. Thus, X10 has a handful of orthogonal constructs for space, time, sequencing and atomicity. X10 smoothly combines the current dominant paradigms for shared memory computing and message passing.

The idea behind X10 is that there has always been considerable theoretical research in concurrency. For example, in imperative languages, C ILK [4] [35] has introduced some novel ideas such as work-stealing for symmetric multi-processors (SMPs). Titanium [143], Co-Array Fortran [113] and Unified Parallel C (UPC) [59] have introduced the Partitioned Global Address Space (PGAS) model [42] in JAVA, Fortran and C respectively, despite only in Single Program Multiple Data (SPMD) frameworks. However, the state of the art in concurrent high-performance computing continues to be *library-based* (e.g. OpenMP [26] for shared-memory concurrency and MPI [17] for message-passing) rather than *language-based.* Mainstream languages have been slow to adopt concurrency, and they still suffer from several problems. For example, JAVAT M [72] employes a single global heap, which does not scale with the number of processors. Complex memory models [21] are needed to enable efficient implementation on modern multi-processors. JAVA does not support multidimensional arrays, user-definable value types, relaxed exception model, aggregate operations, etc. [106] [105]. uniform shared memory is no longer appropriate for such machines. To overcome these limitation, the DARPA HPCS program designed an explicitly parallel programming language for clustered computing, X10 [133]. The fundamental goal of X10 is to enable scalable, high-performance, high-productivity programming for high-end computers for traditional numerical computation workloads (such as weather simulation, molecular dynamics, particle transport problems, etc.) as well as commercial server workloads. X10 is explicitly parallel, since it relies on powerful compilers to automatically extract enough parallelism to keep hundreds of thousands of nodes busy. For productivity, X10 belongs to the familiar statically typed, class-based, object-oriented programming mould; X10 is intended to be readily accessible to programmers in JAVA-like languages. Thus X10 is intended to support in an integrated fashion the set of problems that are today addressed by libraries such as OpenMP and MPI bolted onto base programming languages such as Fortran or C. The language has been implemented via a translator to JAVA, developed using the Polyglot compiler framework. Local vs remote memory latency and bandwidth ratios for large scale-out machines are often higher than two, perhaps three orders of mangitude. The X10 approach to this dilemma is to introduce the notion of a *place*. A place consists of a collection of data and activities that operate on the data. A computation may consist of millions of places. A programmer may think of a place as an MPI task or a node in a distributed Java Virtual Machine (JVM) with its own heap and collection of threads. An asynchronous activity is created by a statement $async(p)s$ where p is a place expression and s is a statement. Such a statement is executed by spawning an activity at the place designated by p to execute statement s. An activity is created in a place and remains resident at that place for its lifetime. Both s and p may access lexically scoped final variables. Each activity has a sequentially consistent view of the data at that place and may operate only on the data at the place. It may reference data at other places, but must operate on them only by launching asynchronous activities (at the place where the data lives). Thus, X10 supports a Globally Asynchronous, Locally Synchronous (GALS) computation model, familiar from hardware design and embedded systems research. Unlike other PGAS languages, X10 is not SPMD. In other words, different (collections of) activities may run at each place. Any activity may use the place expression here to reference the current place. Places are assumed to be totally ordered; if p is a place expression, then $p.next$ is a place expression denoting the next place in the order. There are no expressions for creating a new place, rather each computation is initiated with a fixed number of places. Each object carries its location through a final field location. Access to non-final fields is permitted only for objects at the same place. Any attempt to access remote mutable data results in a BadPlaceException (BPE). Since X10 supports fine-grained asynchronous, parallel activities, a reliable mechanism is needed to detect termination. X10 provides a $finish$ construct for such purpose. Intuitively, $finish\ S$ executes S and suspends until all activities created while doing so have terminated (normally or abruptly). JAVA-like languages support a notion of monitors, i.e. the programmer must write code that explicitly obtains and releases locks [72]. However, locks are a very low-level and error-prone synchronization mechanism, making it very easy for programmers to write erroneous code that underlocks (causing race conditions) or over-locks (causing deadlock). For this reason, X10 supports atomic blocks [77] [76] [78]. An X10 computation consists of a large number of asynchronous activities scattered across space. The notion of time in X10 consists in a sequence of phases. In each phase, activities (scattered across multiple places) read and write shared data (e.g. a distributed array). Once all activities have performed one phase of their calculations, each is informed of this global quiescence and computation moves to the next phase, and the process repeats. For instance, in a molecular dynamics application, it may be necessary for a controller activity to determine that each molecule has computed the force incident on it from all other molecules, and hence its instantaneous acceleration a. The controller may then advance simulation time, causing each molecule to determine its new position p and velocity v (as a function of its mass m, a and old p and v). In SPMD languages this phasing is accomplished using the notion of a (split-phase) barrier. For instance, UPC provides a single barrier for all threads in a computation, accessed through $upc\,notify$ (signal that this thread has reached the barrier) and $upc\,wait$ (wait until all threads have reached this barrier). X10 clocks can be thought of as obtained from split-phase barriers while: i) permitting dynamic creation, ii) permitting dynamic (de-)registration of activities, and iii) ensuring that operations are race-free (hence determinate). Concretely, a clock is a data-structure that may be dynamically created; an activity may create as many clocks as it wishes. Conceptually each clock is associated with an integer that specifies the current phase of the clock; this integer is initially zero, and is incremented each time the clock advances. A clock is said to advance to the next phase when all activities registered with it have quiesced.

Chapel: Chapel is a global-view parallel language that supports a block-imperative programming style. Syntactically, Chapel diverges from other block-imperative languages like C and Fortran, mainly because it has different goals (e.g., support for generic programming and a better separation of algorithm and data structures). Chapel defines the term *locale* as the units of a parallel architecture which have uniform access to the machine's memory. For example, on a cluster architecture, each node and its associated local memory would be considered a *locale*. Chapel supports a locale type and provides every program with a built-in array of locales to represent the portion of the machine on which the program is executing. Locales are used for specifying the mapping of Chapel data and computation to the physical machine. The parallelism in Chapel is not described using a processor- or task-based model, but in terms of independent computations implemented using threads. Rather than giving the programmer access to threads via low-level fork/join mechanisms and naming, Chapel provides higher-level abstractions for parallelism using anonymous threads that are implemented by the compiler and runtime system, thus providing generality while relieving the users of the burden of thread management. Chapel supports data parallelism using a language construct known as a domain that is used to define the size and shape of arrays and to support parallel iteration. The term array in Chapel represents a general mapping from an index set of arbitrary type to a set of variables. Chapel supports task parallelism using stylized forms of specifying parallel statements and synchronizing between them (e.g., cobegin, forall). indexes have no values or algebra in relation to one another, i.e. they are simply anonymous indexes, each of which is unique. Opaque domains can be used to represent graphs or link-based data structures in which there is no requirement to impose an index value or ordering on the nodes that comprise the data structure. Chapel supports an index type that is parameterized by a domain value. A variable of a given index type may store any of the indexes that its domain A domain's indexes may be decomposed between multiple locales, resulting in a distributed allocation for each array defined using that domain. Domain distributions also provide a default work location for threads implementing parallel computation over a domain's indexes or an array's values. Since every array is constrained to its defining domain, two arrays declared using the same domain are guaranteed to have the same size, shape, and distribution throughout their lifetimes. This allows the compiler to reason about the distribution of aligned arrays. Even if Chapel supports a number of traditional distributions, a primary goal of the language is to allow users to implement their own distributions when they have application-specific knowledge for how an array should be distributed that is not captured by the standard distributions [58]. Chapel supports a structured form of producer/consumer semantics on arbitrary variables for their synchronization. It also supports atomic sections to indicate that a group of statements should be executed as though they occurred atomically from the point of view of any other thread. By specifying intent rather than mechanism, atomic sections result in a higher-level abstraction for users than locks. Chapel's goals include allowing the programmer to control where data is stored and where computation takes place on the machine. Domain distribution is one example of locality control. Moreover, users may initially wish to use a distribution from the standard library, and later write their own custom distribution once performance concerns become more critical. When iterating over a domain, each iteration will typically execute on the locale that owns that index, as specified by the domain's distribution. Users can override this behavior using an *on* clause, which references a locale and specifies that the given computation should take place there. *On* clauses can also be applied to variable declarations to cause variables to be allocated on a specific locale. Typically, variables that are not explicitly managed in this way are allocated in the locale on which the thread is executing. Two arrays in Chapel are assigned using a single mechanism regardless of whether or not they shared the same distribution. In such a model, the programmer needs to rely on feedback from the compiler or performance analysis tools to understand the difference in performance between the two cases. However, the advantage of this model is that the program's syntax keeps the execution model clear to the programmer and compiler. Chapel's designer have chosen to emphasize programmability over execution model transparency. Chapel has very general mechanisms for distributing data over processors. It is also significantly easier to program with respect to MPI. However, it infers the communication from the source code and the data distribution. On the other side, the programmer has complete control of the communication in MPI programs, being able to make it efficient. A PGAS compiler can hardly obtain alone the same efficiency in terms of communication.

2.2.3 Automatic Parallel Code Generation

Designing and developing parallel programs has historically been a manual process. The programmer is typically responsible for both identifying and actually implementing parallelism. However, very often, manually developing parallel codes is a time consuming, complex, error-prone and iterative process. For a number of years now, various tools have been made available to assist the programmer with converting serial programs into parallel programs (e.g., parallelizing compiler or pre-processor). A parallelizing compiler generally works in two different ways: fully automatic or programmer directed. In the first case, the compiler analyzes the source code and identifies opportunities for parallelism. The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance. Usually, loops are the most frequent target for automatic parallelization. In the second case, instead, the programmer explicitly tells the compiler how to parallelize the code, using "compiler directives" or possibly compiler flags. This approach can be sometimes used in conjunction with some degree of automatic parallelization. The most common compiler generated parallelization is done using on-node shared memory and threads (such as OpenMP). OpenMP is a programmer directed parallelization strategy, since it requires the programmer to instrument the sequential code with parallelization primitives (pragmas). At today, automatic parallelization still suffer from many issues, such as, for example, performance degradation, lower flexibility with respect to manual parallelization, the parallelization is often limited to a subset (mostly loops) of code, and finally automatic tools may actually not parallelize code if the compiler analysis suggests there are inhibitors or the code is too complex.

As discusses in [30], to overcome such limitations, research in parallelizing compiler should focus in providing improvements on the following aspects, which are already consolidated in manual parallelization:

- *Feasibility analysis*: the compiler should determine whether or not a given problem can actually be parallelized. It should analyze the degree of parallelism of the application, according to the desired target architecture. Moreover, it should identify those areas of the program (if any) which are disproportionately slow, or cause parallelizable work to halt or be deferred, as for example I/O. Sometimes it is possible to restructure the code, by applying transformations and optimizations, to reduce or eliminate unnecessary slow areas. Finally, this preliminary analysis should identify inhibitors to parallelism, such as data dependence.

- *Partitioning*: one of the first steps in designing a parallel program consists in dividing the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning. There are two basic ways to partition computational work among parallel tasks: *domain decomposition* and *functional decomposition*.

  - *Domain Decomposition*: the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.

  - *Functional Decomposition*: the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

  Functional decomposition lends itself well to problems that can be split into different tasks, such as ecosystem modeling, signal processing and climate modeling.

- *Communication*: the need for communications between tasks depends on the specific problem. Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. For example, consider an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work. These types of problems are called *embarrassingly parallel* because their parallelization is very straight-forward. Very little inter-task communication is required. On the other hand, most parallel applications are not quite so simple, and do require tasks to share data with each other. For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data. There are a number of important factors to consider when designing a program's inter-task communications:

– *Cost of communications*: inter-task communication virtually always implies overhead. Machine cycles and resources that could be used for computation are instead used to package and transmit data. Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work. Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

– *Latency vs. Bandwidth*: the latency is the time needed to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds. The bandwidth is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec. Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

– *Visibility of communications*: with the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer. With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The compiler may not even be able to know exactly how inter-task communications are being accomplished.

– *Synchronous vs. asynchronous communications*: synchronous communications require handshaking between tasks that are sharing data. This can be explicitly specified by the programmer, or it may happen at a lower level unknown to the programmer. Synchronous communications are often referred to as blocking communications since other work must wait until the communications have completed. Asynchronous communications allow tasks to transfer data independently from one another. Asynchronous communications are often referred to as non-blocking communications since other work can be done while the communications are taking place. Interleaving computation with communication is the single greatest benefit for using asynchronous communications. The compiler needs to perform different transformation and optimizations according to the type of communication established by the programmer or by the run-time system in use.

- *Synchronization*: there exist different types of synchronization constructs for parallel programs:
  - *Barrier*: usually implies that all tasks are involved. Each task performs its work until it reaches the barrier. It then stops, or “blocks”. When the last task reaches the barrier, all tasks are synchronized. Then, in some cases, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.
  - *Lock/semaphore*: can involve any number of tasks. Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock/semaphore/flag. The first task to acquire the lock “sets” it. This task can then safely (serially) access the protected data or code. The tasks which attempt to acquire the lock must wait until the task that owns the lock releases it. Such constructs can be blocking or non-blocking
  - *Synchronous communication operations*: involves only those tasks executing a communication operation. When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task, which agrees to receive the message.

  Managing synchronization is the most challenging task for a compiler during automatic parallelization.

- *Load Balancing*: load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time. Load balancing is important to parallel programs for guaranteeing good performance. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance. To achieve load balancing two main aspects must be considered:
  - *Equally partition the work each task receives*: for array/matrix operations where each task performs similar work, the compiler should evenly distribute the data set among the tasks. For loop iterations where the work done in each iteration is similar, the compiler should evenly distribute the iterations across the tasks.

– *Use dynamic work assignment*: certain classes of problems result in load imbalances even if data is evenly distributed among tasks. Among the examples, sparse arrays (some tasks will have actual data to work on while others have mostly "zeros"), adaptive grid methods (some tasks may need to refine their mesh while others don't), N-body simulations (some particles may migrate to/from their original task domain to another task's; the particles owned by some tasks may require more work than those owned by other tasks). When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a scheduler. As each task finishes its work, it queues to get a new piece of work. It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

2.2.4 Parallelizing Irregular Applications

Irregular programs are programs organized around pointer-based data structures such as unbalanced trees, unstructured grids, and graphs. Such applications result difficult to partition in a balanced way, they suffer of poor locality and they are characterized by high-intensive communication and synchronization. For all these reason, irregular applications represent, at today, the most challenging class of applications to efficiently parallelize on modern supercomputers. Recent investigations by the Galois project have shown that many irregular programs have a generalized form of data-parallelism called amorphous data-parallelism [89]. However, in many programs, amorphous data-parallelism cannot be uncovered using static techniques, and its exploitation requires runtime strategies such as optimistic parallel execution. The work in [121] describes the design and implementation of a tool called ParaMeter that produces parallelism profiles for irregular programs, aiming to estimate the amount of amorphous parallelism in irregular applications. Parallelism profiles are an abstract measure of the amount of amorphous data-parallelism at different points in the execution of an algorithm, independent of implementation-dependent details such as the number of cores, cache sizes, load-balancing, and so on. ParaMeter can also generate constrained parallelism profiles for a fixed number of cores. The work in [50] presents a single-chip multiprocessor that engages aggressive speculation techniques to enable dynamic parallelization of irregular, sequential binaries. Thread speculation and data value prediction are combined to enable the processor to execute dependent threads in parallel. The architecture performs a novel form of dynamic thread partitioning and includes an aggressive correlated value predictor. Microarchitectural structures manage inter-thread data and control dependencies. However, such approach targets shared memory architectures. The work in [73] introduces some automatic parallelization and optimization techniques for irregular scientific. These techniques include communication cost reduction for irregular loop partitioning, inter-procedural optimization techniques for communication preprocessing when the irregular code has the procedure call, global vs. local indirection arrays remapping methods, and OpenMP directive extension for irregular computing. However, also this work targets shared memory HPC systems.

2.3 Unknown, Uncertain and Unpredictable Information

Modern computing systems are usually subject to many sources of unknown or uncertain information, which often is also unpredictable. Unpredictability emerges in different ways in different aspects of the technology, during the design process. For example, nowadays, computer architectures are usually composed of a heterogeneous mixture of processing cores (e.g., FPGAs, GPGPUs, DSPs) combined with a single or multi/many core CPUs. Such processing units usually interact with external modules, such as memories, IP cores, or sensors. As such architectures are becoming more and more complex, they usually integrate components with variable latency and unpredictable behavior. For example, the latency of the communication with the external modules can be unknown, uncertain and sometimes unpredictable. Moreover, the scaling down of technology has brought significant deviations from the nominal values of many technology parameters, resulting in wide delay variations in circuit units (process variation [135]). Furthermore, a program can be inherently uncertain: it may feature an unknown number of loop iterations, unknown outcomes of conditional instructions evaluation, or unknown values of the incoming arguments of a function. These kind of incomplete information often significantly affect the performance of both embedded systems and HPC systems. In the following the impact of unknown information is deepened separately for these two domains.

Unknown Information in Embedded Systems Lately, the role of unknown, uncertain and unpredictable information in the Embedded Systems domain is significantly growing. Conventional embedded systems are not able to deal with incomplete information at design time. They are usually designed by completely specifying the desired behavior at design time. In other words, the set of instructions executed at each control step is established and fixed at design time. Unfortunately, the exact timing behavior of each unit cannot be guaranteed in any situation. For example, aging or thermal degradation may unpredictably affect the performance of the components. Moreover, the number of cycles required to complete the computation can be unknown and possibly unpredictable for modules with variable latency. When interacting with external modules, conventional hardware cores have to conservatively consider the worst case approach, to prevent the system from a failure, consequently affecting the performance [9]. When, instead, the latency varies unpredictably, as in the case of components degradation, conventional cores cannot guarantee correct execution at all. These factors play a dominant role, especially in those domains where the correct execution of the specification is critical (e.g., automotive, avionics, aerospace), otherwise catastrophic effects may occur. For example, the current trend in automotive consists in replacing more and more mechanical controllers with electronic devices, which manage the mechanical parts (e.g., controllers for brakes, gas pedal and steering wheels are now managed electronically). Unexpected or unpredictable events in this field can cause system failure, with consequent dangerous situations. Alternatives to conventional hardware cores have been proposed. However, none of them currently supports all the kinds of incomplete information, and they all present some limitations.

Unknown Information in HPC Systems Modern HPC systems are not immune from the performance worsening effect caused by unknown information at design time. In this domain, the inherent uncertain nature of a program can decrease the performance, due for example to the need of additional computation, either in form of additional instructions in the code or as additional job for the operative system. This is the case, for example, of the termination checks when the number of iterations for a loop is unknown. Irregular applications are significantly affected by the presence of unknown information, mainly due to the unpredictable structure of memory accesses. This class of applications well fit a distributed memory architectural model, due to the significant amount of data and to the data-intensive nature of the computation. The standard way of programming distributed architectures employs message passing to communicated among the nodes. The Message Passing Interface (MPI), is the de-facto standard for message passing, especially for high performance systems. With message passing programming models, the programmer usually exploits Single Program, Multiple Data (SPMD) control models. SPMD models require to associate each core with a process at the beginning of the computation. Such process will operate on its own chunk of data for the entire computation. In this model, communication happens only in precise application phases. Given the irregular, unpredictable and highly intensive nature of the synchronization in irregular applications, developing such applications with a SPMD programming model is not easy. Irregular applications employ datasets difficult to partition, thus shared memory abstractions are preferred. They also present data accesses to unpredictable location, and dynamically spawn new concurrent activities as the data is explored. At today, designers of irregular applications for HPC systems need to deal with these numerous sources of incomplete information, which make irregular applications the most challenging class of applications in terms of performance in the HPC domain.

3 Design Methodologies of High-Performance Embedded Systems

Modern embedded systems are based on MultiProcessor Systems-on-Chip (MP-SoCs). Architectures such as Tilera's TilePRO 64 or TileGX, Kalray's MPPA or Adapteva's Epiphany integrate hundreds or thousands of simple cores, interconnected together through fast Networks-on-Chip (NoCs). Due to the large amount of processing units they usually employ distributed non-uniform memory hierarchies, with fast, non-coherent scratchpads connected to each core. For this reason, the design methodologies of such systems seems to benefit from ideas coming from the HPC domain. The term High-Performance Computing (HPC) refers to a branch of computer science that concentrates on developing supercomputers and software to run on supercomputers. This Chapter discusses how some issues and challenges which are typical to the HPC domain affect the design of modern embedded systems. The Chapter is organized as follows: Section 3.1 overviews the current trend in the design methodologies of modern Embedded Systems, while Section 3.2 shows the main design issues typical of the HPC domain which are lately affecting also the ES domain. Finally, Section 3.3 discusses how this work handles adaptivity in both ES and HPC domains.

3.1 Current Trend in the Design of Modern Computing Systems

The current digital era is the result of longstanding and increasingly critical societal issues, as for example population explosion, global climate change, and human health and wealth. All these factors are causing an increasing pace of change in computing systems, which also reflects a reinforcing interplay between genetic, social, cultural and technical capacity. The key aspects characterizing modern trends include: increasing request of computational capacity, growing power efficiency requirements, improvements in computer communication systems, and exponential growth of data. In the following each one of the aspects characterizing current trends in the design of computing systems will be deepened.

Increasing Computational Demand Modern computing systems are rapidly evolving towards miniaturized electronic circuits etched into a single chip. The prediction about this evolution is described by the Moore's Law: "*the number of transistors that can be placed on an integrated circuit doubles approximately every 2 years*". In other words, each new technology generation over the past five decades doubled transistor density and increased frequency, while simultaneously reducing the power per transistor. Thanks to the Moore's law, even more demanding applications could improve performance with minimal changes to the software. However, a major paradigm change is lately taking place: while Moore's Law is keeping pace in terms of transistors density, new computing generation are characterized by only minor frequency increases and minor decreases in power dissipation per transistor. Indeed, further increasing the frequency leads to a significant increase in terms of power consumption for the device, and consequent overheating issues. Moreover, the decrease in power dissipation per transistor is limited by the growing density of transistors and by the reduction of their size. For these reasons, the current approach consists in exploiting the functionality of multiple processing units. Unfortunately, this is not transparent to most applications: existing software must be radically modified to efficiently execute on parallel architectures. The complexity of this task is one of today's main challenges.

Increasing Power Requirements The power needed by each device is not reduced accordingly to the increasing number of devices that can be packed into a single chip. This phenomena is known as *the end of Dennard scaling*. The scaling theory formulated by Dennard states that the power consumption per transistor scales with the reduction of transistors' size. Since we already reached the power limit, soon it will no longer be possible to use all the devices on a chip simultaneously. Some functionality should be turned off at a certain operation time to meet power constraints. Moreover, current chip technology is approaching the limits of physics [100]: there is a finite point at which one cannot further shrink the size of transistors on a chip and still have them function. Unless nanotechnology actually becomes operational, this trend will end around 2022 [99]. If true, this could have a wrenching negative economical effect worldwide.

Communication Minimization Many emerging classes of scientific applications are irregular (e.g., computer vision, machine learning, data mining, computational geometry, SAT solving, etc.). Irregular applications usually have very big datasets, which are difficult to partition in a balanced way. Moreover, they usually present irregular control flow, which makes inefficient synchronous programming models, such as SPMD. The synchronization requirements, characterizing this class of applications, differ significantly from those of traditional parallel computations. Instead of coarse-grain barrier synchronization, irregular computations require synchronization primitives that support efficient fine-grain atomic operations. The most common implementation mechanism for atomic operations uses mutual exclusion locks. However, the overhead of acquiring and releasing locks significantly decreases the performance on such large datasets. Furthermore, synchronization and communication represent data transfers, which are the main source of power dissipation during the computation. For these reasons, communication minimization is an important issue in the design of modern computing systems.

Exponential Growth of Data The rapid increase in the amount of published information or data (*Data Deluge*) is one of the most significant realities which our society is experiencing. As the amount of available data grows, the problem of managing the information becomes more difficult. This phenomena is affecting all the domains of information technology. The management of information is one of the most significant 21st century exponential trends. Information management consists of 3 major divisions: storing (23% growth per year), communicating (28% growth per year), and computing (58% growth per year) [98]. In spite of the 23% annual growth in data storage, scientists have recently passed the point where more data is being collected than we can physically store. This storage gap will widen rapidly in data-intensive fields [97]. In recent years, the amount of data passing through the global communication infrastructures has been increasing at a phenomenal rate. This expansion is largely attributed to the rising popularity of mobiles, social networks and cloud services, together with the expansion of Internet usage. In particular, networks are handling more data-rich contents and rising traffic from mobile devices such as smartphones and tablets. By allowing users to access contents through the Internet almost anywhere and anytime, smartphones are generating 10 to 20 times more data traffic than conventional mobile phones. More and more corporations are Cloud Computing based. These Cloud Computing services have been rapidly growing for years and represent one of the leading sources of increased business data traffic on networks. Moreover, thanks to the the rapid growth of the Internet, a large amount of information is being created and made available by individuals, business and government organizations. Such information, which are both structured and unstructured, need to be processed, analyzed, and linked. A study of the information explosion phenomena estimates that 95% of all current information is unstructured [117] (e.g., graphs, unstructured grids, unbalanced trees). Such kind of information significantly increases data processing requirements compared to structured information. The storing, managing, accessing, and processing of this vast amount of data represents a fundamental need and an immense challenge in order to satisfy needs to search, analyze, mine, and visualize this data as information.

Data-intensive computing is a class of parallel computing applications using a data parallel approach to process large volumes of data (typically terabytes or petabytes in size - Big Data problems). As most of the data are unstructured, most of the data-intensive algorithms are *Irregular Applications*. As previously mentioned, many important scientific applications are irregular. Among the examples, image processing, molecular dynamic simulations, sparse matrix computation and climate modeling [74]. Irregular applications apply computation independently to each data item of a set of data. This theoretically allows the degree of parallelism to be scaled with the volume of data. However, many design issues restrict the achievement of this goal. Current data-intensive computing platforms typically use a parallel computing approach combining multiple processors and disks in large commodity clusters. The data are partitioned among the available computing resources and processed independently to achieve performance and scalability. However, irregular data structures are difficult to partition in a balanced way, since unpredictable fine-grain synchronization is required among the tasks. They also suffer of poor locality. It is easy to generate load unbalance, which significantly worsen the performance. Indeed, modern computing systems rely on regular computation and locality exploitation to reach the peak performance.

3.2 HPC Design Issues in the Embedded Systems Domain

Thanks to technology scaling, architectural innovations, advances in compilation, and introduction of more compact and more powerful embedded processors, modern embedded systems have computing resources that by far surpass the computing power of the mainframes of the past decades. As previously mentioned, embedded systems are becoming HPC capable, and the two domains are converging to a new generation of computing platforms, called *embedded supercomputing systems* [66]. Nowadays, the common challenges in the design of supercomputers and ESs are many. The two domains are lately experiencing similar design trends. Designing these two classes of architectures requires a holistic approach that relies on tightly coupled hardware-software co-design methodologies [70]. Moreover, reconfigurable computing (RC) is lately becoming an exciting approach for high-performance computing, which at today use to integrate reconfigurable cores. The main advantage of such systems is that they can adapt to static and dynamic application requirements better than those with fixed hardware. Furthermore, the power/performance ratio for reconfigurable hardware is at least an order of magnitude better than that of general-purpose processors. However, for RC technology to be deployed on a large scale, a number of important gaps in theory and in practice have to be addressed, as for example efficient interfaces to

RC components and to external devices, or support for efficient hardwired and softcore instruction processors. Since 2000s the efforts to design very aggressive and very complex wide issue superscalar processors have essentially come to a stop, despite new progress in integration technology. Given the above discussed limitations in further exploiting the Moore's law, in the last decade, it has become more and more obvious that the quest for the ultimate performance on a single chip uniprocessor is becoming a dead-end [37]. Indeed, even if there are still significant amounts of unexploited Instruction-Level Parallelism (ILP) left, it becomes more and more difficult to extract it. At the same time, further increasing the clock frequency is also getting more and more difficult because of heat problems and energy consumption.

