Low power application architecture adaptation using SMT solvers

Émilien Kofman

▶ To cite this version:

Émilien Kofman. Low power application architecture adaptation using SMT solvers. Other [cs.OH]. COMUE Université Côte d'Azur (2015 - 2019), 2017. English. NNT : 2017AZUR4009 . tel-01534440

HAL Id: tel-01534440
https://theses.hal.science/tel-01534440v1

Submitted on 7 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

T H È S E

pour l'obtention du grade de docteur en sciences

de l'Université Côte d'Azur Mention : **Informatique**

présentée et soutenue par

Émilien Kofman

Low