For these reasons, the current approach in both the domains consists improving the performance by exploiting the tonalities of multiple processors, instead of scaling the performance by improving the single processor performance. We assisted to a massive paradigm shift towards parallel architectures, which led to integrated multiprocessors on chip (Systems on Chip - SoCs) in the ES domain, and to massively parallel supercomputers in the HPC domain. This paradigm shift has a profound impact on all aspects of the design of high-performance systems, and consequently also of embedded supercomputers. Parallel architectures introduces new design complexity issues, by increasing for example the complexity of memory hierarchies or by requiring the design of new interconnection strategies, since bus-based interconnects are no longer suited. Memory subsystems and interconnections represent a critical part of the design because of scalability issues. Scaling these subsystems in a resource-efficient manner to accommodate the foreseen core count is a major challenge, especially for Systems-on-Chip (SoCs). Indeed, the off-chip bandwidth is expected to increase linearly rather than exponentially. As a result, a high on-chip cache performance is crucial to cut down on bandwidth. Thus, cache hierarchies are in need of innovation to make better use of the resources. However, parallel architectures which employ a shared memory model require efficient support for cache coherence. A great deal of attention was devoted to scalable cache coherence protocols in the late 80s and the beginning of the 90s. Subsequently, the latency/bandwidth trade-off between broadcast-based (snooping) and point-to-point based (directory) cache coherency protocols has been introduced. However, now that systems can host hundreds of cores on a chip, technological parameters and constraints are quite different. Modern parallel architectures often consists in an heterogeneous mixture of processing units, each one of them having its own design complexity issues. For example, special-purpose cores have significant impact on the memory hierarchy of the system, and require specially designed communication protocols for fast data exchange among them. A major challenge is the design of a suitable high-performance and flexible communication interface between less traditional computing cores (e.g. FPGAs) and the rest of the multi-core system. Moreover, programming such heterogeneous parallel systems is difficult and time-consuming. The programmer must explicitly express and manage concurrency in terms of synchronization among threads and communication among processors. Traditional software-based synchronization methods do not provide alone complete support for the parallelization of complex embedded HPC systems. For this reason hardware-based approaches are often needed. Common programming languages featuring arbitrary pointer manipulations (like C or C++) make automatic parallelization extremely difficult. This problem is very crucial, since the performance gain is strictly limited by the amount of parallelism that we can extract from the user's code, regardless of how many processors are available. Furthermore, modern applications are becoming more and more demanding, many of them have hard or soft real-time requirements, and they often show irregular nature. These elements further complicate the generation of efficient parallel code. Indeed, modern parallel architectures need sophisticated compilation techniques to generate highly optimized code. While automatic parallelization has been quite successful in the past for homogeneous shared memory architecture, currently, compiler technology is not able to provide good result. The quality of the results obtained for other set of application types, such as pointer-based programs, and for heterogeneous machines, with potentially different memory models, is far lower. Traditional approaches to compiler optimizations are based on static analysis and transformation which can no longer be used in a computing environment that is continually changing. Compiler adaptation through tuning can represent a solution at this problem. However, tuning the optimization heuristics and parameters for new processor architectures is a time-consuming process. This process can be automated by machine learning. Finally, as the demand for power efficient devices grows, compilers have more and more to consider energy consumption in addition to space and time. The key challenge is to exploit compile-time knowledge about the program to use only those resources necessary for the program to execute efficiently. The compiler is then responsible for generating code where special instructions direct the architecture to work in the desired way. This would allow to gate off unused parts or in dynamically resizing expensive resources with the help of the run-time system.

Exploiting the parallelism offered by multi-core architectures requires powerful new programming models, which allow the programmer to express parallelism, data access, and synchronization. Parallelism can be expressed as parallel constructs and/or tasks that should be executed on each of the processing elements. According to the work in [37], most probably, the best solution resides in a combination of different programming paradigms. For example, the concept of task expressed by OpenMP could be combined with the concept of "places" defined by X10, or "regions" defined by Fortress, or "locale" defined by Chapel, which identify addressable domains. OpenMP simplifies the programmer work by abstracting the communication details, while the addressable domains allow to distribute the computation across a set of machines in cluster environments. The data access and the synchronization models can be distributed (i.e., through message passing models, like MPI) or shared (i. e., using global memory models, like OpenMP, or global memory abstractions, like PGAS). The simplicity of parallel programming approaches is becoming as important as the performance they yield. Ideally, the programming model should allow the programmer to transparently work with shared and distributed memory at the same time. However, current attempts, like Co-Array Fortran, UPC, X10, Fortress, and Chapel among the others, still reflect the difference between these two separated domains. In other words, the way to perform data access and synchronization depends on where the computation is performed. As hardware accelerators can also be seen as different execution domains with local memory, it is interesting to note that solving this challenge will also provide transparent support to run on accelerators.

Run-time systems assume an important role in embedded supercomputing. The run-time system consists of a collection of facilities, such as dynamic memory allocation, thread management and synchronization, middleware, virtual machines, garbage collection, dynamic optimization, just-in-time compilation and execution resources management. It handles dynamic behavior that cannot be determined through static analysis by the compiler. However, operating system and runtime research should be more closely coupled to hardware and compiler research in the future in order to integrate multi-core heterogeneous systems and to seamlessly support dynamic reconfiguration and interoperability. Indeed, the run-time systems should automatically or semi-automatically adapt to different heterogeneous multi-cores based on the requirements of the applications, available resources, and scheduling management policies. Even in presence of unknown information at compile-time, the compiler can provide parametric results to support such tasks.

In conclusion, it is clear that the paradigm shift to multiprocessor architectures is so profound that it is affecting almost all aspects of system design in both ES and HPC domain, contributing to the convergence of the two domains. A lot of research and tool development are needed to provide good performance on such complex architectures.

3.2.1 Design of Irregular Applications on HPC Systems

Despite high-performance computing designers use to couple MPI with SPMD, irregular applications do not benefit from this programming model. Indeed, SPMD models require balanced partitioning of the application among the nodes to provide good performance. For this reason shared memory abstractions, such as the Partitioned Global Address Space (PGAS) programming model, are preferred for irregular applications, which presents dataset very difficult to partition in a balanced way. PGAS assumes a global memory address space which is logically partitioned among the nodes. Every memory portion is local to each process or thread. PGAS attempts to combine the advantages of a SPMD programming style for distributed memory systems (as employed by MPI) with the data referencing semantics of shared memory systems. This is more realistic than the traditional shared memory approach of one flat address space, because hardware-specific data locality can be modeled in the partitioning of the address space. The PGAS programming model is implemented in many languages. However, many of these languages still target a SPMD control model, thus resulting inadequate for irregular applications. As an example, Unified Parallel C (UPC) is an extension of the C programming language designed for high-performance computing on large-scale parallel machines. The programmer is presented with a single logically shared, physically partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. As previously mentioned, UPC uses a SPMD model of computation in which the amount of parallelism is fixed when the program starts, typically with a single thread of execution per processor. It thus provide good performance only for regular computation. At the state of the art, the literature does not show any efficient design methodology for irregular applications on distributed memory systems.

Memory Management for Irregular Applications The memory management in HPC systems depends on the target memory architecture, as well as on the parallel programming model adopted. The target memory architecture can be shared, distributed and non uniform, or distributed and uniform. As above mentioned, C-like languages extensions are employed for describing the application, mainly because of their simplicity to use. The main strength of such languages is the hiding of the architectural details. However, the main strength of such languages is also their main limitation, since they abstract too many important details for maximizing the performance of modern computing systems. Indeed, without considering the underlying architecture, it results difficult to write programs which exploit the availability of multiple functional units, possibly heterogeneous, running in parallel. For this reason, several extensions for better expressing parallelism in standard languages have been proposed. Such extension aim to define parallel programming models. As previously discussed, one of the most important and well-known examples is OpenMP [26], which extends C/C++ and Fortran programs with task-parallel language constructs. The developer identifies parallel kernels in the code, and he or she inserts the constructs provided by the parallel programming model in the code, in the form of pragmas and API calls. Since OpenMP expresses the parallelism by identifying parallel kernels in the code, this model is known as computation-parallel (or task-parallel), in contrast to other models, such as MPI, which identify parallelism by partitioning the data (i.e. data-parallel models) [74]. OpenMP targets shared memory multiprocessors with uniform memory architectures, abstracting the communication management. The model works well for architectures with few nodes, while it does not provide adequate scalability for modern supercomputers, which relies on a multitude of nodes, tightly interconnected to private, non coherent cache memories to exploit locality. Recently, modifications to OpenMP compilers that enhance the support of these architectures have been proposed [31]. Nevertheless, they still require knowledge of the underlying architecture to trigger data partitioning. Moreover, OpenMP programs risks to incur in the data race problem. Whenever two parallel tasks, such as two iterations of a parallel loop, access the same memory location, and at least one of them tries to write that location, a data race condition may occur. In this case, different parallel schedules may provide different answers. Due to their not repeatable nature, data race problems are difficult to locate and eliminate. Furthermore, shared memory architectures are mainly bus-based. This often leads to bus saturation, especially for memory-intensive high-performance computing applications, such as irregular applications. Other solutions, such as crossbar switches, have been proposed. However, they came at a significant cost, either in terms of expense to interconnect or in loss of performance. Due to these issues, irregular applications usually target distributed memory systems. Such systems are characterized by scalable networks, which make large systems much more cost-effective, at the cost of increased programming complexity. Every time a processor needs to access a data which is not stored in its own local memory, it needs to send a remote request to the processor which hosts the required data in its local memory. In other words, distributed memory systems are naturally programmed with message passing programming models. Despite it is possible to generate code for message passing systems from OpenMP, the user has no way to handle communication or data placement [83]. This can result in significant performance loss. Message passing programming models can effectively map the architecture of novel many-core supercomputers. The Message Passing Interface (MPI) [24] [17] is the de-facto standard for message passing for high-performance systems. With message passing programming models, the programmer usually exploits Single Program, Multiple Data (SPMD) control models, where, at the beginning of the computation, each node is associated with a process that operates on its own chunk of data. Typically, the data associated to a given processor are those which the processor will use during its computation, in order to reduce communication overhead. The communication usually happens only in precise application phases. Developing irregular applications with these programming models is not easy. Irregular applications employ datasets difficult to partition, thus shared memory abstractions are preferred. They also present data accesses to unpredictable location, and dynamically spawn new concurrent activities as the data is explored [74]. High-Performance Fortran (HPF) [128] [83] represents another example of parallel programming model. It extends the Fortran language according to the data-parallel programming model. HPF has been introduced for distributed memory parallel computers as an alternative to MPI. Despite HPF embodies good ideas on how to extend a language to incorporate data parallelism, it did not achieve large success [83]. The reasons for that were multiple. On one hand the compiler technology was immature at that time. In order to exploit the new functionalities introduced by HPF, the compiler community would have to rethink about compilation strategies, and to rewrite big part of the compiler. On the other hand, the complex relationship between performance and implementation made it difficult to identify and correct performance issues. Moreover, HPF was poorly portable. Given a single version of a program, HPF was meant to produce multiple versions of the parallelized program, according to the target architecture. Most of the users went back to MPI when they realized that this feature was missing.

The high performance computing community has introduced the Partitioned Global Address Space (PGAS) programming model [20] as an approach to support a shared memory abstraction over distributed memory large-scale parallel machines without neglecting data or thread locality. The PGAS programming model is implemented in languages such as Unified Parallel C (UPC) [59], Co-Array Fortran [80], the Global Arrays (GA) Toolkit [109], X10 [133] and Chapel [58], among several others. These programming models rely on communication run-time libraries which manage data exchange between distributed address spaces, such as ARMCI [110] and GASNet [36]. The UPC compiler automatically transforms UPC programs (C programs with parallel annotations) in C programs that employ the underlying communication layers. However, it still targets a SPMD control model, thus resulting inadequate for irregular applications. X10 and Chapel, on the other hand, support asynchronous task execution and the possibility to reason about data locality. OpenSHMEM [45] is a specification standardizing an increasing number of SHMEM implementations, a communication library that uses one-sided communication and a global address space. TSHMEM is a lightweight SHMEM implementation for Tilera processors. However, also TSHMEM [91] still implements a SPMD control model and misses some of the features that may enable more efficient execution of irregular applications. The goal of this work is to introduce a set of compiler transformations for a global address space library optimized for irregular applications.Finally, the Explicit Multi-Threading (XMT) is a parallel programming model designed for exploiting on-chip parallelism [108]. The origins of XMT date back to when the Cray Inc. released the XMT supercomputer in 2005 [61]. Such architecture was designed to have low-overhead parallel threads and high on-chip memory bandwidth. The XMT environment includes also a XMT compiler and a behavioral simulator. The compiler front-end is a SUIF-based [60] translator, which converts XMT constructs into regular C code with assembly templates. It also automatically transforms parallel regions delineated by spawn-join statements in parallel function calls. The compiler back-end is GCC-based [11]. Among the features of the XMT environment, a simple thread execution model and an efficient prefix-sum instruction for synchronizing shared data accesses. Despite low thread overhead, thread coarsening is still necessary to some extent, but it can usually be automatically applied by the XMT compiler [137]. The prefix-sum instruction provides more scalable synchronization than traditional locks, and the simple run-until-completion thread execution model (i.e. no busy-waits) does not impair performance [107]. However, the Explicit Multi-Threading programming model suffers of poor portability. It is hard to obtain the same advantages on other architectures. Moreover, this approach is not scalable, since it is based on shared memory systems.

Beside a shared memory abstraction, other features that allow a more efficient execution of irregular applications are multithreading and fork/join control models. Multithreading allows tolerating, rather than reducing, the latencies for accessing data, both in local and remote memory locations. The fork/join control model enables dynamic creation of fine grained threads, allowing better exploitation of the inherent applications' parallelism. In the high performance environment, these features can be found on the Cray XMT [61], which is the successor of the Tera MTA and Cray MTA-2 multithreaded supercomputers. These custom machines for irregular applications also provide a custom compiler, which extracts fine grained threads from nested loops. Parallelism discovery happens semi-automatically through pragma annotations. The proposed approach in the HPC domain aims at designing a compiler infrastructure for irregular applications with similar functionalities to those of the Cray XMT's compiler. However, this work targets clusters of supercomputers, rather than custom multiprocessors. Furthermore, a run-time, rather than hardware components, provide the features than enhance execution of irregular applications. Finally, this work tries to limit as much as possible developer intervention in parallelism discovery and code optimization, by empowering the compiler of the execution of as many tasks as possible in the design process.

3.3 Adaptivity Management in ESs and HPCs

As previously discussed in Chapter 2, unknown, uncertain and unpredictable information are assuming a growing role in the design of modern computing systems. Furthermore, with the convergence of embedded and HPC systems, these two domains are experiencing similar issues. Indeed, in both cases, designers are shifting toward more and more flexible systems, which are able to dynamic adjust their behavior according to run-time events. Moreover, in both cases there is growing need of automatic tools and frameworks to help studying the adaptivity properties of applications and architectures. This thesis work aims at addressing this issue by providing support for adaptivity analysis in the two domains. The more the applications show inherent parallelism, the more adaptivity analysis will provide efficient designs. More in detail, embedded systems are characterized by finer grained parallelism, with respect to supercomputers. Thus, in general, even if embedded systems and HPC systems share the same design issues, they require to face the problem of adaptivity management at different abstraction levels. For this reason, this work proposes two different approaches to address the same problem in the two domains.

Adaptivity Analysis Support for ES Design Unknown information and unpredictable events in the embedded systems domain are addressed in this work by providing a dynamic scheduling technique for the design automation of adaptive hardware cores. Adaptivity analysis is integrated into the PandA framework to define an explicit activation mechanism for the instructions. Such mechanism requires hardware support. For this reason, this work also proposes a novel adaptive controller architecture. By explicitly activating the instructions execution, based on run-time conditions, this approach supports either unpredictable events which can potentially lead to system failure, and those who can actually speed up the execution, by dynamically revealing the parallelism.

Adaptivity Analysis Support for HPC Design Unknown information and unpredictable events in the HPC domain are usually handled by run-time systems. However, such systems only have visibility of run-time events, without being able to study the correlation among them. On the other side, compilation frameworks cannot statically predict run-time events, but they have a privileged point of view about the applications' properties. For this reason in this work adaptivity analysis in modern supercomputers is addressed by providing compiler support to a proper run-time system. The proposed technique is integrated into the LLVM compiler, and it targets cluster of supercomputers running the Global Memory and Threading (GMT) run-time system. Adaptivity analysis in this case performs at first automatic parallelization of irregular applications, and then it applies a novel set of transformations which help the run-time system to efficiently execute in parallel such challenging class of applications.

4 Adaptivity Analysis in Embedded Systems Domain

Modern hardware cores necessarily have to deal with many sources of unknown or uncertain information, which often is also unpredictable. These systems need to continually function in unpredictable environments. For example, they constantly face the possibility of encountering component failure, or unexpected situations in which they may be forced to operate under lower capacity. Consequently, many of such systems are designed to provide different levels of quality of service. On the other side, unexpected situations may allow to operate under higher capacity, if the system is able to exploit these changes. Systems designers must take into account that components with variable latency and unpredictable behavior are becoming predominant in modern hardware chips. Moreover, uncertainty is also due to the scaling down of technology, which has brought significant deviations from the nominal values of many technology parameters, resulting in wide delay variations in circuit units (*process variation* [135]). CMOS process variability is a major challenge in deep-submicron SoC designs. The variations in transistor parameters are complicating both timing and power consumption prediction. Furthermore, a program can be *inherently uncertain*: it may feature an unknown number of loop iterations, unknown outcomes of conditional instructions evaluation, or unknown values of the incoming arguments of a function. All these issues must be necessarily considered to prevent the system from a failure (e.g., a wrong execution).

Conventional hardware cores underperform when dealing with unknown or uncertain information. Indeed, common High-Level Synthesis (HLS) approaches require to specify the complete behavior at design-time. In other words, the set of instructions executed at each control step is established and fixed at design time. Unfortunately, the exact timing behavior of each unit cannot be guaranteed in any situation, as in the case of aging, thermal degradation or presence of components with variable latency. For these reasons, conventional embedded systems present significant restrictions in supporting uncertainty. When interacting with external modules (e.g., memories, sensors, IP cores), conventional hardware cores have to conservatively consider the worst case approach, to prevent the system from a failure, consequently affecting the performance [41] [118]. When, instead, the latency varies unpredictably, as in the case of components degradation, conventional cores cannot guarantee correct execution at all. These factors play a dominant role, especially in those domains where the correct execution of the specification is critical (e.g., automotive, avionics, aerospace), otherwise catastrophic effects may occur. Moreover, potentially independent code fragments must be sequentialized in presence of inherent uncertainty of a program, due for example to unknown number of loop iterations, unknown outcome of conditional instructions evaluation or unknown value of the incoming arguments of a function.

One way to achieve high dependability, while simultaneously increasing the performance, is to make the system adaptable to changes, if possible, without sacrificing maintainability. In such cases, the designer must cater not only for the temporal and functional system properties, but also for its ability to dynamically adapt itself, as a response to external and/or internal stimuli. To be able to reason about adaptivity, proper modeling and analysis framework, suitable for adaptive systems, are needed. Trying to address this problem, this Chapter introduces a novel high-level synthesis flow for the design of adaptive systems. Manually implementing efficient hardware designs requires great skills and extensive knowledge about the underlying infrastructure. RTL designs quickly become complex, making their creation, testing and integration a time-consuming and error prone process. Automating the design of such systems can reduce the development time from weeks to minutes. Thus, this thesis work proposes the integration of novel design phases into a typical HLS flow. The proposed approach is modular, since it does not require complete restructuring of the conventional HLS phases, thus facilitating the integration into preexisting (and possibly commercial) tools, while simplifying maintainability.

Adaptivity analysis aims at identifying the adaptivity level of an application. In general, adaptivity analysis can be employed to discover many kinds of adaptive traits of the program. For example, it can reveal how much the specification can benefit by multithreading though the analysis of thread level parallelism, structure of memory accesses and associated latency, which can overlap to the computation when switching from one thread to another. As another example, it can suggest which is the best partitioning of an application over the available resources, for example by building proper intermediate representations, which can be used as fast simulators to measure changes in terms of execution time on the different resources. Among the different possible uses of adaptivity analysis, this thesis work focuses on dynamic scheduling techniques. More in detail, adaptivity analysis is used to measure the instruction level parallelism which is inherited in the specification. In this way, not only it is possible to exploit ILP, but it is also possible to identify an explicit activation mechanism for the instructions, which serves as support for execution under uncertain conditions. The results of this analysis are used to define a novel dynamic scheduling technique, called dynamic Activating Conditions (AC)-scheduling.

As previously discussed, conventional hardware cores are usually composed of a datapath and a centralized Finite State Machine (FSM) acting as controller [145]. The former determines the operations to be executed in each clock cycle based on the control flow, while the latter actually performs the operations. To achieve increased parallelism level, by exploiting inherent uncertainty of the specification, the FSM can be extended with additional states to support the selection at run-time, i.e. when unknown information will be resolved, of the proper scheduling among a set of scheduling orders, precomputed at design time. However, this increases the area up to the Cartesian product of the number of states [67], often producing unfeasible solutions. In general, the area grows proportionally to the adaptivity level which the architecture can support. Thus, this kind of cores, like conventional cores, usually accept only a subset of the possible behaviors for the system, with consequent considerable restrictions. The techniques proposed in literature to support unbounded modules [88] have some limitation in the amount of parallelism that the resulting core can exploit. On the other side, the literature proposes several dynamic scheduling (e.g., [125]) and speculation techniques to improve the cores performance by handling inherent uncertainty of applications [57]. However, such approaches only address inherent uncertainty of the program, while they do not support other sources of uncertainty. Despite significant research effort has been devoted in this direction, at today, the centralized finite state machine (FSM) remains the most common architectural model for the controller in high-level synthesis. Such model results incompatible with adaptive systems implementation. Alternative architectural models proposed so far for the controller part, such as Petri nets based parallel controllers and distributed controllers, led to communication and synchronization overhead, excessive area overhead and limitations related to decomposition techniques. For these reasons, in this thesis work the dynamic AC-scheduling targets a novel architectural model for adaptive systems, aiming at overcoming the limitations of existing hardware core architectures.

The main contributions of this thesis work can be summarized as follows:

- Definition of Activating Conditions (ACs)
- Definition of Adaptivity Analysis for AC-based Dynamic Scheduling
- Definition of a proper Intermediate Representation (IR), namely the Extended Program Dependence Graph (EPDG), needed to implement the dynamic AC-scheduling technique
- Definition of Minimum Control-Flow Dependences (MCFDs), needed to build the EPDG
- Design flow automation and integration of adaptive cores automatic design into HLS tools

- Definition of a set of optimizations and extensions for the AC-based Dynamic Scheduling technique
- Definition of an architectural model for adaptive hardware cores

The remainder of this Chapter is organized as follows. Section 4.1 introduces the dynamic AC-scheduling technique for adaptive cores. Moreover, it defines the concepts of Minimum Control Flow Dependences (MCFDs), Extended Program Dependences Graph (EPDG) and Activating Conditions (AC). Then, it outlines how the AC-based dynamic scheduling methodology can be implemented and integrated into a typical HLS flow. Finally, it overviews some possible optimizations and extension for the proposed technique. Section 4.2 describes the proposed target architecture to match adaptivity analysis and to provide support for its implementation. Finally, Section 4.3 shows some results, obtained by integrating the dynamic AC-scheduling into the PandA framework.

4.1 Adaptivity Analysis for AC-based Dynamic Scheduling

This Section describes the Dynamic AC-Scheduling approach, introduced to overcome the above discussed limitations of modern hardware cores, by increasing their adaptivity capabilities. The proposed technique exploits adaptivity analysis to define a new model of adaptive embedded systems, and to provide a formal methodology for the high-level synthesis of such systems. More in detail, the dynamic AC-scheduling allows to automate the design of hardware cores which can dynamically adapt the instructions scheduling according to behaviors which are unknown, uncertain and unpredictable at design-time. Adaptivity analysis for the design of embedded systems is defined in this thesis work as a technique to define and compute a set of conditions under which each instruction can be executed. Such conditions are named Activating Conditions (AC). The introduction of the ACs is based on the observation that the exact timing behavior of many components, modules and functional units in the architecture is possibly unknown. For this reason, assigning a control step to each operation in the specification may restrict the performance, or even worst it may result in incorrect behaviors of the system. Since unknown information prevents to establish at design time the optimal scheduling for the instructions, the ACs parametrically express uncertain information. At run-time, when unknown information is resolved, the ACs indicate when an instruction can execute, since its dependences are satisfied. In other words, every unknown information represents a parameter in the AC. Every time some unknown information is resolved, it is substituted in the corresponding AC. At run-time, once all the parameters have been substituted with the corresponding run-time information, the corresponding AC is evaluated. If the AC is satisfied, then the corresponding instruction is safely executed. Thus, this mechanism allows dynamic scheduling of the instructions, which can autonomously establish when to execute, based on the information provided by the corresponding AC. In this way, neither assumptions about components latency nor worst case approach are required. To implement adaptivity analysis a proper Intermediate Representation (IR) has been defined, namely the Extended Program Dependence Graph (EPDG), able to show the available parallelism, while providing a parallel execution model of the specification. Adaptivity analysis for the design of adaptive hardware cores requires a proper hardware support to implement AC-based dynamic scheduling. For this reason, this thesis work proposes a novel architectural model, which substitutes the centralized FSM with a lightweight adaptive controller, where the operations can reorder their execution at run-time, according to the ACs evaluation. The architecture, which will be discussed in Section 4.2, is based on a very simple token-based communication mechanism. Moreover, a proper formalization for the design process, which allows its automation and integration into a typical HLS flow in a modular way, will be provided. Experimental results, discussed in Section 4.3, show significant performance increase, with limited area overhead, with respect to state-of-the-art approaches.

4.1.1 Motivating Example for AC-based Adaptivity Analysis in ES Design

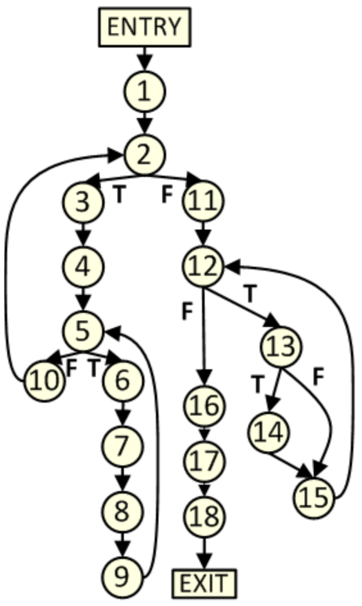
This Section highlights advantages and drawbacks related to common HLS techniques, by means of a motivating example. The main goal of adaptivity analysis in the embedded systems domain is to support uncertainty in the design automation of modern hardware cores. This issue is addressed by proposing a dynamic scheduling methodology. For this reason, this motivating example concentrates on the aspects of each traditional technique which can fit the design automation of such flexible hardware cores. Moreover, considering the purpose of this thesis, particular attention will be devoted to the activation mechanism for the instructions.

The sequential nature of C-like languages, used for describing the specification, mismatches with the inherent parallel nature of hardware cores. For this reason, the scheduling task in high-level synthesis is usually based on the analysis of an Intermediate Representation (IR), which represents the original program in an equivalent form (usually graphs), more suitable for exposing parallelism. The choice of the IR is extremely crucial, since it may significant affect the performance of the scheduling algorithm, and consequently of the entire system. In the context of the design of adaptive hardware cores, the IR must be able to provide a parallel execution model for the specification. In other words, the arcs in the graph should be able to express the partial ordering relation among the instructions, without missing needed information and at the same time without overspecifying sequentializations when not necessary. The most common approach for the scheduling of embedded systems employs the Control-Data Flow Graph (CDFG) [32] as IR [65] [52] [51] [69]. Usually, the use of the CDFG is coupled with a list scheduling algorithm [65], which statically assigns at design time a control step to each instruction. This information is then used to obtained the centralized FSM, representing the controller for the design. As previously discussed, this approach cannot deal with unpredictable components. Moreover, this approach shows restrictions due to the IR itself. Indeed, the CDFG exposes the parallelism at the basic block level, i.e. groups of instructions belonging to different basic blocks are sequentialized, while instructions belonging to the same basic block can simultaneously execute, if they are data independent. As a consequence, this approach well fits the synthesis of data flow intensive specifications, while limiting parallelism exploitation for control flow intensive applications. Indeed, the CDFG is based on the Control Flow Graph (CFG). The CFG contains some control flow arcs, which represent neither data nor control dependences. These arcs are superfluous for scheduling purposes and may cause useless sequencing. As an exam-

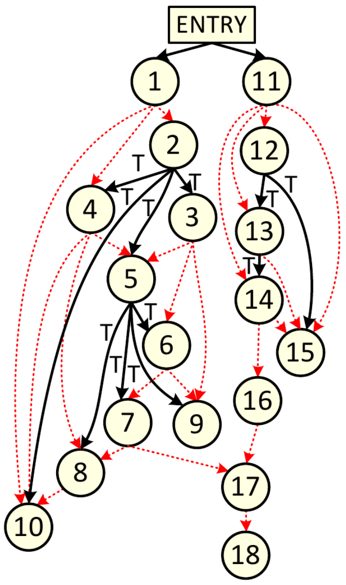
```
        void motiv(int a, int b, int c, int * out){
            int i, j, n, t, k, z, res;
  1:        i = 0;
  2:        while ( i<a ){         // loop1
  3:            j = 0;
  4:            n = i + 1;
  5:            while ( j<n ){     // loop2
  6:                t = j + 1;
  7:                c = a + t;
  8:                i = i + c;
  9:                j = j +1;
                }
 10:            i = i + 1;
            }
 11:        k = 0;
 12:        while (k < 10){        // loop3
 13:            if (a > k)
 14:                b = k + a;
 15:            k = k + 1;
            }
 16:        z = a * b;
 17:        res = z + c;
 18:        *out = res;
        `
```

Figure 4.1: Motivating example C code.

ple, consider the code fragment in Figure 4.1, containing three loops: $loop_1$, $loop_2$ nested into $loop_1$, and $loop_3$. The loops $loop_1$ and $loop_2$ are independent from $loop_3$, since neither data nor control dependences occur among the instructions in the corresponding bodies. In the following the notation $loop_i$ will be used indiscriminately to indicate the $i-th$ loop in the specification and the set of instructions which belong to that loop. Figure 4.2a shows the CFG associate to the code in Figure 4.1, which highlights how such independent loops are sequentialized through the arc between instructions 2 and 11. Such dependence is present in the CDFG as an arc between the basic block which contains instruction 2 and the basic block which contain instruction 11. Alternatives to the CDFG have been proposed, such as the Program Dependence Graph (PDG) [62]. Such graph contains the minimum data and control dependences, thus not causing potential unnecessary sequencing. The PDG for the specification in Figure 4.1 is shown in Figure 4.2b, where red dotted arcs represent data dependences, while black solid ones represent control dependences.



(a) CFG

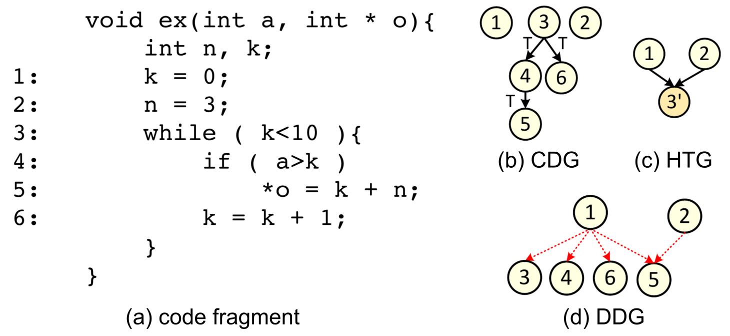


(b) PDG

Figure 4.2: CFG (a), and PDG (b), for the specification in Figure 4.1.

Unfortunately, the PDG alone does not generally provide all the information needed to build the controller. Indeed, since it does not contain control flow information at all, it lacks of the subset of control flow arcs that are needed to ensure correct execution. This information, which is mainly present in control flow intensive specifications, is essential to create correct designs. For example, consider the instruction 16 in the PDG in Figure 4.2b. It must be activated after instruction 14 due to a data dependence. However, since instruction 14 belongs to a loop, then instruction 16 must wait for the termination of $loop_3$ to guarantee correct execution. Such information, represented by the arc from 12 and 16 in the CFG, is instead missing in the PDG, which thus allows the instruction 16 to read the wrong value of the variable b. As another example, consider instruction 12 in the motivating example of Figure 4.1, which is data dependent on instruction 11, hence it can be executed after instruction 11 has been executed. However, once the first iteration of $loop_3$ is terminated, instruction 12 must be reactivated. Again, this information, that is represented in the CFG through the arc between 15 and 12 is not present in the PDG.

The notion of Activating Conditions (ACs) has been introduced in [96] for the generation of parallel source code through parallelizing compilers. This thesis work provides a methodology for coarse grained task scheduling on multiprocessor architectures. Such technique seems to theoretically fit well the design of hardware cores with support for uncertain and unpredictable information. However, it should be extensively revised to provide correct and efficient designs, when employed for fine-grained instruction scheduling of adaptive embedded systems. Indeed, this approach aims at exploiting parallelism, by implementing auto-scheduling with the support of the Hierarchical Task Graph (HTG) [122] [71], which is used as IR. At each hierarchical level, the HTG is a Task Graph enhanced with control and data dependences, thus encompassing parallelism at all levels. The ACs for each task node in the HTG are computed according to its incoming data and control dependences. Independent task nodes at the same hierarchical level can be simultaneously executed. While this approach theoretically well fits software-based auto-scheduling techniques, the use of the HTG as IR restricts parallelism exploitation when implementing hardware-based auto-scheduling, where it is possible to take advantage of finer parallelism. As an example, consider the code frag-



```
      void ex(int a, int * o){
          int n, k;
1:        k = 0;
2:        n = 3;
3:        while ( k<10 ){
4:            if ( a>k )
5:                *o = k + n;
6:            k = k + 1;
          }
      }
```

Figure 4.3: Example code (a), and corresponding CDG (b), HTG (c) and DDG (d).

ment in Figure 4.3a. As shown in the corresponding Data Dependence Graph (DDG) in Figure 4.3d, instruction 5 is data dependent on instruction 2. Since instructions from 3 to 6 belong to a loop, they will be represented as a single composite node $3'$ in the corresponding HTG, shown in Figure 4.3c. By construction, in the HTG the execution of the node $3'$ will start after instruction 2 has been executed, due to the data dependence between 2 and 5. This means that, despite the instructions 3, 4 and 6 are totally independent from instruction 2, their execution will be prevented until instruction 2 will be executed. The more the loop bodies are large, the more this problem affects the performance.

4.1.2 The Extended Program Dependence Graph

Let V be the set of instructions in the specification, and $E = \bigcup e(v_i, v_j)$ the set of dependences among each pair of instruction $v_i, v_j \in V$. The *Extended Program Dependence Graph (EPDG)* is a direct graph $G(V, E)$, where each arc $e(v_i, v_j) \in E$ represents a dependence between its source instruction v_i and its target instruction v_j. In other words, the execution of v_j is triggered when a condition depending on v_i is satisfied. In this way the EPDG defines a conditional precedence relation on their execution order which is valid only when both source and target instructions must be executed. As suggested by its name, the *EPDG* extends the PDG with *minimum control flow dependences*. The set V of vertexes in the EPDG is equal to the set of vertexes V_{PDG} in the PDG ($V \equiv V_{PDG}$), while the set E_{PDG} of arcs in the PDG is, in first analysis, a subset of the set E of arcs in the EPDG ($E \supseteq E_{PDG}$). However, Section 4.1.7 will show how the set of data dependences in the EPDG can be reduced. When these optimizations are enabled, the set of arcs in the EPDG is not anymore a superset of the set of arcs in the PDG. Starting from the PDG, control flow arcs must be added to build the EPDG in those situations in which data dependence arcs do not suffice to guarantee correct execution, as above described for instruction 16 in the example PDG in Figure 4.2b (see the extract in Figure 4.4), or in which control and data dependence arcs not suffice to extract information about the next instruction to be executed, for example to manage loop body execution for the iterations from the second one on, as previously discussed for instruction 12 in the example PDG in Figure 4.2b (see the extract in Figure 4.4).

Definition *The Extended Program Dependence Graph (EPDG) is a direct graph $G(V, E)$ where each node $v \in V$ represents an operation of the specification, and each arc $e(v_i, v_j) \in E$ represents a dependence between two operations. The dependences in the EPDG can be minimum data dependences, minimum control dependences or minimum control-flow dependences.*

The EPDG corresponding to the motivating example of Figure 4.1 is shown in Figure 4.5.

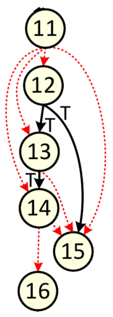


Figure 4.4: Extract of the PDG in Figure 4.2b.

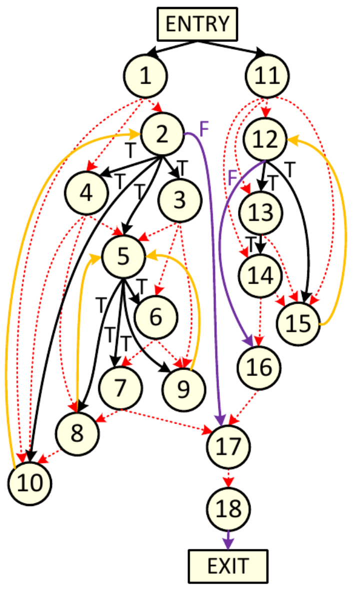


Figure 4.5: EPDG for the example in Figure 4.1.

4.1.3 Data Dependences Management

In first analysis, the entire set of data dependences which are present in the PDG is included in the EPDG, since they are defined as minimal. However, as it will discussed in Section 4.1.7, such a set can be reduced when the role of the IR is to provide a parallel execution model. The entire set of dependences in the PDG will be still needed for other HLS tasks, such as register allocation. Data dependences define a precedence relation between two instructions. Such relation can be unconditional or conditional, depending on the context. For example, the data dependence between operation 1 and operation 2 in the PDG of Figure 4.5 is unconditional, since for every execution trace of the application, instruction 2 must always wait for the execution of instruction 1. Considering instead the data dependence between the instructions 14 and 15, again in Figure 4.5, it results that both the operations do not belong to every execution trace of the program. More in detail, the execution traces containing instruction 14 are a subset of those containing instruction 15. In other words, if a give execution trace contains 14, then it will contain also 15, but the converse is not true. In particular, when the condition associated to operation 13 is false, then 14 is not executed, while 15 must be executed regardless of the outcome of this condition. For this reason, the data dependence between 14 and 15 is conditional, and it depends on the evaluation outcome of the condition associated to 13. Thus, it is correct to say that operation 15 must wait for operation 14 only when 13's condition is false. The outcome of such condition is unknown at design time. Thus, the activating condition associated to operation 15 will be parametric. Every instruction in the specification needs to satisfy all its data dependences before to execute, since only after all the data predecessors terminate it will be able to elaborate the correct data. There is no need to consider other than direct data predecessors, since the predecessors will in turn manage their own data dependences in the same way.

4.1.4 Control Dependences Management

The set of control dependences in the EPDG and in the PDG are exactly the same set. As for data dependences, for every instruction in the specification, i.e. for every node in the graph, it is possible to consider only direct control arcs, since control predecessors will in turn manage their own direct control dependences. An instruction can have multiple control predecessors, as in case of $break$ constructs. Contrarily to data dependences management, every node does not have to wait for all its control predecessors. Indeed, when an instruction in control dependent on multiple instructions, for every single execution trace of the program there will be a single control predecessor which activates the execution of the current instruction. For this reason, as soon as one of the control predecessors terminates its execution, the current instruction can be safely executed.

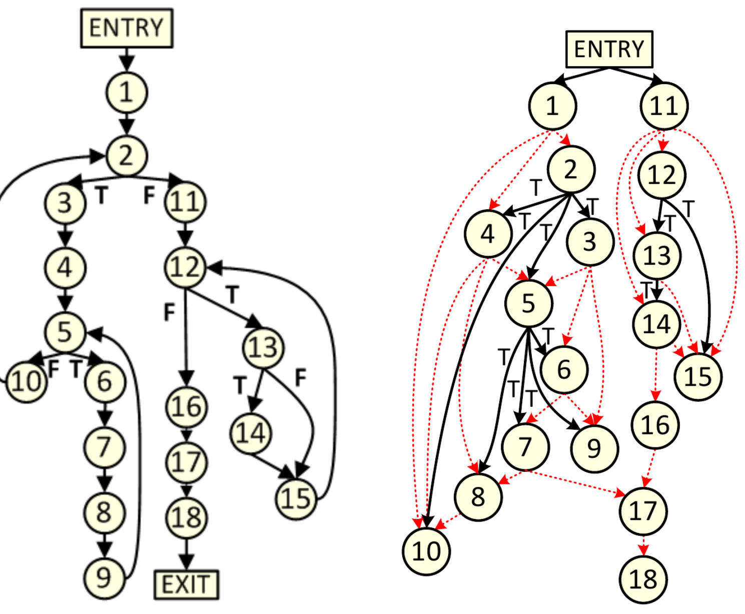
4.1.5 Control Flow Dependences Management

Control flow dependences define precedence relations among the instructions, according to the sequential control flow of the original program. For example, given two instructions i and j, j is control-flow dependent on i iff i is a conditional instruction, i.e. i alters the sequential control flow, and there exists a path from i to j in the acyclic CFG. Considering the example in the motivating example of Figure 4.1, the instruction 11 is control-flow dependent on the instruction 2. The corresponding CFG is proposed again in Figure 4.6a for sake of simplicity of exposition. Since this relation is not minimal, potentially useless synchronizations are included. To define control flow dependences and minimum control flow dependences, the following paragraph clarifies some notation issues.

Notation Given an instruction node i in the CFG, let the set $OE = \{(i, L) : L \in \{-, T, F, I\}\}$. The set OE is composed of couples, where the first element is a node i of the CFG, and the second element is the label of one of the outgoing arcs associated to i. Thus, the set OE implicitly represents the set of outgoing arcs for each node in the CFG. If i is a conditional instruction, L can represent the condition evaluation outcome, true 'T' or false 'F', associated with the branch (i, L), an integer number 'I' associated with a branch (i, L) of a switch construct or a don't care condition '$-$' due to a goto instruction. Otherwise, $L \equiv \{-\}$, since unconditional instructions have only unlabeled outgoing arcs in the CFG. For example, considering the CFG in Figure 4.6a, the set OE is:

$$OE = \{(1, -)(2, T)(2, F)(3, -)(4, -)(5, T)(5, F)(6, -)...\}$$

In the following the set of all the labels associated with a node i will be de-



(a) CFG (b) PDG

Figure 4.6: CFG (a), and PDG (b), for the specification in Figure 4.1.

noted as L^i. For example, for node 2 in the CFG in Figure 4.6a, it will be $L^2 = \{T, F\}$. For each arc $(i, L^i) \in OE$, let $R_{(i,L^i)}$ the set of nodes that are reachable in the acyclic CFG from i through the arc (i, L^i). Considering the acyclic CFG corresponding to the CFG in Figure 4.6a, it results:

$$R_{(1,-)} = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}$$

$$R_{(2,T)} = \{3, 4, 5, 6, 7, 8, 9, 10\}$$

$$R_{(2,F)} = \{11, 12, 13, 14, 15, 16, 17, 18\}$$

...

Let $R^{out}_{(i,L^i)} = \{v \in R_{(i,L^i)} : loop_v \supset loop_i\}$, $R^{in}_{(i,L^i)} = \{v \in R_{(i,L^i)} : loop_v \subset loop_i\}$, $R^{eq}_{(i,L^i)} = \{v \in R_{(i,L^i)} : loop_v \equiv loop_i\}$, and $R^{in}_{(i,L^i)} = \{v \in R_{(i,L^i)} : loop_v \not\subseteq loop_i \wedge loop_v \not\supseteq loop_i\}$. It results: $R_{(i,L^i)} = R^{out}_{(i,L^i)} \oplus R^{in}_{(i,L^i)} \oplus R^{eq}_{(i,L^i)} \oplus R^{in}_{(i,L^i)}$, that is $R^{out}_{(i,L^i)} \cap R^{in}_{(i,L^i)} \cap R^{eq}_{(i,L^i)} \cap R^{in}_{(i,L^i)} = \emptyset$ and $R^{out}_{(i,L^i)} \cup R^{in}_{(i,L^i)} \cup R^{eq}_{(i,L^i)} \cup R^{in}_{(i,L^i)} = R_{(i,L^i)}$. For instance, considering the pairs $(2, T)$ and $(2, F) \in OE$ obtained from the CFG in Figure 4.6a, since the instruction 2 belongs to $loop_1$ it results:

$$R^{out}_{(2,T)} = R^{in}_{(2,T)} = R^{in}_{(2,F)} = R^{eq}_{(2,F)} = \emptyset$$

$$R^{eq}_{(2,T)} = \{3, 4, 10\}; R^{in}_{(2,T)} = \{5, 6, 7, 8, 9\}$$

$$R^{out}_{(2,F)} = \{11, 16, 17, 18\}; R^{in}_{(2,F)} = \{12, 13, 14, 15\}$$

Let $R_{(i,\overline{L^i})}$ the set of nodes that are reachable from i in the acyclic CFG, but those passing through the arc (i, L^i). For node 2 in the acyclic CFG corresponding to the CFG in Figure 4.6a it results $R_{(2,\overline{T})} = R_{(2,F)}$ and vice versa. For arcs associated with switch constructs, $R_{(i,\overline{L^i})}$ is the union of the nodes reachable from the other arcs associated with the switch. Moreover, let DR_i the set of instructions reachable from i in the DDG and CR_i the set of instructions reachable from i in the CDG. For example, considering the PDG in Figure 4.6b, it results:

$$DR_1 = \{2, 4, 5, 8, 10\}; DR_2 = \emptyset; ...$$

$$CR_1 = \emptyset; CR_2 = \{3, 4, 5, 6, 7, 8, 9, 10\}; ...$$

Finally, for each pair of instructions i,$j \in V$, where $i \in loop_i$ and $j \in loop_j$, let $IL_{(i,j)}$ the set of the intermediate loops between the two instructions, that is $IL_{(i,j)} = \{l \in loops : loop_i \subseteq l \subset loop_j\}$. For example, referring to the CFG in Figure 4.6a, assuming, without loss of generality, that operations not belonging to any loop belong to a dummy loop $loop_0$, representing the root of the loop forest, it results:

$$IL_{(12,16)} = \{loop_3\}, IL_{(5,17)} = \{loop_1, loop_2\}, ...$$

Clearly if $loop_i$ is not nested into $loop_j$ then $IL_{(i,j)} = loop_i$. Notice that the three loops in the motivating example of Figure 4.1 have a single exit point: instruction 2 for $loop_1$, instruction 5 for $loop_2$ and instruction 12 for $loop_3$. However, the proposed methodology supports also loops with multiple exit points. The same definitions apply to the PDG, defining the sets $RPDG^{in}_{(i,L)}$ and $RPDG^{out}_{(i,L)}$, respectively.

Definition Given two instructions $i, j \in V$ in the acyclic CFG, if $j \in R^{out}_{(i,L)}$, with $L \in \{T, F\}$, then $j \in CR_i$.

In other words, the operations belonging to the set $R^{out}_{(i,L)}$ are control dependent on the operation i, under the condition L.

There exist two kinds of control-flow dependences: *Forward-Control-Flow Dependences* and *Backward-Control-Flow Dependences*.

Definition Given two instructions, $i, j \in V$ in the acyclic CFG, j is Forward-Control-Flow dependent on i iff $\exists\ L \in L^i : j \in R_{(i,L)}$.

The relation due to a Forward Control Flow Dependence (FCFD) preserves the sequential ordering among the instructions: the execution of j must follow the execution of i.

Definition Given two instructions, $i, j \in V$ in the acyclic CFG, iff $loop_i = loop_j \neq loop_0$, then i is Backward-Control-Flow dependent on j and vice versa.

The relation due to a Backward Control Flow Dependence (BCFD) indicates which instructions of the loop can be executed as first at the next loop iteration: once i has been executed at the $k\text{-}th$ iteration of the loop, the $(k+1)\text{-}th$ iteration can start with any of the instructions belonging to the same loop, including i itself. The effective execution order will be established by the other dependences.

Definition *A Control Flow Dependence is a Minimum Control Flow Dependence (MCFD) iff the synchronization among the involved instructions is needed to ensure correct execution.* The synchronization is needed in two cases. First, when the instruction source of the control-flow dependence must precede the instruction target, provided that both the instructions source and target must be executed. Second, when the incoming data and/or control dependences of the instruction target do not suffice alone to establish when it can be activated.

For example, the instruction 16 in the motivating example of Figure 4.1 must wait for the termination of $loop_1$. Such loop terminates when the condition associated to instruction 2 is false. Thus, to ensure correct execution, instruction 16 must wait for the condition associated to 2 to be false. This is true for every run of the program, since both 2 and 16 belong to every execution trace of the specification. Hence, the execution of the instruction 2 must precede the execution of 16. In this case both 2 and 16 must always be executed, since they belong to every feasible execution trace of the program. Moreover, supposing to modify the function prototype in "*int motiv(int a, int b, int c, int *out)*", and supposing to add an instruction "*19: return 0*", such new instruction would be neither data nor control dependent on any other instruction of the specification. However, the execution of 19 should follow the execution of all the other instructions, since it would cause the program termination. In this case the incoming data/control dependences of instruction 19 do not suffice to correctly establish when it can be executed.

Property *There exist three kinds of Minimum Control Flow Dependencies (MCFDs):*

i *Inter-loop forward MCFDs*

ii *Intra-loop forward MCFDs*

iii *Backward MCFDs*

Inter-loop forward MCFDs

Theorem 4.1.1 *Inter-loop forward MCFDs*
Given a loop-exit operation i, with the associated loop-exit condition $L \in L^i$, and a generic instruction $j \in \{R^{out}_{(i,L)} \cup R^{in}_{(i,L)}\}$, if $j \notin CR_i$, then an inter-loop forward MCFD with label $L \neq \{-\}$, $MCFD(i, j, L)$, occurs among i and j in the following two cases:

i there exists an instruction $z \in \{R^{in}_{(i,\overline{L})} \cup R^{eq}_{(i,\overline{L})}\}$ such that j is data dependent on z, and i belongs to the outermost loop among those in the set $IL_{(z,j)}$, that is $\forall l \in IL_{(z,j)} : loop_i \subseteq l$, with $i \in loop_i, loop_i \in IL_{(z,j)}$

ii j is a return statement and for all the dependences $MCFD(i, k, L)$ added due to the previous case, j is not reachable from k in the PDG.

Proof *Case i. The data dependence among z and j forces j to be executed after z, if z has to be executed. However, z belongs to a loop $loop_z \not\subseteq loop_j$, since $z \in \{R^{in}_{(i,\overline{L})} \cup R^{eq}_{(i,\overline{L})}\}$, while $j \in \{R^{out}_{(i,L)} \cup R^{in}_{(i,L)}\}$. Consequently, either $loop_z \supset loop_j$ or $loop_z$ and $loop_j$ are not nested. In both cases z must be executed multiple times before the proper value is produced for the correct execution of j. In other words j must wait for the last execution of z, i.e. for the execution of z at the last iteration of $loop_z$. Indeed, since $j \notin CR_i$, there is not a control dependences chain, starting from i, inducing j to satisfy the data*

dependence with z. Unfortunately, the termination of a loop cannot be always statically established, since the number of iterations can depend on unknown information at design-time. Thus, j must necessarily wait for the execution and evaluation of an exit point operation to be sure of the termination of $loop_z$ and of all the loops $loop_z$ is nested into and $loop_j$ is not, excluding $loop_j$ itself, i.e. of each loop belonging to $IL_{(z,j)}$.
Case ii. if a return statement j, following a loop in the control flow, is independent from the execution of that loop, then a synchronization must be added to prevent the termination of the program before the loop execution is finished, provided that the dependences added due to the previous case do not ensure such synchronization yet.

The dependences introduced due to the *Case i* needed to handle data dependences across loops. As an example consider the data dependence among the instructions 7 and 17 in the PDG of Figure 4.6b. The instruction 7 belongs to $loop_2$, which is nested into $loop_1$, while the instruction 17 does not belong to any loop. Thus, the instruction 17 must wait for the termination of $loop_1$ to read the correct value of the variable c. In other words, since $loop_1$ has a single exit point which depends on the condition associated to instruction 2, and more in detail the loop terminates when such condition is false, then instruction 17 must wait for $2F$. For this reason a minimum control-flow arc with label F must be added in the EPDG between 2 and 17. $IL_{(7,17)} = \{loop_1, loop_2\}$, where $loop_1$ is the outermost loop in $IL_{(7,17)}$, with exit operation 2 and exit condition F. Since $17 \in R^{out}_{(2,F)}$ and $7 \in R^{in}_{(2,\overline{F})} \equiv R^{in}_{(2,T)}$, then an inter-loop forward minimum control flow dependence occurs between 2 and 17 with label F. *Case ii*, instead, prevents the termination of the program before all the loops have been executed. Another issue addressed by inter-loop forward MCFDs is the prevention of incorrect early program termination. For example, supposing to modify instruction 17 in order to remove the data dependence between 7 and 17 in the PDG of Figure 4.6b, the instruction 18 would become independent from the execution of both $loop_1$ and $loop_2$. Hence, to force the program to wait for the complete execution of such loops before to terminate, an inter-loop forward minimum control flow dependence must be added. In this case, it is added between 2 and 18 with label F. However, considering again the PDG in Figure 4.6b, the correct termination of the program is already ensured, since a MCFD has been added between 2 and 17, and 18 is reachable from 17 in the PDG. Thus, in this case, the dependence between 2 and 18 is not needed.

Intra-loop forward MCFDs

Theorem 4.1.2 *Intra-loop forward MCFDs*
Given an instruction i, belonging to a loop $loop_i$, and a generic instruction

```
                int motiv_1(int a, int b, int c, int *out) {
                      int v1, v2, v3, v4, v5, v6, v7, v8;
                        int internal1, internal2;
                        v7 = v;
                        v2 = v7;
        LABEL1          internal1 = v2 & (1u);
                        if (internal1 == (0u)) {
                            v4 = v2 >> (1);
                            v1 = v4;
                            goto LABEL2;
                        } else {
                            if (u < v2) {
                                v5 = v2 - u;
                                v3 = v5;
                            } else
                                v3 = v2;
                            v6 = v3 >> (1);
                            if (v6 != (0u)) {
                                v1 = v6;
        LABEL2                  v2 = v1;
                                goto LABEL1;
                            } else {
                                v3 = v6;
                                v8 = (int) (v3);
                                internal2 = u << v8;
                                return internal2; }
                        }
                }
```

Figure 4.7: Motivating example for intra-loop MCFDs: GCD kernel

$j \in \{R^{eq}_{(i,L)} \cup R^{in}_{(i,L)}\}$, with $L \in L^i$, if $j \notin \{CR_i \cup DR_i\}$, then a intra-loop forward MCFD with label L occurs among i and j in the following cases:

i i is a conditional instruction and there exists an instruction $z \in \{(R^{eq}_{(i,\overline{L})} \cup R^{in}_{(i,\overline{L})}) \cap CR_i\}$ such that a loop-carried data dependence occurs among z and j. In this case a MCDF occurs also between i and all the instructions x such that $x \in \{R^{eq}_{(j,L^j)} \cap DR_j\}$

ii i is a generic instruction, j is a return statement and $DR_i = CR_i = \emptyset$

iii $\forall x \in loop_j : j \notin CR_x$, j is a loop exit and $\forall L \in L^i : \{CR_i \cap (R^{in}_{(i,L)} \cup R^{eq}_{(i,L)})\} = \{DR_i \cap (R^{in}_{(i,L)} \cup R^{eq}_{(i,L)})\} = \emptyset$

Proof *Case i. The instructions j and z belong to the loop $loop_i$, or to a loop nested into $loop_i$, since $j \in \{R^{eq}_{(i,L)} \cup R^{in}_{(i,L)}\}$ and $z \in \{(R^{eq}_{(i,\overline{L})} \cup R^{in}_{(i,\overline{L})}) \cap CR_i\}$.*

However, they are reachable from different branches, i.e. with different outcome of the condition associated with i. The loop-carried data dependence between z and j means that if z must be executed at a certain iteration k, then

the execution of z must precede the execution of j at the iteration $k + 1$. However, since $j \notin \{CR_i \cup DR_i\}$, there not exists a path from i to j in the PDG. This implies that without enforcing a dependence between i and j, j can be executed before i, i.e. before knowning whether the execution will follow the branch with label L or $\overline{\text{L}}$. In other words, the control-flow path between i and j in the CFG is recognized as useless synchronization. However, if the outcome of the condition indicates to take the branch with label L at least once, z must be execute before j, i.e. j must wait for the condition associated to the label $\overline{\text{L}}$. Thus, in this situation, the execution of j is anticipated before its loop-carried data dependence is satisfied.
Case ii. If $CR_i = DR_i = \emptyset$, then i has not successors in the PDG. This means that one of the parallel execution flows can terminate. For this reason i is connected with each reachable return statement.
Case iii. If j is a loop exit and it is not control-reachable from any instruction inside the loop, then its execution is dependent only on data dependences. It is the case, for example, of do-while loops. This can lead to overlap the execution of different iterations of the loop. If an hardware support for such situation is not present, loop pipelining can be prevented by adding an intra-loop MCFD between each instruction i such that i has not outgoing data/control dependences with target another instruction inside the same loop, and j.

This kind of dependences is needed to manage a subset of loop-carried data dependences in do-while loops, which require further synchronization. As an example, consider the code fragment in Figure 4.7, which shows the kernel of the Greatest Common Divisor (GCD) algorithm. Instruction 15 writes the variable $v2$, which is read by instruction 8 at the next loop iteration. Thus, a loop-carried data dependences occurs between instruction 15 and instruction 8. Hence, the execution of 15 at iteration i should precede the execution of 8 at iteration $i + 1$. However, these two instructions belong to different control regions, as shown in the corresponding PDG in Figure 4.8. Instruction 15 belongs to the control region controlled by instruction 4, while instruction 8 belongs to a control region controlled by some instruction outside the loop. The PDG, in this case, does not provide any condition to ensure correct execution. Moreover, since instruction 15 must execute regardless of the value assumed by the outcome of the conditions evaluation for instruction 4, the it results: $15 \in CR_4$. Contrarily, $8 \notin CR_4$, as shown in the corresponding PDG in Figure 4.8. Finally, 4, 8 and 15 belong to the same loop. In this case, the execution of 8 must be postponed until 4 is executed and the outcome of its associated condition is F. Hence, an Intra-Loop MCFD must be added between 4 and 8 with label F. In this way, for every loop iteration, if the condition associated to instruction 4 is true, then 15 is executed and the current iteration terminates without executing 8. *Case ii* considers all the instructions don't having outgoing dependences. They are connected to the return node to indi-

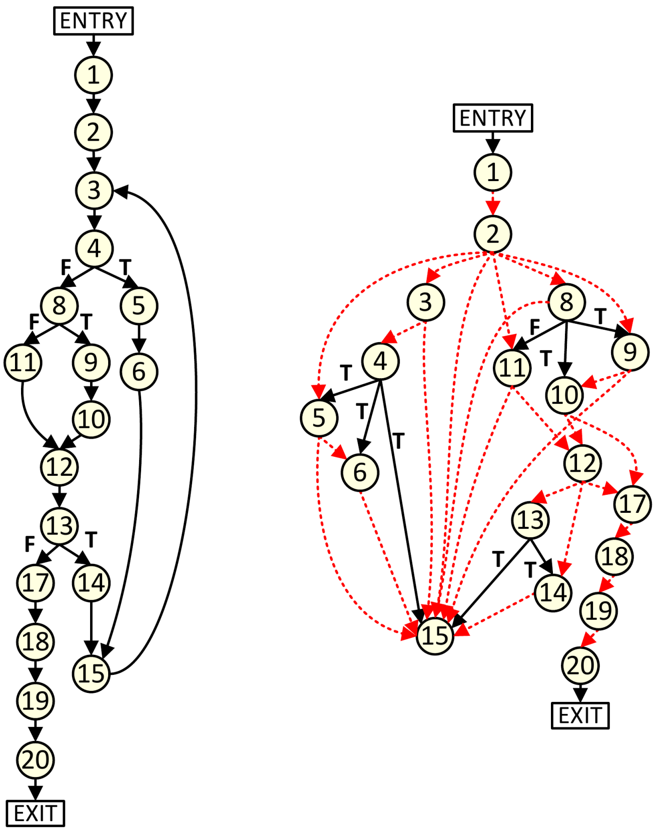


Figure 4.8: *CFG and PDG for the example code in Figure 4.7.*

cate that one of the parallel execution flows can finish. Currently, dependences are introduced due to *Case iii*. Further optimizations on this direction, such as architectural extensions for supporting loop pipelining, do not require this kind of dependences. When loop pipelining is enabled, the dependences due to *Case iii* are useless.

Backward MCFDs

Definition Given a loop l_i, let $l_i^{First} = \{x \in l_i, \forall y \in l_i, \forall L^y : x \neq y$ and $x \notin RPDG^{out}_{(y,L^y)}$ the set of instructions that can be executed as first among those of l_i. Let $l_i^{Last} = \{x \in l_i, \forall L^x : RPDG^{out}_{(x,L^x)} \cap l_i = \emptyset\}$ the set of instructions that must be executed as last among those of l_i.

The instructions belonging to l_i^{First} are those not having as predecessors instructions belonging to the same loop l_i in the PDG, while the instructions belonging to l_i^{Last} are those not having as successors instructions belonging to l_i in the PDG. In other words, the instructions belonging to l_i^{First} have not incoming data/control dependences with other instructions of l_i, and the instructions belonging to l_i^{Last} have not outgoing data/control dependences with other instructions of l_i.

Theorem 4.1.3 *Backward MCFDs*
A backward MCFD occurs among each instruction in l_i^{Last} and each instruction in l_i^{First}.

Proof *In absence of a pre-determined scheduling order, each instruction in l_i^{First} can be the first instruction that is executed inside the loop body, and similarly, each instruction in l_i^{Last} can be the last instruction that is executed inside the loop body. Hence to manage the loop restart, from the second iteration on, each instruction of l_i^{First} must wait for the execution of all the instructions in l_i^{Last}. Hence, a Backward MCFD occurs between each instruction in l_i^{Last} and each instruction in l_i^{First}.*

Backward MCFDs are needed to establish when loop iterations should start. Consider for example the instruction 2 in the PDG of Figure 4.6b. Since 2 is data-dependent on 1, the first time the loop is executed, instruction 2 can start after instruction 1 has been executed. Since instruction 2 is the entry point of $loop_1$, it is the first instruction to be executed at each iteration. Similarly, since none of the instructions in $loop_1$ is dependent on 10, such instruction belongs to the set of operations in the loop which can be executed as last instruction of the iteration. Hence a Backward MCFD must be added between 10 and 2, meaning that the next iteration of $loop_1$ can start from 2, after 10 has been executed at the previous iteration. Backward MCFD have been formalized as above described considering the absence of an hardware support for loop pipelining. If such support would be developed, the described formalization should be slightly modified taking into account loop- carried data dependences. In such way, the exploitable parallelism would increase, since the overlapping between different loop iterations would be allowed.

4.1.6 Activating Conditions

The instructions Activating Conditions (ACs) are logic formulas which express the instructions dependences. Every arc in the EPDG represents a dependence between source and target instructions. In other words, the execution of the target instructions is subject to some condition which depends on the source instruction. Thus, the arcs in the EPDG define a *conditional order relation* among the nodes, meaning that if the condition is satisfied, then the source and target operations must be sequentialized. Such condition represents a run-time event, such as the execution of an instruction, or the evaluation of a conditional instruction with a certain outcome. Considering the set V of instructions in

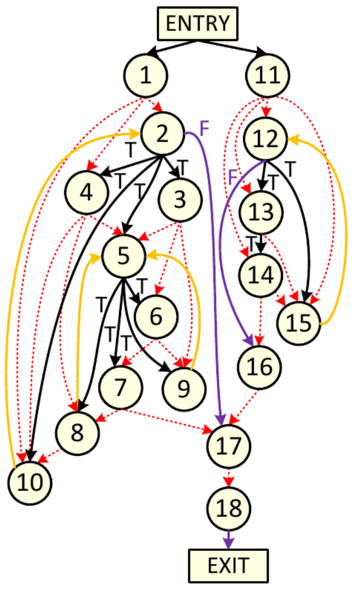


Figure 4.9: EPDG for the example in Figure 4.1.

the specification and the set E of dependences among them in the EPDG, the Activating Condition $AC(i)$ of an instruction $i \in V$ is a logic function of other instructions in V and of conditional instructions evaluation outcomes, i.e., of the subset of conditional (i.e., control and control flow) arcs $E_C \subseteq E$. The AC associated with each instruction in the specification is dynamically triggered at run-time, when its dependences are satisfied. When $AC(i)$ assumes a high value, the instruction i can safely start its execution. In the following the ACs are defined. Moreover, it is shown how to compute them through the analysis of the EPDG.

Operators For the ACs computation two operators must be defined: the *and* ($\wedge$) and the *or* ($\vee$) operators. The ACs are built by combining the instructions dependences though these two operators. The $\wedge$ and $\vee$ operators have a similar meaning with respect to the boolean operators of product and sum, respectively. Indeed, the $\wedge$ operator indicates that all the input conditions must be satisfied for the output to be true, while the $\vee$ operator indicates that at least one input condition must be satisfied for the output to be true. However, there exists a difference with respect to the boolean operators, due to their temporal meaning. The operators introduced here are considered with memory. Indeed, it may happen that the input are satisfied in different clock cycles. For sake of simplicity of exposition, Figure 4.9 proposes again the EPDG corresponding to the motivating example of Figure 4.1. For example, instruction 4 in the

EPDG of Figure 4.9 is control-dependent on 2. Thus, it must be executed after 2, and only if the condition associated to 2 has been evaluated as true. Moreover 4 is data-dependent on 1. Thus, it must wait for the data produced by 1. Hence, its activating condition must be a function of both 1 and $2T$, combined trough the $\wedge$ operator: $AC(4) = 1 \wedge 2T$. Since instruction 2 is dependent on instruction 1, the execution of 2 and 1 will never overlap. Thus, the inputs of $AC(4)$ will be satisfied in different control steps. The hardware implementing the $\wedge$ operator must store the information concerning the already satisfied inputs. It should collect the information about when each input becomes true, as described in [40]. Only when all the inputs are collected, the function is satisfied and the corresponding instruction is executed. This aspect must be carefully considered when introducing the formulation for the ACs. Moreover, inputs that become true in different control steps can cause a wrong double activation of the output when they are ORed. However, such situation will never happen, due to the proposed formulation. Indeed, the only conditions that are ORed correspond to the activation of different control paths, that cannot be simultaneously activated at a certain run of the program, since they are mutual exclusive. Thus, the $\vee$ operator can be simply implemented as a boolean sum. Another consideration about the proposed and operator is related to when the associated module must be reset. Since such operator has memory, it is needed to reset the corresponding module when the associated operation belongs to a loop. For every loop operation, each time a loop starts a new iteration, the associated and modules must be reset to ensure that the modules do not store values which refer to previous iterations. The and modules must be reset also when a loop terminates. This is needed, for example, when an instruction belonging to a loop depends on another instruction belonging to an inner loop.

ACs Formulation An AC is composed of three parts. The first one (AC_c) guarantees that *control dependences* are satisfied, the second one handles *data dependences* (AC_d), while the third one (AC_{cf}) manages the *control flow*. These three parts are combined through the *and* operator:

$$AC(i) = AC_c(i) \wedge AC_d(i) \wedge AC_{cf}(i) \tag{4.1}$$

One or more of these three parts can be missing. When none of the three parts is present, the instruction can start as soon as the program starts. In this case the instruction depends only on the pseudo-instruction *Entry*.

- **Control Dependences** For the AC expressing control dependences, it is possible to use the formulation described in [96]:

$$AC_c(i) = \bigvee_{c \in C} c \tag{4.2}$$

where C is the set of instructions which i is control dependent on. In other words, instruction i could be control dependent on multiple instructions, since it can be part of multiple execution traces of the program. However, each time i must be executed it is due to the activation of a single control path. Thus, the control activating conditions belonging to C can be simply ORed.

- **Data Dependences** For data dependences the formulation in [96] must be extended, since the proposed approach doesn't rely on a hierarchical model, that, as shown before, restricts the performance when targeting hardware-based dynamic scheduling. The proposed formulation for the ACs expressing data dependences is:

$$\begin{aligned} AC_d(i) = & \left[\bigwedge_{d \in D^{l_i}_{ineq}} d \vee \left(\bigvee_{j \in V} (j, L) \right) \right] \wedge \\ & \left[\bigwedge_{d \in D^{l_i}_{out}} d \vee \left(\bigvee_{j \in V} (j, L) \right) \vee be_{l_i} \right] \end{aligned} \tag{4.3}$$

where instruction i, belonging to the loop l_i, is data dependent on each instruction $d \in D = \{D^{l_i}_{ineq} \cup D^{l_i}_{out}\}$. More in detail, the set $D^{l_i}_{ineq}$ is the set of instructions belonging to loop l_i or to a loop that is nested into l_i, which instruction i is data dependent on. The set $D^{l_i}_{out}$ is the set of instructions not belonging neither to l_i nor to a loop that is nested into l_i, which instruction i is data dependent on. For each instruction d, the set of the (j, L) represents a branch; j is the instruction source of the branch and L is the label associated to the branch. Each branch (j, L) is such that d is reachable from j, and i is reachable from (j, L) without passing through d. More in detail, the term $\bigvee_{j \in V}(j, L)$ represents the negated control path for instruction d. When the control path is negated, instruction d should not be executed. The proposed formulation ensures that i will wait for the execution of d in any situation, except when d does not belong to an active execution trace in the current run of the program. Each instruction $d \in D^{l_i}_{ineq}$ either belongs to the loop l_i, which is the loop containing also instruction i, or belongs to a loop nested into l_i. Each instruction $d \in D^{l_i}_{out}$, instead, belongs to a loop, different from l_i, that must not be nested into l_i.

The first part of the expression, into the first pair of square brackets, is needed to handle data dependences among instructions belonging to the same loop, or to nested loops. For example, instructions 6 and 7 in the EPDG of Figure 4.10, both associated to $loop_2$, belong to this class. Another example is the pair of instructions 8 and 10 in Figure 4.10,

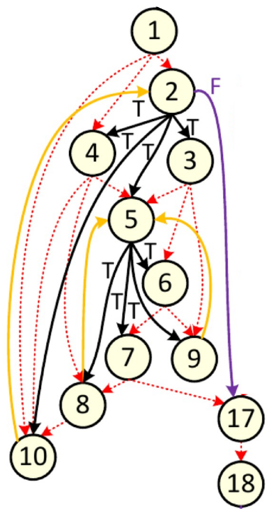


Figure 4.10: Extract from the EPDG for the example in Figure 4.1.

where 8 belongs to $loop_1$, and 10 to $loop_2$, which is nested into $loop_1$. In these cases, the instruction target of the dependence must wait for the execution of the instruction source if and only if there not exists a control path leading to the execution of the instruction target without passing through the execution of the instruction source. Alternatively, if such path exists, the instruction target must wait for the source if and only if such path has not been activated when the AC of the instruction target is going to be evaluated. For example, considering again instructions 7 and 8 in the EPDG in Figure 4.10, observe that such a path does not exists, since the two instructions are activated under the same control conditions. Hence it result: $AC_d(8) = 7$. As another example, consider

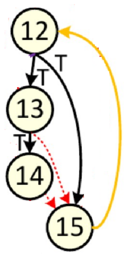


Figure 4.11: Extract from the EPDG for the example in Figure 4.1.

instructions 14 and 15 in Figure 4.11. It may happen that, a certain run of the program, the instruction 15 must be executed, even if the instruction 14 has not been executed. In fact, the execution of 14 is subject to the evaluation of the condition associated with instruction 13, while 15 must be executed regardless of it. Hence, it will result: $AC_d(15) = 14 \vee 13F$.

The second part of the expression, into the second pair of square brackets, is needed to handle data dependences among instructions that do not belong to the same loop, and such that the loop associated with the instruction target of the dependences is not nested into the loop associated with the instruction source of the dependence. For example instructions 3 and 5 in Figure 4.11 belong to this class. Instruction 3 belongs to $loop_1$ and instruction 5 belongs to $loop_2$, nested into $loop_1$. In this case, the execution of instruction 5 is subject to the execution of 3. However, 3 will be executed a single time for each iteration of $loop_1$, while 5 must be executed a number of times equal to the number of iterations of $loop_2$ for each iteration of $loop_1$. Thus, the execution of instruction 3 will be able to activate instruction 5 only for the first iteration of $loop_2$. The same consideration holds for the dependence between 4 and 5 in the same figure. For the subsequent iterations, the activation of 5 will be handled by the term be_{loop_2}. It results:

$$be_{l_i} = \bigwedge_{v \in last(l_i)} (v, L) \tag{4.4}$$

where, the set $last(l_i)$ contains those instructions whose successors do not belong to l_i, or, in other words, those instructions that can be executed as last operations of the loop. Due to this property, all the instructions in $last(l_i)$ will be source of a back-arc, with label L, having the loop entry point instruction as target. For example, considering $loop_3$ in Figure 4.11, it results: $last(loop_3) = \{8, 9\}$, and thus $be_{loop_3} = 8 \wedge 9$. As a consequence, the data activating condition for instruction 5 in Figure 4.11 is: $AC_d(5) = (3 \wedge 4) \vee (8 \wedge 9)$. Intuitively, it means that at the first iteration of $loop_3$, instruction 5 must be executed after both instructions 3 and 4 have been executed. From the second iteration on, the first part of the data AC is false, and thus the next iteration will start when the term $(8 \wedge 9)$ is satisfied.

- **Control Flow Dependences** The control flow dependences between loop exit instructions and instructions external to the loop must be distinguished from the other ones, since they are handled separately. The latter, in fact, are handled like data dependences belonging to $D^{l_i}_{ineq}$, while the former indicate that the target instruction must wait for the loop termination. However, a loop may have multiple exit points. For this reason they must be ORed. Let CF_{exit} be the set of control flow dependences coming from loop exits, and CF_{std} the remaining control flow dependences, and $CF = CF_{exit} \bigoplus CF_{std}$ (where $\bigoplus$ represents the direct sum operator). The dependences in CF_{exit} can be further divided according to the loop they come from. We will denote with $CF_{exit}^{l_i}$ the control flow dependences coming from the loop l_i. Moreover, considering the subset of nodes in the specification which are sources of control flow dependences, we indicate with $exits$ the set of loop exit nodes and with std the set of the remaining nodes. The formulation for the control-flow part of the Activating Conditions is:

$$
\begin{aligned}
AC_{cf}(i) = & \left[\bigwedge_{l_i \in loops} \left(\Big(\bigvee_{k \in exits} CF_k^{l_i} \Big) \vee \Big(\bigwedge_{j \in V} (j, L) \Big) \right) \right] \wedge \\
& \left[\bigwedge_{std} CF_{std} \vee \bigvee_{j \in V} (j, L) \right]
\end{aligned}
\tag{4.5}
$$

The first part of the formula, inside the first pair of square brackets, manages control flow dependences coming from loop exit operations. As above mentioned, such dependences must be ORed when they come from the same loop, since at each run of the program a single exit point for each loop will be taken. Moreover, the last product term has been added to handle the case in which i is not reachable from the the exit point of the involved loop. As an example, consider instructions 2 and 17 in Figure 4.11, related by a control flow dependence. In this case, the two instructions belong to different loops, and the dependence has been added to enforce instruction 17 to wait for the termination of $loop_1$, that contains instruction 2. Since $loop_1$ has a single exit point, and since there not exists a path in the EPDG leading to the execution of 17 without passing from 2 it will result: $AC_{cf}(17) = 2F$.

The second part of the formula, inside the second pair of square brackets, manages control flow dependences that do not come from loops. As above mentioned, they are treated as data dependences, since the path that activates the execution of the source of the dependence may not be activated at a certain run of the program. Figure 4.12 shows the complete set of Activating Conditions for the specification in the motivating example of Figure 4.1. The algorithm for the ACs computation works as follows. For each node in the EPDG, it simply takes its incoming arcs, representing the dependences whose it is subject to. Then it computes the ACs according to the described formulas.

4.1.7 Optimizations

This Section introduces one class of optimizations that can be applied during the design process. More in detail, this the first class consists in the elimination

NODE	AC(NODE)
1	*Entry*
2	$1 \vee 10$
3	$2T$
4	$2T \wedge 1$
5	$2T \wedge ((3 \wedge 4) \vee (8 \wedge 9))$
6	$5T \wedge 3$
7	$5T \wedge 6$
8	$5T \wedge 4 \wedge 7$
9	$5T \wedge 3 \wedge 6$
10	$2T \wedge 1 \wedge 4 \wedge 8$
11	*Entry*
12	$11 \vee 15$
13	$12T \wedge 11$
14	$13T \wedge 11$
15	$12T \wedge 11 \wedge 13 \wedge (14 \vee 13F)$
16	$12F$
17	$2F$
18	17
EXIT	18

Figure 4.12: *Activating Conditions for the example in Figure 4.1.*

of transitive data dependences, which is possible when certain conditions hold. Such optimization aims at reducing the size of the Activating Conditions, and consequently the area of the resulting circuit.

Transitive Data Dependences Reduction

As above mentioned, the set of data dependences defined in the PDG is not minimal when the graph is used to provide a parallel execution model for the specification. More in detail, there exists a subset of transitive data dependences which can be ignored for the purpose of computing the instructions activating conditions. A data dependence between an instruction i and another instruction j is transitive iff there exists a path in the Data Dependence Graph (DDG) from i to j, which traverses other nodes. In other words, in this case the DDG would contain a direct arc between i and j, which represents the transitive data dependences. Moreover, i and j would be also connected by means of another path. The concept of *Activating Path* is needed to explain the idea behind data dependences reduction: given an instruction a, the Activating Path AP_a is the set of control paths that lead to the execution of a. For example, if a belongs to the then branch of an if- then-else construct, a will be executed every time the condition associate to the if-then-else construct is true. Hence, calling b the conditional instruction associated to the if-then-else construct, it will result $AP_a = (b - True)$. Let us consider the transitive data dependences shown in Figure 4.13. In this case, the instruction b is data dependent on the instruction a, and the instruction c is data dependent on both a and b. The dependence between a and c is transitive, since there exists a path between a and c which traverse node b. The following defines when it is safe to remove the transitive dependence between a and c.

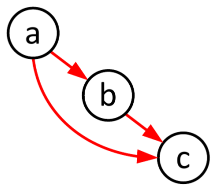


Figure 4.13: Transitive data dependence among the instructions a and c.

Theorem 4.1.4 *Given three instructions a,b and c, where b is data dependent on a, and c is data dependent on both a and b, the transitive data dependence between a and c can be removed iff at least one of the following conditions holds:*

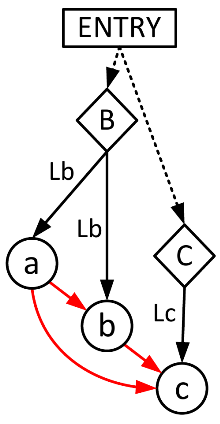
1. $AP_b \subseteq AP_a$

2. $AP_b \subseteq AP_c$

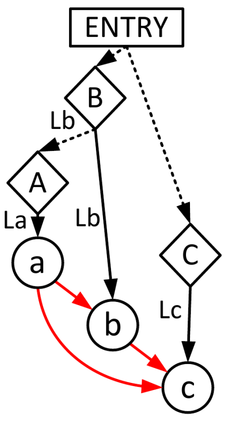
Proof In the following the two cases will be proved separately.

1. Let us examine the case when a and b have the same activating path: $AP_b = AP_a$, as shown in Figure 4.14a. Red arcs represent data dependences, while black arcs represent control dependences. Let use remove the data dependence between a and c, and let us suppose by contradiction that removing such dependence, c is allowed to execute before a, thus violating the data dependence. Since $AC(c) = b + B(!L_b)$, then either that c waits for b's execution or the condition associated to node B does not assume value L_b. If the condition L_b is false, then instruction a must not be executed (because $AP_a = L_b$). Thus, instruction c has not to wait for a's execution, and the original data dependence between a and c is satisfied even without the transitive arc. In other words, in this case the transitive arc is not needed to guarantee correct behavior. When, instead, the condition associate to node B assumes value L_b, then c must necessarily wait for b's execution.

   Instruction b must necessarily wait in turn for a's execution. Indeed, since a and b have the same Activation Path, it results: $AC(b) = a$. Hence, even this case, the hypothesis that c is allowed to execute before a is an absurd, and the arc between a and c can be safely removed.



(a) $AP_b \equiv AP_a$



(b) $AP_b \subset AP_a$

Figure 4.14: Case I: AP_b is equivalent to AP_a (a) or AP_b contains AP_a (b).

If $AP_b \subset AP_a$, as shown in Figure 4.14b, a and b are not activated under the same control conditions. Red arcs represent data dependences, while black arcs represent control dependences. Every time a is activated, b is activated too, while the converse is not always true. In other words, every time the condition associated to node A assumes value L_a, then both a and b will be executed. Moreover, if b is not activated, then also a will never be activated. In other words, if the condition L_b associated to node B does not assume value L_b, then both a and b will not be executed. Considering $AC(c) = b + B(!L_b)$, let us remove the data dependence between a and c, and let us suppose by contradiction that, by removing such dependence, c is allowed to execute before a. Also in this case, if the condition associated to node B does not assume value L_b, then c can correctly execute before a, since a does not belong to the current execution trace. Otherwise, c must necessarily wait for b, that in this case will be activated, since $AP_b = L_b$. Since b is activated, also a is activated. Thus b must wait in turn for a, which it is data dependent on. Hence, also in this case the hypothesis that c can execute before a is an absurd, and the transitive data dependence between a and c can be safely removed.

2. Let us consider the case $AP_b = AP_c$, as shown in Figure 4.15a. Let use remove the data dependence between a and c, and let us suppose by contradiction that, by removing such dependence, c is allowed to execute before a, thus violating the data dependence. Since b and c are activated under the same control conditions (i.e. since $AP_b = AP_c$), then it results $AC(c) = b$. Thus, c must necessarily wait for b's execution. The activat-

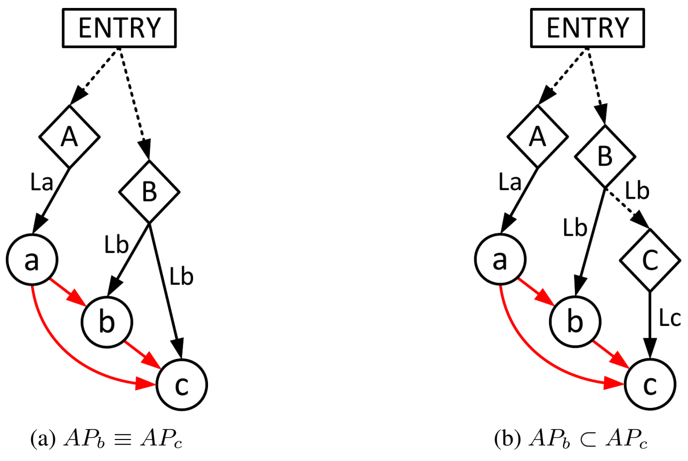


Figure 4.15: Case II: AP_b is equivalent to AP_c (a) or AP_b contains AP_c (b).

ing condition of instruction b is $AC(b) = a + A(!L_a)$. Thus, instruction b can be executed either if a is not in the current execution trace or if a has already been executed. In both cases the data dependence between a and c will be implicitly satisfied due to the data dependence between b and c. Hence in this case, the hypothesis that c can execute before a is an absurd, and the elimination of the arc between a and c is safe.

If $AP_b \subset AP_c$, b and c are not activated under the same control conditions, as shown in Figure 4.15b. However, every time c is activated, also b is activated. In other words, if the condition associated to node B assumes value L_b, then both b and c must be executed. The vice versa is not always true. Moreover, if b is not activated, then also a will never be activated. In other words, if the condition associated to node B does not assume value L_b, then both b and c will not be executed. Since it results $AC(c) = b$, also in this case the hypothesis that c can execute before a is an absurd, and the transitive arc between a and c can be safely removed.

4.1.8 High-Level Synthesis Flow with Adaptivity Analysis Support

The proposed dynamic scheduling methodology for automating the design of flexible hardware cores relies on the computation of the Activating Conditions for the instructions in the specification. The ACs express the instructions dependences in form of logic formula, thus embedding uncertain information which will be resolved at run-time. The proposed automatic approach starts from a C-like behavioral specification. Supporting instructions auto-scheduling allows to overcome restrictions due to unknown or uncertain information at design time, with a limited area overhead, while simultaneously increasing parallelism exploitation. For this reason, the proposed approach aims to formalize the Instructions *Activating Conditions* in order to automate their computation, for the integration into a synthesis flow. With respect to [96], the ACs are considered at a finer granularity level, i.e. at instruction level. The first step to identify the ACs consists in the definition of a proper IR able to show the dependences, while exposing the inherent parallelism and simultaneously providing a parallel execution model. The PDG can serve as starting point for the purpose, since it does not include control flow arcs causing potentially unnecessary sequencing. However, as previously mentioned, it does not contain control flow arcs that are instead necessary to guarantee correct execution. Thus, the PDG is extended by adding the required control flow arcs, leading to the definition of a novel IR called *Extended Program Dependence Graph (EPDG)*. The EPDG is then analyzed to compute the Activating Conditions, according to a set of formal rules. Figure 4.16 shows how the proposed methodology can be integrated into a typical high-level synthesis (HLS) flow, automating the design process of self-adaptive cores by providing the support for hardware-based auto-scheduling. More in detail, the task for the EPDG construction must be added in the front-end of the HLS flow. Then, the task of the ACs computation is added to the typical HLS tasks.

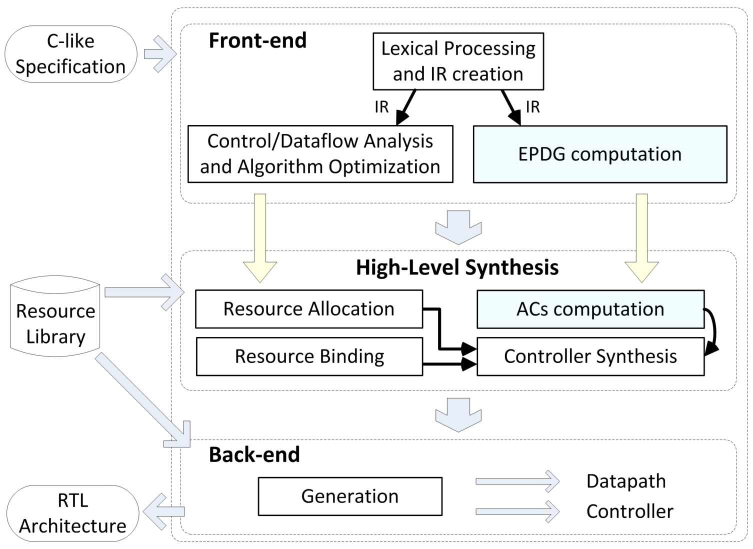


Figure 4.16: Typical High-Level Synthesis flow extended with dynamic AC-scheduling support.

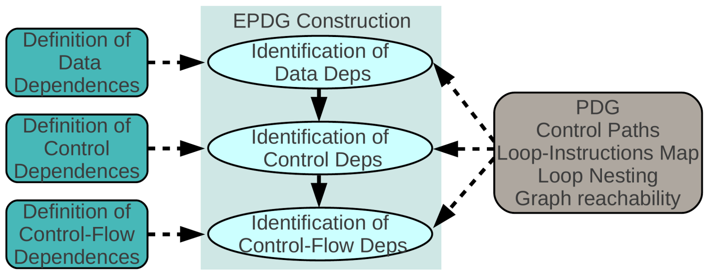

EPDG Construction
Definition of Data Dependences
Definition of Control Dependences
Definition of Control-Flow Dependences
Identification of Data Deps
Identification of Control Deps
Identification of Control-Flow Deps
PDG
Control Paths
Loop-Instructions Map
Loop Nesting
Graph reachability

Figure 4.17: EPDG construction flow

EPDG Construction The algorithm for the EPDG construction works as follows. For each node in the PDG, it creates an equivalent node in the EPDG. Then it starts visiting the arcs of the PDG. When a data dependence arc is encountered in the PDG, an equivalent data dependence arc is created in the EPDG. Similarly, when a control dependence arc is encountered in the PDG, an equivalent control dependence arc is created in the EPDG. Indeed, data and control dependences are minimum in the PDG. Moreover, the algorithm pre-computes the information needed to manage instructions activating conditions (e.g., control path for the instructions, operations belonging to each loop, loop nesting structure, reachability in the PDG, etc.). When source and target of the data dependence belong to different loops, where the target loop is not nested into the source loop, the arc is marked as special, to manage inter-loop data activating conditions, as will be shown later in this Chapter. As, for definition of Minimum Control Flow Dependences, the addition of a control flow dependence either is subject to these information or depends on data/control dependences, control flow arcs in the EPDG are added after the examination of data and control dependence arcs in the PDG. This mechanism is shown in Figure 4.17.

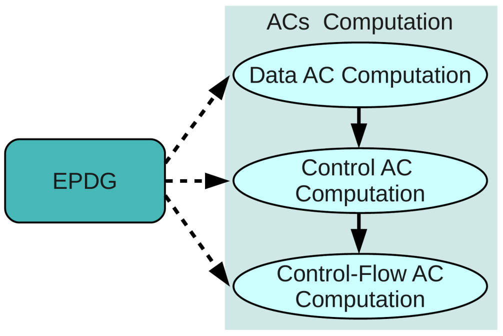


Figure 4.18: Activating Conditions computation flow

ACs Computation Figure 4.18 illustrates the automatic flow for the ACs computation. The Activating Conditions are composed of three parts, which manage data, control and control-flow dependences, respectively. Given the definition of each of these parts, the algorithm visits the nodes of the EPDG and recursively applies the definition to every node.

4.2 Hardware Cores with Support for Adaptive Execution

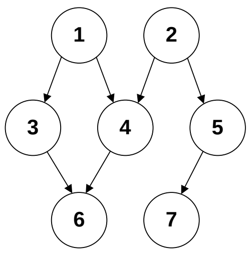
This Section presents a novel architectural model for adaptive embedded systems. The proposed hardware core embeds a standard datapath, as in conventional embedded systems, while it substitutes the centralized FSM with a lightweight token-based innovative controller, which is able to automatically support dynamic scheduling. Such controller allows run-time reordering of the instructions according to the information provided by the Activating Conditions (ACs). The proposed hardware core is able to adapt its behavior according to run-time conditions, thus managing different sources of unknown, uncertain and unpredictable information. The set of dependences of each instruction (or group of instructions, according to the desired granularity level) is expressed as a logic formula (i.e., the AC) and implemented as part of the controller. At run-time (i.e., when unknown information is resolved), the controller is able to perform dynamic scheduling by activating each instruction as soon as its dependences are satisfied (i.e., when the AC is satisfied) and the needed resource is available. The components within the system interact with each other and with external modules by means of a simple token-based communication schema, guided by the hardware implementing the ACs. Moreover, the same token-based schema is used to manage resources availability. Thus, such approach does not incur neither in the area issues nor in communication and synchronization overhead, which are typical of the FSM approach and of many other traditional approaches. The proposed controller can be easily built starting from the behavioral specification. Furthermore, it natively supports units with variable latency, without needing the creation of a complex FSM architecture. Adaptive hardware cores seems to promisingly handle unknown, uncertain and unpredictable information.

4.2.1 Motivating Example for Adaptive Controller

This Section introduces an illustrative example to motivate the proposed architecture. Let us assume that the desired hardware core has to implement the simple functionality shown in Figure 4.19a. The corresponding Data Dependence Graph (DDG) is shown in Figure 4.19b. The nodes in the DDG represent the operations in the original specification. The arcs in the DDG represent data

1: $a = y - x$
2: $b = x/y$
3: $c1 = a * z$
4: $c2 = a * b$
5: $c3 = b * z$
6: $d1 = c1 + c2$
7: $d2 = c3 + z$

(a) Fragment of code



(b) Data Dependence Graph (DDG)

Figure 4.19: An illustrative example to motivate the proposed architecture.

dependences to be satisfied for the execution of each instruction: each destination node elaborates the data produced by the source nodes associated to its incoming arcs. Since this particular specification presents only data flow, i.e. it does not contain conditional instructions which modify the control flow, the corresponding parallel execution model in this case can be obtained by the only analysis of the data dependences. In other words, the instructions can be reordered and executed in parallel as long as the parallel schedule does not violate the constraints expressed in the DDG. For this reason, the necessary and sufficient condition to safely execute each destination node is that all its predecessors in the DDG have terminated their execution. For example, the operation 4 can start only when the operations 1 and 2 are completed. Considering only the direct predecessors is sufficient because this criteria is recursively applied to all the instructions in the program, starting from the entry point. For example, it is safe to say that instruction 6 can execute after the termination of the instructions 3 and 4, since these instructions will in turn be executed only after the execution of the instructions 1 and 2 completes. Traditional scheduling techniques handle this situation by defining a partial ordering relation among the operations which share the same functional unit. Such ordering is fixed at design time and it cannot change during the execution. This can cause further decrease in performance. For example, if the subtracter's latency is lower than the divider's latency, then the operation 3 can execute as soon as the operation 1 terminates. However, assuming that a partial relation order $4 < 3$ has been defined and fixed at designed time, the execution of 3 is constrained to the termination of 4 in vain. For this reason, this aspect must be carefully managed.

Let us also assume that only one instance for each Functional Unit (FU) is available in the datapath, potentially with variable latency. Therefore, the controller has to manage the concurrency when different instructions, bound to the same unit, are simultaneously ready for execution. For example, if the operations 1 and 2 have the same latency, then the operations 3, 4 and 5 compete for the use of the unique multiplier of the system. If the latency of operation 1 is lower than the latency of operation 2, then instructions 4 and 5 will be ready at the same clock cycle, thus actually requiring the multiplier to manage the concurrency between them. Each FU can require a different number of cycles to perform the computation. This latency is not always known during the synthesis of the controller. It can also change during the execution, since, for example, different execution modes (e.g., varying the execution frequency) can be selected by the monitor to reduce power consumption or to handle overheating issues. When the execution latencies are known at design time, existing methodologies are able to guarantee correct execution, provided that such latencies do not change unexpectedly during the life of the hardware module. Figure 4.20 illustrates this concept by means of exemplifying schedules. When, for example, resource A (i.e., the subtracter) has a nominal latency of 2 clock cycles, operation 3 is normally scheduled to start at clock cycle 2, as shown in Figure 4.20a. However, unpredictable situations can happen during the execution. For example, if at run-time the same FU takes 3 cycles, operation 2 will start before the data has been correctly produced, leading to

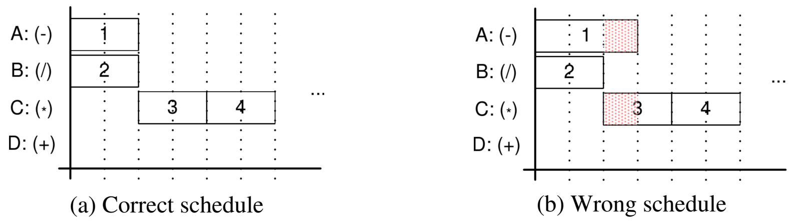


Figure 4.20: Example of correct (a) and wrong (b) execution for the motivating example code in Figure 4.19

an execution error, as shown in Figure 4.20b. For this reason, the designers often adopt an additional margin to reduce the error probability, even if this leads to suboptimal results. When different execution latencies are expected at design time, designers have often to rely on the worst case, again potentially leading to suboptimal results. However, even when an upper bound is known, the correct execution cannot be guaranteed for the entire life of the device, again due to degradation. Unfortunately, in various application fields, this is a strict constraint. Consider, for example, reactive systems in which some computation is activated only if a certain event occurs, leading to undefined response time. In these cases, no assumption can be done on the execution latency, thus it is not possible to schedule the corresponding operation in a given control step. Probabilistic assumptions can be adopted, but, again, this does not guarantee correct results in all the cases. For these reasons, any controller that has to interact with resources potentially having variable latency has to implement a mechanism to synchronize the execution. The common approach is to build adaptive systems through a communication paradigm adjusting the behavior at run-time. However, existing approaches are not suitable in many cases, since they need to predict all possible schedules at design time to implement the corresponding state machine. This process can be tedious and error-prone, easily leading to an explosion of the number of states even for a reduced number of possible run-time variations in the schedule. Finally, it is worth notice that, if all the functional unit of the example have the same fixed latency, then the corresponding State Transition Graph has 5 states and 4 arcs denoting transitions. In order to support uncertain execution and avoid delays due to a pre-computed schedule, it is needed a State Transition Graph with 15 states and 38 transitions. Moreover, the explosion of the number of the states does not depend only on the number of nodes of the specification, but also with the number of possible schedules. Thus, for more complex specifications with respect to the one chosen as example, the FSM approach will lead to an unfeasible complexity.

The main objective of the proposed architecture is to provide support for adaptive behaviors in modern embedded systems by addressing all the discussed issues. The proposed controller adopts a very simple communication paradigm to determine when operations complete their execution. It takes decisions based on the information related to the instructions dependences (i.e. the ACs). This allows to tolerate and to handle many sources of uncertainty, such as performance degradation or modules with unbounded latency, while keeping complexity and communication overhead under control. At the same time, this approach provides a unified vision in embedded systems design, which allows to handle the modules in the architecture uniformly, regardless of their functionality and regardless of if they are integrated or external. The components can be viewed as black boxes with respect to the implemented specification.

4.2.2 Proposed Controller Arhitecture for Adaptive Hardware Cores

We propose an adaptive controller to implement hardware cores able to adjust their behavior according to run-time behaviors, which are possibly unpredictable at design time. Such architecture ensures correct execution of the implemented specification under uncertain execution. The proposed target architecture for the controller is composed of a set of modules interacting according to a very simple token-based paradigm, to avoid communication overhead. The finite state machine construction is not needed, since the activation mechanism is explicitly managed by computing the instructions dependences. Since the controller can be obtained through a bottom-up approach as a com-position of the modules, the problems related to decomposition techniques are avoided. The key idea at the basis of this novel architectural model is based on the observation that the execution of each instruction is subject to a set of *Activating Conditions (ACs)*, which represent the dependences among the instructions. Such ACs define a parametric precedence relation over the set of the instructions, depending on parameters whose value will be known only at run-time. Due to their nature, ACs are easy to encode into logical functions. The semantic behind such logic formulas is that when the dependences associated to an instruction are satisfied, then the corresponding AC assumes true value and the current instruction can execute. The resulting communication protocol is thus a simple token-based mechanism implementing the producer-consumer paradigm. Given an arc e_{ij} among the source node p_i and the target node c_j in the behavioral specification, e_{ij} represents the following constraint: the producer p_i must notify to the consumer c_j when some condition associated to p_i is satisfied. Sometimes this condition is simply the termination of p_i's execution. In other cases it can be, for example, the evaluation outcome of the condition associated to p_i, when p_i is a conditional instruction. Instruction p_i notifies c_j that the condition is satisfied by means of a *token*, implemented through a high value on a wire. When the consumer receives all the tokens associated to its predecessors, then its *activating conditions* is satisfied and its execution can safely start. This communication mechanism ensures the correctness also under uncertain execution. In fact, the operations will start only when all predecessors have effectively completed their execution. Note that, if an operation takes more than expected, its successors will wait for the corresponding token, since their activating condition will be not satisfied. On the other hand, if the operation takes less than expected, the successors can be activated earlier, potentially improving the performance of the design. For this reason, the controller can be considered *adaptive* since it adjusts its behavior at run-time, based on the interactions with the resources.

The proposed controller architecture is composed of a set of interacting modules: the *Execution Managers (EMs)* and the *Resource Managers (RMs)*. An EM is instantiated for each operation in the specification. The Execution Manager is responsible for managing the AC associated to the instruction. It collects all the activation signals associated with the corresponding predecessors, it elaborates the activation conditions by means of the associated logical function and it determines when the operation can be effectively executed. However, due to resource sharing, multiple operations, bound to the same unit, can be simultaneously ready for execution (i.e., their activating conditions are satisfied). For this reason, a Resource Manager is instantiated for each functional unit. The RM encodes a priority list (defined at design time) to determine which operation has to be executed in case of concurrency. Figure 4.21 shows the resulting architecture for the example shown in Figure 4.19. Given an operation, the corresponding EM is connected to the RM of the unit where

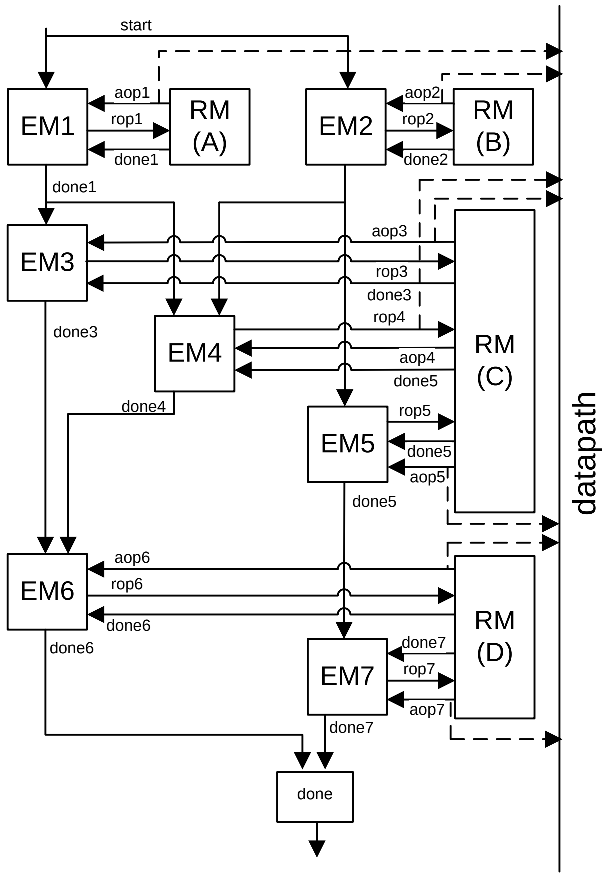


Figure 4.21: Proposed architecture for the motivational example in Figure 4.19.

the operation is bound. Note that each EM receives the signals from the EMs of the operations that activate its associated operation execution. For example, operation 3 can be executed only when operation 1 has been completed. When 3 can start, the related $EM3$ sends a request signal $rop3$ to the corresponding $RM(C)$, where C is the multiplier needed by 3. Then, at each clock cycles, if the unit is not executing any other operation, $RM(C)$ determines which instruction can be executed, based on the request signals and on the priority list. Since concurrency takes into account the request signals, it avoids to sequentialize operations only because of the priority list. Only the operations that are simultaneously ready for execution are ordered according to their priorities. When the resource C becomes available and there are no instructions with a higher priority, $RM(C)$ allows the execution of 3 by means of the signal $aop3$. This signal activates the proper selectors in the datapath, ensuring the correct execution of the operation 3. Moreover, $RM(C)$ notifies the operation ter-

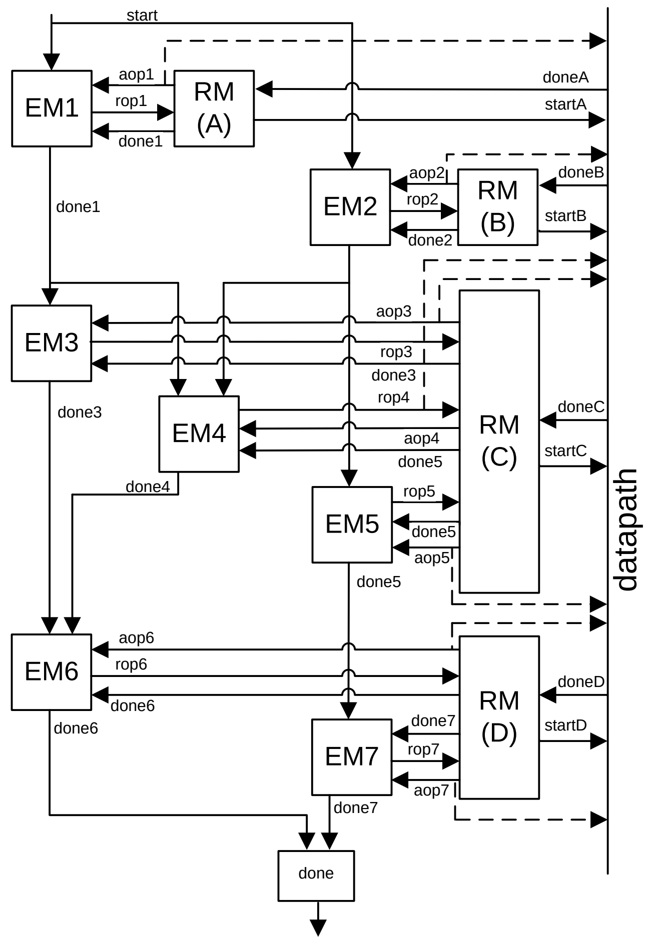


Figure 4.22: Extended version of the proposed architecture for the motivational example.

mination to $EM3$ and to all the EMs of the operations depending on 3 (i.e., operation 6) by means of a done signal $done3$.

When the functional units have uncertain execution latency, the designer must provide explicit communication between the controller and the functional units themselves to establish exactly when the computation has been completed, and thus to correctly activate the dependent instructions. Therefore, when the activating condition is satisfied, the EM sends a signal to the corresponding functional unit to start the execution of the operation. When the operation has been performed, the functional unit notifies the RM and the EM through a done signal, that is forwarded, as in the previous case, to the EMs of the dependent instructions. The resulting architecture implementing the example shown in Figure 4.19, where all functional units have been considered with variable latency, is shown in Figure 4.22. Note that the implementation of function calls (i.e., operations implemented by complex submodules with start/done signals) having variable latency is naturally supported by this architecture. In conclusion, this communication scheme ensures execution correctness even under unpredictable response time of the involved functional units. Thus, the proposed architectural model is able to guarantee a totally adaptive behavior at run-time.

4.3 Experimental Evaluation

This Section discusses some experimental results obtained to validate the proposed approach. More in detail, Section 4.3.1 describes and characterizes the set of benchmarks used for the validation of the automatic synthesis of adaptive embedded systems. Section 4.3.2 shows the capabilities of the proposed adaptive controller by implementing the example in Figure 4.19 in Verilog. Finally, in Section 4.3.3, the HLS results obtained by integrating the dynamic AC-scheduling technique in the PandA design framework, to automatically generate adaptive hardware cores, are discussed.

4.3.1 Experimental Setup

The proposed methodology for the dynamic AC-scheduling has been implemented in C++ and integrated into the PandA framework [19]. The design has been time-constrained at a clock period equal to 10ns.

The benchmarks used for the experimental evaluation are:

- *adpcm*, *gsm*, from the CHStone test suite [75]
- *bubble_sort*, *fastWalshTransform*, *popCnt*, *yuv2rgb*, from [6]
- *convolutionSeparable*, from [9]
- *wavesched*, from [67]

4.3.2 Experimental Results: adaptive hardware cores

To validate the proposed controller, both the architectures, in Figure 4.21 and 4.22 respectively, implementing the example in Figure 4.19, have been implemented in Verilog. Then the behavior has been simulated with Xilinx ISim ver. 12.3 [28]. As described in the previous sections, the design has been implemented with a single instance for each functional unit type. The priority values for the operations are shown in Table 4.1. When multiple operations assigned to the same unit are ready, the one with the higher priority will be executed. Then, a second experiment has been performed to analyze how the controller

Table 4.1: Priority values for the operations of the motivational example.

Functional Unit	Operation	Priority Value
A: (−)	1	7
B: (/)	2	6
C: (∗)	3	3
	4	4
	5	5
D: (+)	6	2
	7	1

adapts its execution at runtime. More in detail, each functional unit has been modified to simulate a certain delay before producing the done signal and thus simulating uncertain execution.

The first experiment simulated the execution with a fixed latency, with and without the support for units with variable latency. All functional units of the architecture in Figure 4.21 and 4.22 have been set to complete the execution in one clock cycle. In both the cases, the specification required 5 clock cycles to be executed, i.e. the same obtained through the As Soon As Possible (ASAP) scheduling under resource constraints. Moreover, this result also shows that the additional explicit communication between the controller and the data-path modules, introduced to support units with variable latency, does not affect the performance. In this case, the corresponding FSM has 5 states and 4 arcs denoting transitions. These results confirm that the simple token-based communication scheme does not introduce any overhead in terms of clock cycles. Considering the area occupation, the proposed controller requires 9 Flip-Flops (FFs): 2 FFs in the AM of operation 3 and one FF for each EM. On the other hand, an equivalent implementation based on a standard FSM requires 5 FFs assuming the use of the one-hot encoding, typically used in critical environments [44].

The second experiment assigned variable delays to each functional unit, based on distribution of probability. Table 4.2 reports the assumed probability

Table 4.2: Assumed run-time execution delay probability distribution for each functional unit.

Functional Unit	1cc	2cc	3cc	4cc
A: (−)	15 %	80 %	5 %	0 %
B: (/)	5 %	70 %	20 %	5 %
C: (∗)	8 %	80 %	11 %	1 %
D: (+)	15 %	80 %	5 %	0 %

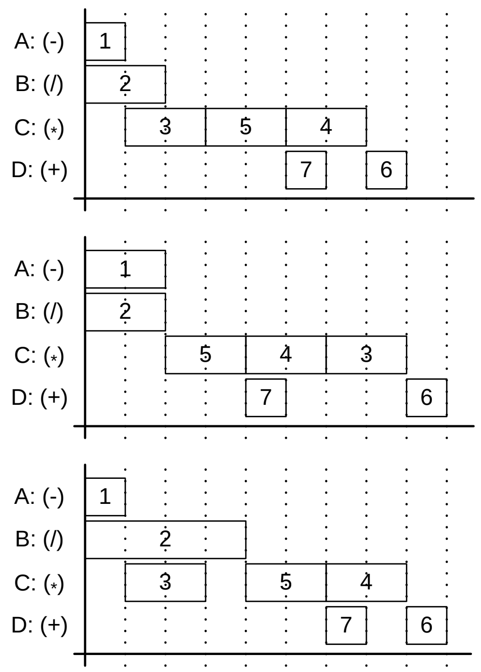


Figure 4.23: Some scheduling results for the proposed example.

distribution for run-time execution delays, in terms of clock cycles, for each FU. These values can represent the deviations from the nominal ones in case of process variation or effects due to degradation. In other cases, such behavior can be obtained when the data-path changes execution mode (e.g., the implementation) for the functional unit at run-time. If the units represent data-dependent submodules (e.g., function calls), these values could represent the variable latency of execution. All combinations of delays have been simulated and the run-time behavior of the controller has been analyzed. The results, shown in Figure 4.23, demonstrate that the proposed controller is able to reorder the operations on the basis of run-time conditions, always producing correct results while reducing the overall execution time when inherent parallelism is present. The FSM can guarantee correct results considering worst-case delays for each functional unit. In this case, the complete execution requires 18 clock cycles, corresponding to a 360% increase of execution time with respect to the best case. However, this cannot support unexpected situations at runtime. On the other hand, the FSM is able to reach the results of the proposed controller by implementing the request/acknowledge paradigm and determining the transitions among all admissible combinations of scheduling. However, the number of states increases with the number of possible scheduling: for the proposed example, the FSM has 22 states and 62 transitions, thus requiring 22 FFs with the one-hot encoding and significant logic for implementing the transition functions, showing that the approach for FSM can become impracticable even for small designs. On the contrary, the proposed architecture still needs just 9 FFs. As previously mentioned, to support variability with conventional hardware cores, all the possible scenarios must be considered in advance. This leads to a significant area overhead, up to the Cartesian product of the number of states in the FSM. Moreover, conventional cores do not support all the sources of uncertainty. Flexible hardware cores overcome these limitations. However, since a methodology for their design automation has not been provided in literature, they are currently difficult to implement and maintain by hand. Thus, the architectural model described in [40] has been implemented, and the proposed dynamic scheduling technique has been integrated in the design process of such cores.

In conclusion, the proposed architecture is able to adaptively schedule the operations, performing run-time adjustments that ensure correct results in all the cases. Moreover, the complexity of the proposed architecture is linear with the number of operations and functional units, very simple compared to the FSM that grows exponentially with the number of possible run-time situations. Moreover, the proposed architecture is able to deal also with conditions that were unexpected at design time, showing good performance and always producing correct results.

4.3.3 Experimental Results: dynamic AC-scheduling

To validate the dynamic AC-scheduling technique, the proposed approach has been compared with the most common approach in hardware synthesis, consisting of a list-based algorithm for the scheduling and targeting a centralized FSM controller based architecture. This choice is motivated by the presence of numerous approaches in literature for the high-level synthesis. The comparison with the most common approach should simplify other possible comparisons to the reader.

Table 4.3: Simulation: Clock Cycles

Benchmark	**list FSM**	**ACSched Flexible**	**Gain**
adpcm	76113	69571	8.59%
bubble_sort	731	354	51.57%
convolutionSeparable	47187	42839	9.21%
fastWalshTransform	33	11	66.67%
gsm	5804	2983	48.60%
popCnt	397	292	26.45%
wavesched	73	39	46.57%
yuv2rgb	168	139	17.26%
Average Gain			***34.36%***

Table 4.3 shows the simulation results in terms of clock cycles needed to executed the synthesized application. Since the architectural model of flexible controllers is token-based and does not lead to communication overhead, the reported clock cycles indicate the time needed to effectively perform the operations. Stalls due to synchronization/communication are not required with this architectural model. The gain obtained is strictly related to the amount of parallelism available in the specification. Data-flow intensive parallelism is well exploited also by traditional techniques, as shown in the results for the benchmark yuv2rgb. At the opposite, control-flow intensive specifications, as wavesched, show a significant gain in performance with the proposed approach. More in detail, yuv2rgb contains a single loop, while wavesched contains two parallel loops, where the first one contains an if-then-else construct. Hence, wavesched is a typical example showing how the list-FSM approach, as other standard approaches, is limited by the presence of control constructs, especially if they are nested into each other. The results obtained with standard techniques represent the lower bound of the performance provided proposed approach. It is useful to remark that conventional approaches, which the proposed approach is being comparing to, cannot guarantee correct results at all in presence of unknown information.

Table 4.4: Synthesis: Area

	list-FSM			**ACSched-Flexible**			**Gain**		
Benchmark	**FF**	**LUT**	**PAIRS**	**FF**	**LUT**	**PAIRS**	**FF**	**LUT**	**PAIRS**
adpcm	5122	22900	22900	18586	20225	25055	-262.86%	11.68%	-9.41%
bubble_sort	153	935	937	365	741	910	-138.56%	20.75%	2.88%
convolutionSeparable	1359	4432	4471	3571	3945	5497	-162.77%	10.99%	-22.95%
fastWalshTransform	681	2345	2395	1289	2267	2721	-89.28%	3.33%	-13.61%
gsm	5915	19188	19261	16503	15516	21673	-179.00%	19.14%	-12.52%
popCnt	390	1581	1619	618	1513	1769	-58.46%	4.30%	-9.26%
wavesched	182	474	480	613	572	786	-236.81%	-20.67%	-63.75%
yuv2rgb	428	2047	2056	851	1171	1697	-98.83%	42.79%	17.46%
Average Gain							*-153.32%*	*11.54%*	***-13.89%***

Table 4.4 shows the results of the RTL synthesis on a Xilinx Virtex 5 FPGA. Even if the number of Flip-Flops increases, this number is still significantly smaller than the number of LUTs, which is instead reduced by the proposed approach for most of the presented benchmarks. In FPGAs, when the number of FFs and LUTs is unbalanced, an increase of the smaller is hidden by the bigger, since the resources are allocated by cells. This result is well shown by the numbers of LUT-FF pairs, which, for the proposed approach, are reasonably close to those obtained with the list-FSM approach. More in detail, the proposed technique leads to an average increase of less than 14% of the total area. Moreover, at the state-of-the-art, the literature does not provide efficient register allocation and binding techniques for the considered architectural model. All the existing techniques restrict parallelism exploitation if coupled

with a flexible architectural model. Thus, a unique binding approach has been considered, with a consequent increase in the number of allocated registers, that further researches in such direction could significantly reduce. Finally, the maximum frequency of the circuit obtained through the proposed approach is close to the one obtained with the FSM-based methodology, because the critical path of the design resides in the datapath. Slight differences have been observed, reflecting different optimization performed by the synthesis tool.

The main result achieved by this thesis work is that it provides a scheduling technique for the design automation of flexible cores, which support unknown, uncertain and unpredictable information. An unified approach to address all the sources of uncertainty was missing in literature. Moreover, the proposed techniques provide an increase in performance with a limited area overhead.

5 Adaptivity Analysis in HPC Systems Domain

The current generation of HPC systems is characterized by increasingly complex architectures, which are difficult to efficiently program. They often integrate heterogeneous processing units, such as graphic co-processors, accelerators and digital signal processors, which interact with hundreds or thousand of general purpose multiprocessors to maximize the performance while reducing power consumption. The large amount of cores makes it difficult to provide a fast, coherent memory architecture. For this reason, these systems usually exploit distributed non-coherent and non-uniform memory hierarchies, with fast, non-coherent caches connected to each core. This allows to provide scalability on an increasing number of cores, while limiting circuit complexity and power consumption. Nevertheless, it also poses significant challenges, which make these architectures much more complex to program [142]. The developer must extract large amounts of parallelism, while orchestrating communication among the cores to optimize application performance. He or she can (and should) exploit an unprecedented amount of parallelism, while guaranteeing load balancing to reach the target performance. The application should be efficiently partitioned and mapped on the various processing elements. Moreover, the developer needs to handle data localization while minimizing communication between the cores, since the network bandwidth represents the bottleneck of such large-scale systems. In general, the bandwidth is one order of magnitude lower than the computational capacity. Thus, programming these systems requires significant programming skills, and it is time consuming and error-prone.

The design process of modern supercomputers must take into account that many behaviors of the application will be known only at run-time. The application's behavior is fundamental to guide the design process. For example, if the application is characterized by frequent accesses to regular data structures, the programmer can take advantage of temporal and spatial locality by restructuring the code to allow coalesced memory accesses at the node level, or by aggregating memory accesses to allow bulk synchronization among the tasks at the system level, thus reducing the pressure on the bandwidth. When instead the application accesses irregular data structures, the locality level of the program is unknown at design time. Indeed, even if usually poor local, such applications may still present some level of locality, which will be discovered only at run-time. Thus, in this case the designer can only try to estimate the locality level by means of some heuristics. As another example, when the application presents complex data structures and/or complex control flow, it may result difficult to partition and map it over the available resources in a balanced way. This causes unoptimized use of the resources (i.e., overloading of one or more resources), which in turn decreases the throughput of the system by increasing the response time of the overloaded resources. In general, many of the tasks involved in the design of modern HPC systems are affected by unknown, uncertain and unpredictable information. For example, it is not always possible to estimate the locality level of an application, which is crucial to guide partitioning and mapping of the application on the available resources. As another example, when the application contains loops with unpredictable number of iterations, during the partitioning of the code into tasks, the developer must include additional loop termination checks, which impact on the performance. Moreover, the analysis of memory access can often be restricted by the presence of irregular data structures (such as graphs, unstructured grids or unbalances trees). This consequently complicates code partitioning and mapping, often generating load unbalance. These issues have even more significant impact when the application is irregular. Irregular applications present a highly unpredictable behavior. They show data sets difficult to partition, unpredictable memory accesses, unbalanced control flow and fine grained communication. Hand-optimizing every single aspect is hard and time-consuming, and it often does not lead to the expected performance.

For all these reasons, unknown behaviors can significantly impact on the quality of the results. Usually, uncertain information and unpredictable events are handled by means of run-time systems in the high-performance domain. Run-time systems are responsible for tasks creation, mapping and management. They also manage the workload among the nodes of the architecture. The strategy adopted by run-time systems depends on run-time events, which may occur unpredictably. For this reason, such run-times allow the system to dynamically adapt whenever they can benefit from a change. However, such systems only have partial visibility of the application features, inferred by its run-time behavior. For example, they cannot establish if a given behavior is due to the particular set of inputs. In general, run-times allow the system to adapt to a run-time event, without being able to estimate if other similar events will occur, or which is the general trend for a given application. On the other side, compilers are not generally able to handle run-time events, but they have a privileged point of view about the features characterizing an application. For this reason, the idea behind this work is that run-time systems can take advantage of compiler support to increase the performance of parallel execution. More in detail, a combination of these two approaches, where the compiler analysis is used to provide hints to the run-time system, can provide a powerful instrument for efficient parallelization of irregular applications. Indeed, the compiler can naturally study those properties of the application which are transparent to the run-time, and it can apply code transformations and optimizations which simplify the run-time job. Moreover, the run-time system is highly time-constrained: off-line compiler analysis can increase run-time speed by offloading it of some tasks, or by providing code with simpler structure. A natural extension of this approach consists in iteratively applying on-line compiler analysis at some points of the execution, after a preliminary off-line analysis phase, to dynamically provide feedbacks to the run-time.

There is a growing gap between modern complex and highly-parallel HPC architectures and the high-level languages used to describe the application, which were designed for simpler systems and do not consider these new issues. High-level languages are commonly employed due to their ability to hide architectural details, which allows to simplify the programming task, since the programmer can exploit high level construct and data structures, while ignoring the details concerning the underlying architecture. However, C-like languages do not consider many of the issues introduced by manycore architectures. For example, they do not properly expose the inherent parallelism of the specification, which is critical to achieve the expected performance for architectures relying on hundreds or thousands of cores. Furthermore, they generally assume shared memory architectures, thus considering a common and coherent address space across the nodes. Alternatively, they assume accelerator-like architectures, thus completely offloading the computation to custom processors, copying data with bulk transfers only at the beginning and at the end of the computation. Even existing parallel languages extensions for shared memory multiprocessor systems, such as OpenMP, are only suitable at the node level, while they are not sufficient at the systems level. In general, they consider applications and architectures with limited amounts of parallelism. Furthermore, they do not usually expose locality, thus not coping well with non-uniform or distributed memory systems. At the system level, distributed memory systems show to fit better message-passing programming models, such as Message Passing Interface (MPI). In general, modern supercomputers employ a combination of programming models, where for example OpenMP is used at the node level, MPI is used for intra-node communication and other solutions, such as CUDA or OpenCL, are used to manage other components, such as hardware accelerators. The complex behavior of irregular applications further complicates the problem of efficiently programming such systems, making current high-level languages ineffective and forcing developers to spend significant amounts of time in optimizations. Usually, global memory abstractions, such as PGAS, are employed when programming irregular applications, relieving the programmer from the partitioning phase. Moreover, since such applications suffer of poor locality, there are usually low benefits in using caching mechanism. Thus, multithreading becomes indispensable to improve the performance. Finally, irregular applications are characterized by highly intensive, fine-grained synchronization. Since the network bandwidth is significantly lower than the computational capacity of large scale systems, the small messages should be aggregated, thus reducing the pressure on the network.

This Chapter introduces a methodology for the automatic generation of parallel irregular applications. The proposed technique builds on the top of a proper run-time system for irregular applications, the Global Memory and Threading (GMT) system. This methodology aims at producing an optimized parallel version of the code. The compilation approach provides support to GMT to improve the performance of the parallel application running on cluster of supercomputers. For such purpose, the Yet Another Parallel Programming Approach (YAPPA) framework has been implemented as part of the LLVM compiler. YAPPA takes as input a C-like irregular application, instrumented by the programmer with synchronization primitives. Moreover, YAPPA produces as output a parallel application, instrumented with calls to primitives provided by GMT. The main contribution of this Chapter can be summarized as follows:

- Identification of compiler requirements to efficiently generate parallel irregular applications
- Compiler-based transformations: a set of compiler transformations for the automatic parallelization of irregular applications has been proposed. Automatic parallelization aims at reducing development and optimization effort.
- Compiler-based optimizations: a set of transformations for improving the performance of the resulting parallel code has been proposed, focusing on irregular applications.
- LLVM extension: these transformation have been implemented in LLVM and the prototype of the framework has been evaluated on a common irregular kernel (graph Breadth First Search).

The remainder of this Chapter is organized as follows. Section 5.1 introduces the concept of adaptivity analysis for compiler-assisted design of modern HPC systems, with particular attention on irregular applications, and on memory management. The goal of this work in the HPC domain is to provide compiler support for managing unknown information in non-uniform distributed memory supercomputers. The compiler aims to automatically produce an optimized parallel version of the original applications. Section 5.2 overviews the steps that are needed to extend the LLVM compiler for such purpose. More in detail, it describes the compiler requirements for handling irregular applications on such systems. The produced parallel code takes advantage of the functionalities of the GMT run-time systems, which is briefly described in Section 5.2. Section 5.3 describes the proposed compiler transformations and optimizations for the the automatic generation of parallel code, implemented in the Yet Another Parallel Programming Approach (YAPPA) framework. More in detail, this Section describes the steps that YAPPA performs to produce a parallel version of an irregular application targeting clusters of supercomputers, starting from a C-like application, instrumented by the programmer with only synchronization primitives. The Breadth-First Search (BFS) algorithm is used as case study to describe the YAPPA framework. Section 5.4 shows some results for the BFS algorithm.

5.1 Adaptivity Analysis for Automatic Parallelization of Irregular Applications

As previously discussed, unknown information and unpredictable events are usually managed by run-time systems in modern supercomputers. However, since the compiler has a privileged point of you of the application's properties, the idea behind this work is to propose an approach which exploits the potentiality of a combination between compiler and run-time. More in detail, the compiler can provide support by generating an optimized parallel application, which can allow the run-time to better deal with different kinds of irregular applications. However, automatically generating an optimized parallel program is not easy. The compiler should at first restructure the problem so that the task-level parallelism can be effectively exploited, while minimizing intranode communication. This requires to find the dependencies between tasks and to transform the source code to manage the tasks. Then, the original algorithm should be rewrote using a parallel programming notation. The compiler needs to understand which parts of the problem are most computationally intensive and more promising in terms of parallelism, since it is on those parts of the problem that the effort to parallelize the problem should be focused. Once this analysis is complete, the compiler can start the parallelization, by finding the concurrency in the application. After analyzing the concurrency in a problem, the next task is to map the concurrency onto multiple units of execution, which run on a parallel computer. In performing this step, various forces such as simplicity, portability, scalability, and efficiency may pull the design in different directions. The features of the target platform, the characteristics of the programming model adopted, and the features and behaviors of the application must also be taken into account. Also in this case, the presence of unknown information introduces many new challenges, which at today are still ongoing.

5.1.1 Irregular Applications

Many emerging classes of applications, such as computer vision, machine learning and data mining problems, are irregular [144]. This class of applications exploits dynamic and linked data structures such as graphs, unbalanced trees or unstructured grids, whose size is potentially variable during the lifetime of the algorithm. Irregular applications are inherently parallel, since they potentially perform parallel computations for each element of the data structures, and they usually operate on considerable amount of data. Thus, they may seem a good fit for modern massively parallel HPC systems. However, the same data structures are subject to unpredictable, fine-grained accesses, they are very difficult to partition in a balanced way, they have almost no locality, and they present high synchronization intensity. Conversely, modern HPC systems rely on regular computation and locality exploitation to reach the peak performance. Thus, global memory abstractions, such as PGAS, are preferred to non- uniform distributed memory architectures. Moreover, these scientific programs are usually unstructured, sparse, adaptive or block structured. They need run-time support to manage data movement, needed to efficiently implement irregular and block structured scientific algorithms on distributed memory machines and clusters of supercomputers. For these reasons, developing irregular applications on them poses complex challenges and require significant programming efforts.

In general, the difficulty of parallelizing a given program is strongly correlated to its degree of irregularity. The applications that have been parallelized most successfully on large scale multiprocessors are those with a high degree of regularity. Most often, irregular applications benefit from parallelization techniques which are very different from those which result instead effective for regular applications. For example, their performance are generally improved by non-deterministic parallel implementations, since any deterministic schedule of the tasks may result in poor performance for some inputs. As another example, since it is often impossible to predict the amount of computation that will be associated to a given data structure (unknown information), and consequently to a given task, irregular applications benefit from dynamic scheduling and load balancing techniques. Implementing irregular applications on distributed memory machines requires the mapping of the data structures across the nodes. Whether or not the programming model provides a single address space for accessing the memory, the hierarchy cannot be ignored, when trying to achieve high performance. Moreover, the programmer has to manage shared data. There are two main approaches to implement shared distributed data structures. The first one, which is known as *replication*, consists in keeping several identical copies of the data structure in different nodes. This technique provides high throughput for read-only operations, at the cost of consistency traffic for write operations. The second methodology is known as *partitioning*. It consists in dividing the data structure in portions, and in allocating each portion to a different node. Partitioning has opposite characteristics with respect to replication: it does not generate traffic for consistency reasons, since there exists a single copy of each portion of data; on the other side, even read-only operations can require remote accesses, when the required data is allocated to a different node with respect to the one which needs the data. There exist also hybrid approaches, which rely on a combination of these two techniques. Finally, given the amount of unknown, uncertain and unpredictable information, irregular applications can significantly benefit from compiler support: the run-time can work on an optimized version of the code, which can be simpler to handle, since it better shows some properties of the application.

5.1.2 Parallel Programming Model for Irregular Applications

Choosing a particular parallel programming models can be difficult. The learning curve associated with these models is steep and time consuming. Hence, it is not practical to master several notations in order to choose the one to use. According to [102], what programmers and/or compilers need is *a quick way to learn the "flavor" and high level-characteristics of different models in sufficient detail to make an intelligent choice of which notation to invest.* Among the most common parallel programming models, OpenMP and MPI are the most adopted. Parallel programming models differ from each other in their complexity, in how much they require the original serial program to change, in how error-prone they are to work with, in the level of portability, and in the performance they provide for a given class of applications targeting a given parallel architecture. All of these factors must be considered in light of the types of parallel algorithms the designer intends to work with. My position about the choice of a parallel programming model is that none of the existing approaches completely fits the design of parallel irregular applications. On one hand, shared memory models, like OpenMP, are difficult to use on distributed memory systems, since the programmer or the compiler needs to explicitly manage synchronization and communication, which is not provided by the language extension. On the other hand, distributed memory models, like MPI, require the programmer or the compiler to partition the application among the distributed memory of the system. As previously discussed, this task results in load unbalance for irregular applications. PGAS data models represent a good fit to take advantage of the simplicity of shared memory abstractions, thus not requiring the programmer/compiler to partition the application. Such model provides a shared memory abstraction while simultaneously not hiding completely the concept of locality. However, the SPMD model which is usually coupled with PGAS data models, results inadequate for irregular applications, which are characterized by highly intensive fine-grained synchronization.

5.2 Extending the LLVM Compiler

The goal of this work in the HPC domain is to introduce YAPPA (Yet Another Parallel Programming Approach), a compilation framework, based on the LLVM compiler, for the automatic parallelization of irregular applications on modern HPC clusters. YAPPA aims at providing automatic generation of efficient parallel code, focusing on irregular applications. Indeed, at today, parallelizing irregular applications represents one of the most challenging problems in HPC systems design. The compiler transformations and optimizations proposed in the literature so far only cope with regular computation. Trying to fill this gap, the YAPPA compiler provides support to a proper run-time system, thus helping managing unknown information and unpredictable events. This Section introduces some preliminary concept for the definition of an efficient compilation framework for irregular applications. Considering the features of irregular workloads, this work starts by introducing an efficient parallel programming approach for these applications on non-uniform, non-coherent distributed memory systems. Such parallel programming approach is implemented in the GMT (Global Memory and Threading) run-time system. GMT enables a global address space across all the cache memories of the system. It provides latency tolerance through lightweight software multithreading while supporting a simple fork/join parallel control model for dynamic parallelism management. GMT enables a more efficient execution of irregular applications on distributed memory architectures, by addressing some of their main issues. Then, the set of compiler requirements needed for improving the performance of this class of applications are identified. Indeed, irregular applications require to rethink many aspects of compiler transformations and optimizations. A set of transformations that can reduce the development and optimization effort are proposed. Such transformations are implemented in the YAPPA compilation framework, mapped on the top of GMT, which will be discussed in Section 5.3.

5.2.1 GMT: a Global Memory and Threading Run-Time System

GMT is a run-time library that enables fundamental features for irregular applications on distributed memory HPC systems. First, similarly to PGAS programming models, GMT enables a global address space across the cluster. This allows the programmer to develop the application without partitioning the data set. The data structures which are shared among the threads are allocated in the global address space. A special type of data, the *gmt_data_t*, has been defined for such purpose. Such data structures are allocated trough the *gmt_alloc* primitive, provided by the GMT API. The physical location of memory where the data is stored is not exposed to the programmer. Second,

Table 5.1: The GMT Library API exposed to the programmer/compiler.

Primitive	Description
gmt_data_t **gmt_alloc** (uint64_t size, allocPolicy_t allocPolicy)	Allocate space in the virtualized global address space with the specified allocation policy (local, remote, partitioned)
void **gmt_free** (gmt_data_t gmtArray)	Free space in the virtualized global address space
void **gmt_set_data_state** (gmt_data_t gmtArray, gmt_data_state_t state)	Define gmtArray as read-write or read-only
void **gmt_waitCommand**( )	Synchronization primitive: wait for completion of previous non-blocking operations
void **gmt_put** (gmt_data_t gmtArray, uint64_t offset, const void * data, uint64_t size)	Blocking-write a local array in the virtualized global address space starting from the specified offset
void **gmt_put_NB** (gmt_data_t gmtArray, uint64_t offset, const void * data, uint64_t size)	Non-Blocking version of the gmt_put
void **gmt_putValue** (gmt_data_t gmtArray, uint64_t offset, const void * data, uint64_t size)	Blocking-write a value in the virtualized global address space starting from the specified offset
void **gmt_putValue_NB** (gmt_data_t gmtArray, uint64_t offset, const void * data, uint64_t size)	Non-Blocking version of the gmt_putValue
void **gmt_get** (gmt_data_t gmtArray, uint64_t offset, void * data, uint64_t size)	Blocking-read a potion of an array in the virtualized global address space starting from the specified offset and copy it into a local array
void **gmt_get_NB** (gmt_data_t gmtArray, uint64_t offset, void * data, uint64_t size)	Non-Blockingversion of the gmt_get
int64_t **gmt_atomicAdd** (gmt_data_t gmtArray, uint64_t offset, int64_t value, uint8_t size)	Perform atomic addition between the specified value and the specified array in the virtualized global address space, starting from the specified offset
int64_t **gmt_atomicCAS** (gmt_data_t gmtArray, uint64_t offset, int64_t oldValue, int64_t newValue, uint8_t size)	Perform atomic Compare-And-Swap. Use the specified array in the virtualized global address space (compare), starting from the specified offset. Then write the specified value in the virtualized global address space (swap)
void **gmt_parFor** (uint32_t nThr, uint32_t chSize, void (*func) (int, void *), void * args, uint32_t argsSize)	Execute the specified function, corresponding to the loop body, in parallel

GMT implements lightweight software multithreading, which allows to tolerate the latencies for accessing data at remote locations. When a core executes a task which issues an operation to a remote memory location, it switches to another task while the memory operation completes, hiding the access latency with other computation. Third, GMT implements a fork/join control model, which is typical of shared memory systems. Rather than creating different processes at the beginning of the application and then mapping the workload among the nodes (as, for example, in MPI and UPC/GASNet), GMT dynamically creates new tasks. With respect to SPMD control models, typical of message passing, or PGAS programming models, this model better copes with the large amounts of fine-grained and dynamic parallelism of irregular applications. Indeed, since GMT targets irregular applications, most of the parallelism is expected to reside in loops (e.g., graph visiting problems). Table 5.1 shows the primitives that the programmer or the compiler can use to interface with GMT. GMT works by manipulating arrays stored in the global memory space. Among the features offered by GMT, the memory allocation primitives allow expressing locality. A thread can allocate data partitioned (PARTITION) among all the memories of the system, on the memory of the core currently executing the thread itself (LOCAL), or remotely (REMOTE) on all the other memories, except the current one. The analysis of the irregular memory accesses, together with the particular code partitioning helps to establish the allocation policy for each data. Put and get operations allow manipulating data in the GMT memory space, by accessing the requested number of elements at the specified offsets of the arrays. There are blocking and non-blocking operations, with the related wait operation. More in detail, the global data structures are accessed trough the GMT primitives *gmt_put* and *gmt_get*, or through their corresponding non-blocking versions (i.e. *gmt_put_NB* and *gmt_get_NB*). The *gmt_getNB* and *gmt_putNB* primitives are used when a data is only read or only written inside the loop body, as well as when each task reads/writes a different portion of a data structure. The transformation of blocking get/put in their non-blocking version requires the compiler to perform alias analysis over the global memory accesses. To reduce the pressure on the network, the wait operation is not associated every time with a specific non-blocking operation, but rather waits until all previous operations have been completed. GMT also provides primitives for atomic operations over shared data, such as addition (*gmt_atomicAdd*) and compare-and-swaps (*gmt_atomicCAS*). The parallelism is expressed through a parallel-for construct (*gmt_parFor*), which dynamically spawns one or more iterations of a parallel loop as independent tasks. There are allocation policies also for tasks. When GMT spawn new tasks, it can distribute them uniformly across the cores (PARTITION), it can map them on the same core that encountered the parallel for construct (LOCAL), or it can map them remotely (REMOTE) on all the other cores except the current one. Once assigned to a core, tasks are stored with their contexts in local queues allocated in the private memory, and do not migrate. Cores can access their own private caches in few cycles, thus providing a rather efficient software multithreading. Whenever a parent task spawns new children, it suspends its execution until the termination of the all the children. Since barriers are implicit at the join, this approach avoids expensive task termination checks. Furthermore, if a local task queue is full, and the run-time identifies a request to spawn new tasks, the execution continues in the current context. Task creation, memory allocation and memory operations are commands that are sent, routed and received through message passing operations, which are trivially mapped onto the native communication layers of any supercomputer. Tasks are described as functions which, similarly to pthreads, take a structure of parameters in input. GMT automatically provides an iteration identifier, which acts as task identifier, as an argument of the function. Such identifier allows to compute the offsets for accessing the data in the global address space. A developer can directly use GMT to implement irregular applications by explicitly calling the API primitives. However, this means parallelizing the code and carefully manipulating the data in the global address space with the put and get operations. YAPPA builds on top of this run-time system. It automatically transforms applications written in pseudo-sequential C (with only synchronization constructs, where required) with a shared memory abstraction into parallel code that exploits the run-time's primitives to manipulate data in the global address space.

5.2.2 Compiler Requirements for Irregular Applications

Applications may present irregularity in at least three different ways [144]. It may appear in control structures, in the form of conditional statements or in data structures (e.g., dynamic linked lists employed to represent graphs, unbalanced trees, unstructured grids), which usually imply unpredictable fine-grained data accesses and poor spatial and temporal locality. Finally it may appear in the communication patterns, leading to non-determinism and requiring fine-grained synchronization to coordinate the accesses on common memory locations. Nevertheless, irregular programs are inherently parallel. Usually, they contain significant amount of parallelism in the loops that explore the elements of the data structures. Given a shared memory abstraction, which relieves the programmer from data partitioning, and given approaches such as multithreading, which allow hiding data access latencies more efficiently than caching in this poor local class of applications, the main objective of a compiler for irregular applications is thus extracting parallelism by generating tasks from loop iterations. In general, this implies providing better dependence analysis, which in turn allows implementing more effective loop transformations, and detailed control on the tasks' granularity [18]. Among the various transformations, we can find the recognition of linear recurrences (such as reductions) and their rewriting in parallel form, the collapsing of nested parallel loops to help ensuring abundant parallelism, loop fusion, to improve reuse and to reduce overheads, loop fissions to reduce register pressure and to isolate serial sections, loop interchange to reduce overheads of parallelization and to move vectorizable loops, and various loop tiling approaches for better reuse. This is in contrast with current OpenMP compilers, which mostly focus on regular applications. They provide only limited dependence analysis, relying almost only on pragmas and focusing only on the parallelization of the first level of loop nests. This may provide sufficient coarse-grained parallelism for multicore designs (dozen of threads), but it does not cope well with manycores (hundreds of threads) and with the finer grained parallelism required to hide latency through multithreading, especially with distributed memory architectures. Furthermore, there is no support for parallelizing recurrences and reductions. Not even the polyhedral model is currently able to provide some loop transformations for common situations in irregular programs, such as reductions and recurrences, because they are not plausible affine transformations. In addition, the polyhedral model cannot introduce explicit synchronization to protect against races. GMT provides a set of features to address issues of irregular applications running on distributed memory architectures. However, whenever a developer directly accesses the library's API, he or she still has to explore a very wide design space. For example, localizing certain globally accessible data, through the locality parameter for memory allocation operations, may provide performance benefits (by eliminating communication) if a new set of tasks that use those data is spawned with the same locality. This is effective even in presence of lightweight multithreading for latency tolerance. As another example, if a loop accesses the elements of a global array through get operations, reading one location at each iteration, the memory accesses can be hoisted from the loop and transformed in a single get operation (blocking) for an entire (sub)array. Since memory access operations on the global memory space flow through the network, which is usually characterized by high latency, aggregating them provides significant benefits by reducing wasted bandwidth for packet headers. The sub-array may even be locally allocated and locally accessed by each task. As one more example, although having a large amount of tasks provides more opportunities to switch, and thus higher probability to tolerate data access latencies, it may not always be beneficial to extract all the fine-grained parallelism identified in loop nests. In fact, extracting too much parallelism may quickly saturate GMT's queues with tasks that provide only few instructions. This in turn may significantly impact the utilization of a core, which would spend the majority of the cycles for switching among tasks rather than performing useful computation, because of the cost of software multithreading. To limit this issue, GMT allows to specify the number of loop iterations which will be associated to each task (chunk size). Moreover, the compiler infrastructure may also independently perform loop unrolling and software pipelining. Furthermore, sequentially executing some nesting levels of a loop may provide larger tasks with more opportunities to aggregate multiple data transfers. In some cases, parallelizing every nesting level leads to the creation of tasks with a low number of instructions, such as in the case of perfect nesting, so sequential execution is preferred. In some other cases, because of the dynamic nature of the parallelism in irregular applications, the number of loop iterations may be unknown at compile time. Consequently, the chunk size cannot be statically computed. However, the compiler could insert in the code the instructions needed to dynamically compute the chunk size at runtime. Enabling the compiler to perform these explorations and the consequent optimizations, rather than requiring the programmer to do them by hand, significantly reduces the development time. In general, compiler transformations and strategies which are suitable for irregular applications may be different from those usually which result efficient for regular applications. For example, regular applications benefits from large chunk sizes, which allow to maximize spatial and temporal locality and to increase data reuse in caches. Irregular applications, instead, have very poor locality. Thus, with a latency tolerance mechanism in place, such as multithreading, finer grained parallelization is definitely preferred. Nevertheless, full control on the parallelization granularity is still desired, especially in software supported approaches. Consequently, in contrast with current OpenMP-based compilers, which parallelize only the outermost loop level, better dependence analysis is required to enable more efficient loop transforms in optimizing compilers for irregular applications. As above mentioned, the combination of compiler transformations which are suitable for regular applications may most probably be not suitable for irregular applications, and vice versa. Indeed, in this domain, the advantaged and disadvantages related to each transformation or to a combination of transformations is radically different. As an example, consider loop fusion, which takes two or more loops and transform them into a single loop, containing the original loop bodies. Loop fusion has a main drawback when applied to regular applications: since it increases the loop body size, it reduces data locality exploitation by increasing the amount of data in cache, because most probably different loops will access different data structure. However, when loop fusion is applied to irregular applications, which are characterized by random accesses to the memory, increasing the loop body size may expose more computation to overlap to memory accesses. Irregular applications may be thus capable of exploiting the advantages of loop fusion, without suffering of its drawbacks. Another disadvantage of loop fusion is that it could cause the formation of loops with more complex control flow. Since irregular applications are by definition characterized by irregular and complex control flow, this issue may not further affect the performance.

5.3 The YAPPA Compilation Framework

This Section introduces the proposed methodology for the automatic generation of parallel irregular applications on modern supercomputers. This technique has been implemented on the top of the LLVM compiler, introducing the Yet Another Parallel Programming Approach (YAPPA) compilation framework. YAPPA extends the LLVM compiler through a set of new transformations and optimizations for irregular applications. It targets the GMT run-time, executing on top of modern HPC clusters. It takes in input a C/C++ application (with synchronization constructs) and produces a parallel C/C++ application instrumented with GMT primitives. YAPPA analyzes and transforms LLVM's intermediate representation, which describes code in the Static Single Assignment (SSA) form. YAPPA performs the transformations in two steps: data management and loop parallelization.

- *Data management:* in the first step, YAPPA analyzes the code to identify potentially shared data to be allocated in the global address space. Then, it redefines the type of these data as *gmt_data_t*, inserting the appropriate memory allocation primitives. At the same time, it transforms all the accesses to these shared data in GMT memory operations: *gmt_get* for reads, *gmt_put* for writes of arrays in GMT data structures, and *gmt_putValue* for writes of scalar values. In this first step, all the memory operations over share data are temporary identified as blocking operations. At this stage, YAPPA also performs alias analysis on the global memory accesses. It identifies independent memory accesses, as for example: accesses to global data structures, which are only read or only written inside the loop body, or global data structures, which at each iteration are accessed on different elements and do not have loop-carried dependences. In these cases, YAPPA tries to move at the beginning of the loop as many independent memory operations as possible, substituting blocking memory operation primitives with their equivalent non-blocking versions and the related wait before the first use of a value. This transformation has been named *unblocking* of memory accesses.

- *Loop parallelization:* in the second step, YAPPA effectively performs the parallelization, by creating the tasks. It extracts the loop bodies, and it generates the task functions, which have two incoming arguments. The first argument is the iteration identifier, which also works as a task identifier. The actual parameter passed to the function is the iteration index, when the chunk size (i.e. the number of iterations that each task executes) is equal to 1, or the first iteration index of the chunk otherwise. The second argument of the task function is a structure, which contains all the variables that are read inside and defined outside the loop. These also includes references to GMT global data structures. YAPPA performs dependence analysis to identify these variables. It inserts allocation and initialization of this structure into the code. All the variables, which are defined outside the loop, and used only inside the loop, are *localized*. In other words, the definition is moved inside the loop body, thus avoiding to pass as a parameter a variable which is going to be dead at the loop exit. Because of the distributed memory architecture, passing parameters to tasks corresponds to memory copies and data transfers, thus localizing variables potentially provides lower communication overheads. Once YAPPA has created the task function, it computes the chunk size. YAPPA also accepts chunk sizes as a command line option. In this prototype of the compiler, the command line option has been exploited to allow hand tuning of the chunk size according to the performance provided by the parallelized program. The application developer can quickly explore several chunk size alternatives in reasonable times through simple compilation scripts. Future versions of the compilation framework will implement the support for automatically computing the chunk size according to the irregularity level of each loop. The choice of the right size for the chunks depends on information which is potentially unknown at compile time, since the compiler is required to estimate the workload in every task, which is usually unpredictable in irregular applications. Irregularity analysis is not currently supported, it will be available in future versions of the YAPPA compiler. The transformation continues with the insertion of the instructions for computing the number of tasks at run-time, since it is possibly unknown at design time. The number of tasks spawned by the run-time corresponds to the number of iterations divided by the chunk size. If the number of iterations is not an exact multiple of the chunk size, another task is added to execute the remaining iterations. Finally, the transformation concludes with the addition of the call to the *gmt_parFor* primitive, whose inputs are the computed number of tasks, the chunk size, the pointer to the task function extracted from the loop body, the pointer to the structure of the parameters and its size in bytes.

YAPPA parallelizes nested loops by topologically ordering the loops according to their nesting level, and by recursively running the parallelization pass on the output of the previous execution. In general, current common parallelizing compilers do not support parallelization of nested loops, because they only look for limited amounts of parallelism, and the target systems do not require large amounts of fine-grained tasks. On the other hand, a custom solution such as the Cray XMT compiler, which targets a massively multithreaded system, concentrates its analysis and parallelization efforts on nested loops. YAPPA performs loop normalization to simplify the computation of the number of tasks. The first prototype of the compiler allows to select the loops to parallelize through a command line option. The approach is equivalent to annotating parallel loops in the program with pragmas, as it is common in solutions such as OpenMP. Nevertheless, the goal of this work is to extend YAPPA so that it can autonomously and automatically select the loops to parallelize. It is also important to underline that not parallelizing certain nested loops may also provide options for further communication optimizations. YAPPA, in fact, also support another optimization, dubbed *block-hoisting*. Whenever a loop reads a shared array, YAPPA translates these accesses to get operations. However, if the loop only reads scalar values sequentially, one iteration after the other, from the array, and the loop is not parallelized, this only generates a sequence of very fine grained operations inside the same task. Even if GMT can tolerate data access latencies by switching to other tasks, there still is benefit in aggregating as much communication as possible. In such a case, YAPPA hoists the scalar read operations from the loop and aggregates them in a single get operation, writing to an entire local sub-array. The sub-array is allocated in the local memory of the task (i.e., with a standard malloc or with a *gmt_alloc* with the LOCAL policy). In the proposed approach, the eligible loops for parallelization are those which are canonical, and which do not contain *invoke* instructions in their body. YAPPA executes LLVM's *lower-invoke* pass to transform invoke instructions into call instructions. The reason for this is that invoke instructions are a particular type of call instructions that the LLVM intermediate representation uses to handle exceptions in C++ applications. If a called function throws an exception, then the related invoke returns 0. Otherwise, it returns 1. According to the return value, the control flow continues with the normal execution or branches to a landing pad. Because loop bodies may contain one or more invoke instructions, and the proposed parallelization scheme transforms loop bodies in parallel and asynchronous tasks distributed across different processing elements, non trivial mechanisms to identify tasks generating exceptions, and to establish a policy to handle the exceptions would be required. Parallelization of C++ applications also requires supporting object serialization and deserialization, when they are passed as task parameters. In the current YAPPA implementation, if a variable of an object is shared, then the entire object is allocated in the global address space, provided that it does not contain pointer fields. This particular kind of objects is not currently supported. Future optimizations will selectively allocate in global memory only the shared variables, leaving the object and all the other variables in the private memory. In the following, the functionalities provided by YAPPA are explained in detail by means of the most typical example of irregular application: the Breadth First Search (BFS) algorithm.

Algorithm 1 Sequential version of the BFS Algorithm (with synchronization operations where needed).

```
1: while ($Q_N != 0$) do
2:     for ($vId = 0; vId < Q_N; vId++$) do
3:         $uint64\_t\ vertex = gQ[vId]$
4:         $uint64\_t\ curIdx = gIdxs[vertex]$
5:         $uint64\_t\ nextIdx = gIdxs[vertex + 1]$
6:         for ( $uint64\_t\ i = curIdx; i < nextIdx; i++$) do
7:             $uint64\_t\ neighbor = gEdges[i];$
8:             if ($gmt\_atomicCAS(gMarked, neighbor * sizeof(unit64\_t), 0, 1, sizeof(uint64\_t))$) then
9:                 $gMarked[neighbor] = 1;$
10:                $uint64\_t\ Qvalue = gmt\_atomicAdd(gQnext_N, 0, 1, sizeof(uint64\_t));$
11:                $gQnext[Qvalue] = neighbor;$
12:            end if
13:        end for
14:    end for
15:    $gQ$=$gQnext$
16:    $Q_N = gQnext_N$
17:    $gQ_next = 0$
18: end while
```

5.3.1 Example

Algorithm 1 shows the pseudo-code of a simple, sequential queue-based Breadth First Search Algorithm (BFS), where synchronization primitives have been added manually (lines 8 and 10). The algorithm explores a graph described in the Compressed Sparse Row (CSR) representation. The algorithm works as follows. The queue Q contains the vertices to explore in the current iteration, $gQnext$ contains the vertices that will be explored in the subsequent operation. The variables gQ_N and $gQnext_N$ count the number of elements in queue gQ and $gQnext$, respectively. Initially, gQ only contains the root node of the graph. The algorithm loads the edge list of each vertex in the exploration queue. It does so by accessing the array of indexes ($gIdxs$) that, for each vertex, contains the offset at which its edges are located in the array of edges ($gEdges$). Each element of the edge array contains the target vertex for the edge (neighbor for the source vertex), following the CSR representation. It then evaluates each neighbor by checking the corresponding entry in the array of the marked vertices ($gMarked$). If the neighbor has not been previously explored (its marked status is 0), then the neighbor is marked and added to the queue containing the new vertices to explore in the next iteration ($gQnext$). Checking of the marked vertices array and addition to the queue are operations that, when executing in parallel, require synchronization. Since YAPPA is not currently able to automatically manage synchronization, the programmer must express it with atomic constructs. The compare-and-swap ($gmt\_atomicCAS$) has been used to check and access the array of the already analyzed elements

Algorithm 2 Arguments struct for the loop-body function in the BFS Algorithm (no optimizations enabled, or only memory unblocking enabled).

```
**typedef** $struct\ Args\_t\{$
    $gmt\_data\_t\ gMarked;$
    $gmt\_data\_t\ gIdx;$
    $gmt\_data\_t\ gEdges;$
    $gmt\_data\_t\ gQ$
    $gmt\_data\_t\ gQnext;$
    $gmt\_data\_t\ gQnext_N;$
$\}args\_t;$
```

Algorithm 3 Parallelized loop-body function for the outermost loop in the BFS Algorithm (no optimizations).

```
$void\ F(uint64\_t\ iterId, void * Args)\{$
$args\_t*\ args = (args\_t*)Args;$
$uint64\_t\ vertex, curIdx, nextIdx;$
$gmt\_get(args \rightarrow gQ, iterid * sizeof(uint64_t), \&vertex, sizeof(uint64\_t));$
$gmt\_get(args \rightarrow gIdxs, iterid * sizeof(uint64_t), \&curIdx, sizeof(uint64\_t));$
$gmt\_get(args \rightarrow gIdxs, (iterid{+}1){*}sizeof(uint64_t), \&nextIdx, sizeof(uint64\_t));$
**for** ( $uint64\_t\ i = curIdx; i < nextIdx; i++$ ) **do**
    $gmt\_get(args \rightarrow gEdges, i, neighbor, sizeof(uint64\_t));$
    **if** $(gmt\_atomicCAS(args \rightarrow gMarked, neighbor * sizeof(unit64\_t), 0, 1, sizeof(uint64\_t)))$ **then**
        $gMarked[neighbor] = 1;$
        $uint64\_t\ Qvalue = gmt\_atomicAdd(args \rightarrow Qnext_N, 0, 1, sizeof(uint64\_t));$
        $gmt_put(args \rightarrow Qnext, Qvalue{*}sizeof(uint64\_t), \&neighbor, sizeof(uint64\_t));$
    **end if**
**end for**
$\}$
```

$gMarked$, and the array of neighbors for the current element $neighbor$. Moreover, the increment of the index, which reads the global data $gQnext$ is synchronized by means of an atomic addition (*gmt_atomicAdd*) operation. It is not required to specify that those data are shared (and thus allocated in the global address space). When the algorithm explores all the vertices in gQ, $gQnext$ becomes the new gQ and $gQnext$ is reset together with its element counter. The procedure continues until a new iteration of the algorithm finds gQ empty, meaning that all the vertices of the graph have been visited. Each iteration of the algorithm visits a level of the graph.

The proposed implementation of the BFS algorithm provides two potentially parallel for loops (lines 2-6) in Algorithm 1. The typical parallel implementation of this queue-based implementation focuses on the extraction of parallelism of the outermost loop, because with the most complex graphs, after a few iterations, the exploration queues already offer abundant parallelism. YAPPA is able to parallelize both the loops, extracting fine-grained parallelism while also allowing precise control of the task size. Nevertheless, for sake of simplicity of exposition, this example follows the common approach of parallelizing only the outermost loops. This also allows presenting how YAPPA's communication optimizations work.

The original loop body accesses the arrays $gMarked$, $gIdx$, $gEdges$, gQ, $gQnext$, and the scalar $gQnext_N$, which are defined outside the loop. Thus, they all become arguments for the task. YAPPA redefines these arrays as *gmt_data_t*, because all the tasks access them. Algorithm 2 shows the complete arguments data structure. Reads of the elements in the queue gQ, and in the arrays $gIdxs$ and $gEdges$ are all on shared data structures, thus YAPPA translates them to *gmt_get* operations. The addition of a new vertex in $gQNext$ is also on a shared data structure, thus it is converted to a *gmt_put* operation. Algorithm 3 shows how YAPPA transforms the outermost loop, in the case when it does not perform any optimization. The task function takes as input the iteration identifier $iterId$ and the pointer to the structure of the parameters $Args$.

Algorithm 4 shows the parallelized version of the main routine in the BFS algorithm, when no optimizations are enabled. This example shows (for sake of simplicity, only for the array $gIdxs$, since the others are equivalent) how the previous allocations (done with standard mallocs) of the shared data structures are converted to *gmt_malloc* operations. It is also possible to see how the arguments are passed to the task (line 4-10), the computation of the number of tasks (line 11-13) and the call to the *gmt_parFor* primitive (line 15).

Moreover, it is possible to show how the block-hoisting optimization is performed on the same example. The loop on the edge list is not parallelized. However, this loop reads, for each vertex, sequential elements in the list (from $curIdx$ to $nextIdx$ with a simple iterator). So, YAPPA can aggregate all the reads in a single get operation. Because $curIdx$ and $nextIdx$ are not stati-

Algorithm 4 Parallelized version of the main routine in the BFS Algorithm (no optimizations enabled).

```
$uint64_t * gIdxs = gmt_alloc(vertex + 1 * sizeof(uint64\_t);$
...;
while $(Q_N != 0)$ do
    $args\_t\ args;$
    $args.gMarked = gMarked;$
    $args.gIdx = gEdges;$
    $args.gIdxs = gIdxs;$
    $args.gQt = gQ;$
    $args.gQnext = gQnext;$
    $args.gQnext_N = gQnext_N;$
    $uint64\_t\ nThr = Q_N/chunkSize;$
    if $(nThr * chunkSize < Q_N)$ then
        $nThr++;$
    end if
    $gmt\_parFor(nThr, chunkSize, F, \&args, sizeof(args))$
    $gQnext = gQ;$
    $get(gQnext_N, 0, \&Q_N, sizeof(uint64\_t));$
    $put(gQnext_N, 0, 0, sizeof(uint64\_t));$
end while
```

Algorithm 5 Arguments struct for the loop-body function in the BFS Algorithm (block-hoisting enabled).

```
typedef $struct\ Args\_t\{$
      $gmt\_data\_t\ gMarked;$
      $gmt\_data\_t\ gIdx;$
      $gmt\_data\_t\ gEdges;$
      $gmt\_data\_t\ gQ$
      $gmt\_data\_t\ gQnext;$
      $gmt\_data\_t\ gQnext_N;$
      $gmt\_data\_t\ neighbors;$
$\}args\_t;$
```

cally know, it must also dynamically allocate the array. For this example, it is allocated with a standard $malloc$ in the task memory space, so that conversion of memory operations to gets and puts are not required to access it. Algorithm 5 shows the complete argument list for the loop body function. With respect to the case without optimizations, in this case the $neighbors$ array must be added to the parameters list. Algorithm 6 shows the loop body function when

Algorithm 6 Parallelized loop-body function for the outermost loop in the BFS Algorithm (Block-Hoisting enabled).

```
$void\ F(uint64\_t\ iterId, void * Args)\{$
$args\_t*\ args = (args\_t*)Args;$
$uint64\_t\ vertex, curIdx, nextIdx;$
$gmt\_get(args{\rightarrow}gQ, iterid * sizeof(uint64_t), \&vertex, sizeof(uint64\_t));$
$gmt\_get(args{\rightarrow}gIdxs, iterid * sizeof(uint64_t), \&curIdx, sizeof(uint64\_t));$
$gmt\_get(args{\rightarrow}gIdxs, (iterid{+}1){*}sizeof(uint64_t), \&nextIdx, sizeof(uint64\_t));$
$uint64\_t\ {*}neighbors = malloc((nextIdx - curIdx) * sizeof(uint64\_t));$
$gmt\_get(args{\rightarrow}gEdges, cur[0] * sizeof(uint64\_t), neighbors, (nextIdx - curIdx) * sizeof(uint64\_t));$
for ( $uint64\_t\ i = curIdx; i < nextIdx; i++$ ) do
    if $(gmt\_atomicCAS(args{\rightarrow}gMarked, neighbors[i] * sizeof(unit64\_t), 0, 1, sizeof(uint64\_t)))$ then
        $gMarked[neighbors[i]] = 1;$
        $uint64\_t\ Qvalue = gmt\_atomicAdd(args{\rightarrow}Qnext_N, 0, 1, sizeof(uint64\_t));$
        $gmt_put(args{\rightarrow}Qnext, Qvalue{*}sizeof(uint64\_t), \&neighbors[i], sizeof(uint64\_t));$
    end if
end for
$\}$
```

the Block-Hoisting optimization is enabled, while Algorithm 7 shows the corresponding main routine. Finally, Algorithm 8 shows how to perform memory unblocking: the queue $Qnext$ can be updated by means of a gmt_putNB, since the index has been already computed by means of an atomic operation.

5.4 Experimental Evaluation

The main benefit provided by this first version of the YAPPA compilation framework obviously is a reduction in development time of irregular applications on platforms running GMT. However, it is still possible to validate the effectiveness of its transformations. For this reason, this Section shows the performance of the GMT-YAPPA framework on a case study: I selectively apply the transformations on the full queue-based BFS algorithm. The overall performance of each version of the code produced by YAPPA is measured by executing it on a system which emulates a many core design with private caches running GMT.

Algorithm 7 Parallelized version of the main routine in the BFS Algorithm (Block-hoisting enabled).

```
1: $uint64_t * gIdxs = gmt_alloc(vertex + 1 * sizeof(uint64\_t);$
2: ...;
3: while $(Q_N != 0)$ do
4:     $args\_t\ args;$
5:     $args.gMarked = gMarked;$
6:     $args.gIdx = gEdges;$
7:     $args.gIdxs = gIdxs;$
8:     $args.gQt = gQ;$
9:     $args.gQnext = gQnext;$
10:    $args.gQnext_N = gQnext_N;$
11:    $args.neighbors = gneighbors;$
12:    $uint64\_t\ nThr = Q_N/chunkSize;$
13:    if $(nThr * chunkSize < Q_N)$ then
14:        $nThr + +;$
15:    end if
16:    $gmt\_parFor(nThr, chunkSize, F, \&args, sizeof(args))$
17:    $gQnext = gQ;$
18:    $get(gQnext_N, 0, \&Q_N, sizeof(uint64\_t));$
19:    $put(gQnext_N, 0, 0, sizeof(uint64\_t));$
20: end while
```

Algorithm 8 Parallelized loop-body function for the outermost loop in the BFS Algorithm (Memory Unblocking and Block-Hoisting enabled).

```
1: $void\ F(uint64\_t\ iterId, void * Args)\{$
2: $args\_t * args = (args\_t*)Args;$
3: $uint64\_t\ vertex, curIdx, nextIdx;$
4: $gmt\_get(args \to gQ, iterid * sizeof(uint64_t), \&vertex, sizeof(uint64\_t));$
5: $gmt\_get(args \to gIdxs, iterid * sizeof(uint64_t), \&curIdx, sizeof(uint64\_t));$
6: $gmt\_get(args \to gIdxs, (iterid+1)*sizeof(uint64_t), \&nextIdx, sizeof(uint64\_t));$
7: $uint64\_t *neighbors = malloc((nextIdx - curIdx) * sizeof(uint64\_t));$
8: $gmt\_get(args \to gEdges, cur[0] * sizeof(uint64\_t), neighbors, (nextIdx - curIdx) * sizeof(uint64\_t));$
9: for ( $uint64\_t\ i = curIdx; i < nextIdx; i + +$) do
10:    if $(gmt\_atomicCAS(args \to gMarked, neighbors[i] * sizeof(unit64\_t), 0, 1, sizeof(uint64\_t)))$ then
11:        $gMarked[neighbors[i]] = 1;$
12:        $uint64\_t\ Qvalue = gmt\_atomicAdd(args \to Qnext_N, 0, 1, sizeof(uint64\_t));$
13:        $gmt_putNB(args \to Qnext, Qvalue*sizeof(uint64\_t), \&neighbors[i], sizeof(uint64\_t));$
14:    end if
15: end for
16: $\}$
```

Table 5.2: BFS performance (time in seconds, and speedup) when applying the YAPPA compilation framework with GMT on an emulation platform. V is the number of edges for the whole graph, E is the average number of edges for each vertex.

#V-#E	Serial	GMT	GMT-BH
1,000,000-100	1.35	0.52	0.49
1,000,000-1,000	9.83	2.53	2.39
1,000,000-10,000	95.86	22.72	21.27

5.4.1 Experimental Setup

The experimental setup consists in a quad-processor AMD Opteron 6176SE (codename "Magny Cours") with 256 GB of DDR3 RAM. Each processor hosts 2 dies, and each die features 6 cores and a shared L3 cache of 6 MB, for a total of 48 cores. A core includes a private 512 KB L2 cache and private instruction and data L1 caches of 64 KB each. The processors have a frequency of 2.3 GHz. Some of the cores of this system runs a modified instance of GMT, which allocates a private memory area, a private task queue and a private communication command queue pinned in the cache to emulate fast caches. The other cores emulate the interconnection network, by performing the data movement operations from one area to the other. Up to 32 cores are used for processing and 16 for communication. The processing cores host up to 1024 fine-grained tasks.

5.4.2 Experimental Results

Table 5.2 shows the performance, in seconds, of the BFS kernel on the emulation platform, when applying different steps of the YAPPA compilation framework. The first column (*Serial*) shows the serial performance of the code (the code is very similar to the example, with atomic instructions removed). The second column (*GMT*) shows the performance obtained by only applying loop parallelization for GMT. The third column (*GMT-BH*) shows the performance of the kernel when, beside loop parallelization, YAPPA also applies data management optimizations. In particular, for the BFS, the most important optimization is the block-hoisting of the accesses to the edge list. Only the outermost loop level of the benchmark has been parallelized. Graphs of 1 Million vertices have been used, where, for each vertex, the number of average edges has been changed. The size of the graphs range from 0.5 GB to 37 GB. Parallelization allows obtaining reasonable speed ups, given the characteristic of the applications, of the emulated target platform, and the simplicity of the initial sequential code. As expected, the speed up increases as the complexity of the graph (number of edges per vertex) increases. The speed up ranges from

2.5 to 4.22, which for this type of application is reasonable. The block-hoisting optimization provides 6 to 7% higher performance. For comparison, the same queue-based implementation, when parallelized through simple OpenMP and run directly on the host platform (without emulating the target architecture with GMT), only provides speed ups in the range of 2.3 times the performance of the sequential code.

6 Conclusions

Modern embedded systems and high-performance computing systems share different design challenges. Among those shared issues, this thesis discussed how the problem of unknown, uncertain and unpredictable information can be addressed by introducing Adaptivity Analysis in both the domains, at different abstraction levels.

In the Embedded Systems domain, adaptivity analysis has been defined as a design methodology for efficient adaptive hardware cores. Such technique, together with the introduction of a novel architectural model which confer adaptivity capabilities to hardware cores, is proposed as an alternative to the most common approach in High Level Synthesis, which is based on a statically computed scheduling order, and Finite State Machine controllers. Such traditional approach cannot deal with unknown, uncertain or unpredictable information at design-time. Adaptivity analysis aims at filling this gap by introducing an explicit activation mechanism for the instructions, which allows their dynamic auto-scheduling, according to run-time events, which are possibly unknown at design time (e.g., variable latency due to communication with external modules, inherent uncertainty of the specification). This approach allows to dynamically exploit the available parallelism, while ensuring correct results in presence of unknown, uncertain or unpredictable information. At this point, this work introduced a proper Intermediate Representation, namely the Extended Program Dependence Graph (EPDG), which provides a parallel execution model for the specification, according to the dependences among the instructions. The proposed design methodology analyzes the EPDG to compute the conditions that each instruction must satisfy to execute. Such conditions are named Activating Conditions (ACs). As a consequence, the proposed scheduling technique is called dynamic AC-scheduling. The ACs are logic formulas, which express the instructions dependences in a parametric way, since unknown information may occur at design time. At run time, when unknown information is resolved, the ACs are evaluated. When an AC is satisfied, the corresponding instruction can safely execute. The dynamic AC-scheduling, together with the EPDG computation, has been implemented in the PandA framework, thus providing a fully automated design methodology. Furthermore, this thesis introduced a prototype for the architectural design of novel adaptive hardware cores. Experimental results have shown the capability of the resulting design to adapt to unknown conditions, while providing an encouraging increase in performance (around 34% with respect to traditional approaches) with a limited area overhead. Future works in this direction could formalize other HLS tasks according to the AC model. For example, resources allocation and binding or chaining could further increase the performance provided by the proposed methodology, while simultaneously reducing the resulting area. Moreover, supports to HLS optimizations, such as loop pipelining or speculation, could be introduced.

In the high-performance computing domain, unknown, uncertain and unpredictable behaviors are usually managed by underlying run-time systems. Thus, this work employs adaptivity analysis in this domain as compiler support for run-time systems. More in detail, this thesis introduces the Yet Another Parallel Programming Approach (YAPPA) compilation framework, with extends the LLVM compiler with automatic parallelization and optimization of irregular applications. YAPPA builds on top of a proper run-time library, called Global Memory and Threading (GMT). GMT cross combines different parallel programming approaches, which are usually exploited separately in the literature: it integrates Global Address Space (GAS) across cluster nodes, lightweight multithreading for tolerating memory and network latencies, and support for a fork/join programming model. First, YAPPA instruments the sequential code to run GMT primitives (parallelization phase), and then it applies a novel set of transformations to improve the efficiency of the generated parallel code (optimization phase). YAPPA transforms memory operations to use the run-time's communication API, generates the tasks, builds the data structures and inserts the calls to the parallel constructs. YAPPA also performs several optimizations to better use communication resources. The suitability and functionality of the approach has been demonstrated with a prototype of the compiler, generating the parallel code of a typical irregular kernel (graph Breadth First Search) starting from a sequential C specification. The experimental results showed scaling for the parallel version of the code and evaluated the performance of the communication optimizations performed by the compiler, demonstrating an increase in performance. Future work in this direction could extend the framework in order to dynamically provide feedback to the run-time system, as support for load balancing.

Bibliography

[1] AccelDSP Synthesis Tool - Home Page. http://www.xilinx.com/tools/acceldsp.htm.

[2] Agility Compiler - Manual. "Available at: http://ic.engin.brown.edu/classes/EN2911XF07/agility_manual.pdf".

[3] An independent evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool by the Berkeley Design Technology, Inc. http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf.

[4] C ILK-5.3 reference manual. Supercomputing Technologies Group (2000).

[5] C-to-Silicon Compiler - Home Page. http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx.

[6] C to Verilog - Compile Your C code into Verilog. http://c-to-verilog.com/.

[7] CatapultC - Home Page. http://calypto.com/en/products/catapult/overview.

[8] Compaan - heterogeneous compilation. http://www.compaandesign.com/.

[9] CUDA Samples - CUDA Toolkit v5.5. http://developer.nvidia.com/cuda-cc-sdk-code-samples.

[10] "FPFA CPLD and ASIC from Altera. http://www.altera.com.

[11] GCC - GNU Compiler Collection. http://gcc.gnu.org.

[12] High-Level Synthesis with Legup. http://legup.eecg.utoronto.ca.

[13] "Icarus Verilog". http://iverilog.icarus.com.

[14] Impulse CoDeveloper - Home Page. http://www.impulseaccelerated.com/products.htm.

[15] "Intel's Threading Building Blocks Tutorial, http://www.threadingbuildingblocks.org/documentation.php.

[16] "Modelsim - Advanced Simulation and Debugging, http://model.com/.

[17] MPI: A Message Passing Interface Standard Version 2.2. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[18] P. Briggs. Parallelization for LLVM. http://www.montblanc-project.eu/.

[19] PandA: A framework for Hardware-Software Co-Design of Embedded Systems. http://panda.dei.polimi.it/.

[20] "Partitioned Global Address Space (PGAS) home page". "http://www.pgas.org/".

[21] Pugh, W.: Java Memory Model and Thread Specification Revision (2004) JSR 133. http://www.jcp.org/en/jsr/detail?id=133.

[22] ROCCC - Riverside Optimizing Compiler for Configurable Computing. http://www.jacquardcomputing.com/roccc/.

[23] Synphony C Compiler. Available at: http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx.

[24] The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi.

[25] The MontBlanc Project. Available at: https://sites.google.com/site/parallelizationforllvm/.

[26] The OpenMP API specification for Parallel Programming. http://www.openmp.org.

[27] Xilinx BlueSpec - Home Page. Available at: http://www.xilinx.com/products/design_tools/logic_design/.

[28] "Xilinx ISim ver. 12.3 user guide". Available at http://www.xilinx.com.

[29] "Xilinx. Synthesis tools for FPGA devices. http://www.xilinx.com.

[30] xmpipp - Parallel Programming Tutorial. http://xmipp.cnb.csic.es/twiki/bin/view/Xmipp/ParallelProgramming.

[31] A. Marongiu and L. Benini. An openmp compiler for efficient use of distributed scratchpad memory in mpsocs. *IEEE Trans. Computers*, 61(2):222–236, 2012.

[32] S. Amellal and B. Kaminska. "Scheduling of a Control Data Flow Graph". In *IEEE Int.Symposium on Circuits and Systems*, volume 3, pages 1666–1669, May 1993.

[33] C. Andriamisaina, P. Coussy, E. Casseau, and C. Chavet. "High-level synthesis for designing multimode architectures". *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29:1736–1749, November 2010.

[34] R. Bergamaschi. "Behavioral network graph unifying the domains of high-level and logic synthesis". In *Proceedings of the 36th Design Automation Conference*, pages 213–218, 1999.

[35] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[36] D. Bonachea and J. Jeong. GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. *University of California-Berkeley Technical Report.*, 2002.

[37] K. Bosschere, W. Luk, X. Martorell, N. Navarro, M. O'Boyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenström, and O. Temam. Transactions on high-performance embedded architectures and compilers i. chapter High-Performance Embedded Architecture and Compilation Roadmap, pages 5–29. Springer-Verlag, Berlin, Heidelberg, 2007.

[38] B. Brock and K. Rajamani. "Dynamic Power Management for Embedded Systems". In *Proceedings of IEEE International Conference on Systems-on-Chip (SOC)*, volume 54, pages 416–419, September 2003.

[39] E. Brooks and K. Warren. Development and evaluation of an efficient parallel programming methodology, spanning uniprocessor, symmetric shared-memory multi-processor, and distributed-memory massively parallel architectures,. *in Proc. of poster session at SuperComputing*, pages 3–8, 1995.

[40] C. Pilato, V. G. Castellana, S. Lovergine, and F. Ferrandi. "a runtime adaptive controller for supporting hardware components with variable latency". In *in Proceedings of International NASA/ESA Conference on Adaptive Hardware and Systems*, pages 153–160, 2011.

[41] R. Camposano and W. Wolf. *"High-Level VLSI Synthesis"*. Kluwer, Dordrecht, 1991.

[42] W. Carlson, El-Ghazawi, B. T., Numrich, and K. Yelick. Programming in the partitioned global address space model. *Presentation at SC 2003, available at http://www.gwu.edu/upc/tutorials.html*, 2003.

[43] W. W. Carlson and J. M. Draper. Distributed data access in ac. *SIGPLAN Not.*, 30(8):39–47, Aug. 1995.

[44] M. Cassel and F. L. Kastensmidt. "Evaluating One-Hot Encoding Finite State Machines for SEU Reliability in SRAM-based FPGAs". *IEEE International On-Line Testing Symposium*, 0:139–144, 2006.

[45] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. *PGAS '10: the Fourth Conference on Partitioned Global Address Space Programming Model*, 2(3):1–2, 2010.

[46] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[47] A. E. Charlesworth. An approach to scientific array processing: the architectural design of the ap-120b/fps-164 family. *Computer*, 14(9):18–27, 1981.

[48] Y. Chen and Y. Xie. "Tolerating process variations in high-level synthesis using transparent latches". In *Asia and South Pacific Design Automation Conference, ASP-DAC '09.*, pages 73–78, Jan. 2009.

[49] Chris Papachristou and Yusuf Alzazeri. "A method of distributed controller design for RTL circuits". *Proc. of Design Automation and Test in Europe*, 1999.

[50] L. Codrescu, D. S. Wills, S. Member, and J. Meindl. Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE Transactions on Computers*, 50:67–82, 1999.

[51] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *Design Test of Computers, IEEE*, 26(4):8–17, jul. 2009.

[52] P. Coussy and A. Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[53] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing '93. Proceedings*, pages 262–273, 1993.

[54] D. Kaeli and D. Akodes. "The Convergence of HPC and Embedded Systems in our Heterogeneous Computing Future". *in Proceedings of International Conference on Computer Design*, pages 9–11, 2011.

[55] P. D'Alberto, A. Nicolau, A. Veidenbaum, and R. Gupta. "Line Size Adaptivity Analysis of Parameterized Loop Nests for Direct Mapped Data Cache". In *IEEE Transaction on Computers*, volume 54, num. 2, pages 185–197, February 2005.

[56] G. De Micheli. *"Synthesis and Optimization of Digital Circuits"*. McGraw-Hill, 1994.

[57] A. Del Barrio, S. Memik, M. Molina, J. Mendias, and R. Hermida. A distributed controller for managing speculative functional units in high level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 30(3):350–363, march 2011.

[58] R. Diaconescu and H. Zima. An approach to data distributions in chapel. *Int. J. High Perform. Comput. Appl.*, 21(3):313–335, Aug. 2007.

[59] T. El-Ghazawi, W. Carlson, and J. Draper. Upc language specification v1.1.1. *Technical report, George Washington University*, 2003.

[60] R. W. et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

[61] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. *in Proc. of the 2nd Conference on Computing Frontiers*, pages 28–34, 2005.

[62] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. volume 9, pages 319–349, New York, NY, USA, July 1987. ACM.

[63] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[64] G. Bell, T. Hey, and A. Szalay. "Beyond the Data Deluge". *International Journal on Science*, 323(5919):1297–1298, March 2009.

[65] G. De Micheli. *"Synthesis and Optimization of Digital Circuits"*. McGraw-Hill, New York, 1994.

[66] G. Deconinck, V. De Florio, T. Varvarigou, and E. Verentziotis. "The EFTOS approach to dependability in embedded supercomputing". In *IEEE Transactions on Reliability*, volume 51, pages 76–90, 2002.

[67] G. Lakshminarayana, K.S. Khouri, and N.K. Jha. "Wavesched: a novel scheduling technique for control-flow intensive designs". *IEEE Trans. on CAD*, 18(5):505–523, May 1999.

[68] H. Gadke-Lutjens, B. Thielmann, and A. Koch. "A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation". *International Conference on Field Programmable Logic and Applications*, pages 475–482, 2010.

[69] D. Gajski et al. *"High Level Synthesis: An Introduction to Chip and System Design"*. Kluwer, 1992.

[70] V. Getov, A. Hoisie, and H. J. Wasserman. "Codesign for Systems and Applications: Charting the Path to Exascale Computing". In *International Journal on Computer*, volume 44, num. 11, pages 19–21, November 2011.

[71] M. Girkar and C. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22:519–551, 1994.

[72] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[73] E. Gutiérrez, R. Asenjo, O. Plata, and E. Zapata. Automatic parallelization of irregular applications. *Parallel Computing*, 26(13âĂŞ14):1709 – 1738, 2000. <ce:title>Parallel Computing for Irregular Applications</ce:title>.

[74] E. Gutiérrez, R. Asenjo, O. Plata, and E. L. Zapata. Automatic parallelization of irregular applications. *Parallel Comput.*, 26(13-14):1709–1738, Dec. 2000.

[75] H. Tomiyama, S. Honda, Y. Hara, and H. Takada. "Proposal and Quantitative Analysis of the chstone Benchmark Program Suite for Practical C-based High-Level Synthesis". *Journal of Information Processing*, pages 242–254, 2009.

[76] P. Hansen. Structured multiprogramming. *CACM*, 15, 1972.

[77] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, Oct 2003.

[78] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct 1974.

[79] C. J. and Z. Z. An efficient and versatile scheduling algorithm based on sdc formulation. *In Proc. of the 43rd Design Automation Conference ACM/IEEE*, pages 433–438, 2006.

[80] G. Jin, J. Mellor-Crummey, L. Adhianto, W. N. Scherer, and C. Yang. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. *IPDPS 2011: IEEE International Parallel & Distributed Processing Symposium*, pages 1089–1100, 2011.

[81] João M. Fernandes, M. Adamski, and A.J. Proença. "VHDL generation from hierarchical petri net specifications of parallel controllers". *Proc. of IEEE Computer and Digital Techniques*, 1997.

[82] K. Bilinski, E. Dagless, and J. Mirkowski. "Synchronous parallel controller synthesis from behavioural multiple-process VHDL description". *Proc. of European Design Automation Conference*, Sept. 1996.

[83] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[84] K. Kerr. "Visual C++ 2010 And The Parallel Patterns Library". *MSDN Magazine*, February 2009.

[85] O. Konè, C. Artigues, P. Lopez, and M. Mongeau. Comparison of mixed integer linear programming models for the resource-constrained project scheduling problem with consumption and production of resources. *Journal of Flexible Services and Manufacturing*, 25:25–47, 2013.

[86] A. A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Trans. Design Autom. Electr. Syst.*, 7(3):380–412, 2002.

[87] Krzysztof Biliński, E.L. Dagless, J.M. Saul, and M. Adamski. "Parallel controller synthesis from a Petri net specification". *Proc. of European Design Automation Conference*, 1994.

[88] D. Ku and G. De Micheli. Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 11(6):696–718, jun 1992.

[89] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? *SIGPLAN Not.*, 44(4):3–14, Feb. 2009.

[90] L. Oliker. "Green Flash: Designing an Energy Efficient Climate Supercomputer". *in Proeedings of International Parallel and Distributed Processing Symposium*, 0:1–1, 2009.

[91] B. Lam, A. George, and H.Lam. An introduction to TSHMEM for Shared-Memory Parallel Computing on Tilera Many-Core Processors. *PGAS '10: the Fourth Conference on Partitioned Global Address Space Programming Model*, 2012.

[92] D. Leijen, W. Schulte, and S. Burkhardt. "The Design of a Task Parallel Library". *Proceedings of the 24th ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2009.

[93] Y. Li", T. Callahan", E. Darnell", R. Harr", U. Kurkure", and J. Stockwood". "hardware-software co-design of embedded reconfigurable architectures". In *Proceedings of the 37th Annual Design Automation Conference (DAC)*, pages 507–512, 2000.

[94] D. B. Liang Chen and Y. Lin. "MCFX: A New Parallel Programming Framework for multicore systems". *Proceedings of International Supercomputing Conference*, 2009.

[95] S. Lovergine and F. Ferrandi. Harnessing adaptivity analysis for the automatic design of efficient embedded and hpc systems. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 2298–2301, Boston, Massachusetts, USA, 2013. IEEE Computer Society.

[96] M. Girkar and C. D. Polychronopoulos. "Automatic Extraction of Functional Parallelism from Ordinary Programs". *IEEE Transaction on Parallel Distributed Systems*, 3:166–178, 1992.

[97] M. Hilbert and P. Lopez. "Introduction to special issue: Challenges and opportunities.". *Staff of Science*, 331(6018):692–693, 2011.

[98] M. Hilbert and P. Lopez. "The World's Technological Capacity to Store, Communicate, and Compute Information.". *Science*, 332(6025):60–65, 2011.

[99] M. Kaku. "Tweaking Moore's Law: Computers of the Post-Silicon Era. Big Think.". "http://bigthink.com/ideas/42825", 2012.

[100] M. Kanellos. "Intel scientists find wall for Moore's Law". December 2003.

[101] M. Meredith. "a look inside behavioral synthesis", 2004.

[102] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[103] M. McFarland and A. Camposano. in Proc. of the 25th Design Automation Conference. *Tutorial on High-Level Synthesis*, pages 330–336, Jul. 1988.

[104] J. Moreira. "On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors". *PhD. thesis, Univ. of Illinois at Urbana-Champaign*, 1995.

[105] J. E. Moreira, S. P. Midkiff, and M. Gupta. A comparison of three approaches to language, compiler, and library support for multidimensional arrays in java. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, JGI '01, pages 116–125, New York, NY, USA, 2001. ACM.

[106] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high-performance numerical computing, 2000.

[107] D. Naishlos, J. Nuzman, C. W. Tseng, and U. Vishkin. Evaluating Multithreading in the Prototype XMT Environment. *in Proc. 4th Workshop on Multi-Threaded Execution, Architecture and Compliation*, 2000.

[108] D. Naishlos, J. Nuzman, C. W. Tseng, and U. Vishkin. Evaluating the XMT Parallel Programming Model. *in Proc. of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 96–108, 2001.

[109] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, 2006.

[110] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. Panda. High Performance Remote Memory Access Communication: The ARMCI Approach.

[111] M. Nourani, C. Papachristou, and Y. Takefuji. A neural network based algorithm for the scheduling problem in high-level synthesis. In *Design Automation Conference, 1992., EURO-VHDL '92, EURO-DAC '92. European*, pages 341–346, 1992.

[112] D. Novillo. "Memory SSA- A Unified Approach for Sparsely Representing Memory Operations". *in Proc of the GCC Developers' Summit*, 2007.

[113] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[114] P. Ashar, S. Devadas, and R. Newton. "Optimum and heuristic algorithms for an approach to finite state machine decomposition". *IEEE Trans. on CAD*, 10:296–310, 1991.

[115] P. B. Bhat, Y. W. Lim, and V. K. Prasanna. "issues in using heterogeneous HPC systems for embedded real time signal processing applications". In *in Proceedings of International Workshop on Real-Time Computing Systems and Applications*, 1995.

[116] e. a. P. G. Lowney. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1/2):51–142, May 1993.

[117] P. Lyman and H. R. Varian. "how much information?, university of california at berkeley, research report". Technical report, 2003.

[118] I.-C. Park and C.-M. Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 680–685, New York, NY, USA, 1991. ACM.

[119] P. Paulin and J. Knight. Force-directed scheduling in automatic data path synthesis. *in Proc, of the 24th International Design Automation Conference*, pages 195–202, Jul 1987.

[120] P. Paulin and J. Knight. Scheduling and binding algorithms for high-level synthesis. In *Design Automation, 1989. 26th Conference on*, pages 1–6, 1989.

[121] K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-lojo, D. Prountzos, X. Sui, and Z. Zhong. Amorphous data-parallelism in irregular algorithms âĹŮ.

[122] C. D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the 5th international conference on Supercomputing*, ICS '91, pages 252–263, New York, NY, USA, 1991. ACM.

[123] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *"Digital Integrated Circuits"*. 2008.

[124] M. Rajagopalan and V. H. Allan. Efficient scheduling of fine-grain parallelism in loops. *in Proc. 26th International Symposium on Microarchitecture (Austin, Texas)*, pages 2–11, December 1993.

[125] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM.

[126] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *in Proc. 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[127] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schemas for modulo scheduled loops. *in Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon)*, pages 158–169, December 1992.

[128] C. Rice University. High performance fortran language specification. *SIGPLAN Fortran Forum*, 12(4):1–86, Dec. 1993.

[129] S. Lovergine, A. Tumeo, O. Villa, and F. Ferrandi. "Parallelizing Irregular Applications through the YAPPA Compilation Framework". In *poster at SuperComputing (SC13)*, 2013.

[130] S. Lovergine, A. Tumeo, O. Villa, and F. Ferrandi. "YAPPA: a Compiler-Based Parallelization Framework for Irregular Applications on MPSoCs". In *in Proceedings of International Symposium on Rapid System Prototyping*, pages 123–129, 2013.

[131] S. Lovergine and F. Ferrandi. "Instructions Activating Conditions for Hardware-Based Auto-Scheduling". In *in Proceedings of International Conference on Computing Frontiers*, pages 253–256, 2012.

[132] S. Lovergine and F. Ferrandi. "Dynamic AC-Scheduling for Hardware Cores with Unknown and Uncertain Information". In *in Proceedings of 31st IEEE International Conference on Computer Design*, pages 475–478, 2013.

[133] V. Saraswat. Report on the experimental language x10, v0.41. Technical report, IBM Research, 2005.

[134] S.Devadas and R. Newton. "Decomposition and Factorization of Sequential Finite State Machines". *IEEE Trans. on CAD*, 8:1206–1217, 1988.

[135] A. Srivastava, D. Sylvester, and D. Blaauw. *"Statistical Analysis and Optimization for VLSI: Timing and Power"*. Springer, 2005.

[136] U. R. S. V. H. Allan and K. M. Reddy. Petri net versus modulo scheduling for software pipelining. *in Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan)*, November 1995.

[137] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit Multi-threaded (XMT) Bridging Models for Instruction Parallelism. *in Proc. 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 140–151, 1998.

[138] e. a. W. W. Hwu. The superblock: an effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7(1/2):229–248, May 1993.

[139] F. Wang, G. Sun, and Y. Xie. "A Variation Aware High Level Synthesis Framework". In *Design, Automation and Test in Europe, DATE '08*, pages 1063–1068, March 2008.

[140] William Carlson and Tarek El-Ghazawi and Bob Numrich and Kathy Yelick. "Programming in the Partitioned Global Address Space Model", 2003.

[141] Xilinx Vivado Design Suite. Home Page. http://www.xilinx.com/products/design-tools/vivado/.

[142] Y. K. Chen and S. Y. Kung. "Trend and Challenge on System-on-a-Chip Designs". *J. Signal Process. Syst.*, 53(1-2):217–229, 2008.

[143] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[144] K. A. Yelick. "programming models for irregular applications". *in Proc. of Workshop on languages, compilers and run-time environments for distributed memory multiprocessors*, 28:28–31, 1993.

[145] J. Zhu and D. D. Gajski. "A unified formal model of ISA and FSMD". In *Proceedings of the seventh international workshop on Hardware/software codesign*, CODES '99, pages 121–125, New York, NY, USA, 1999. ACM.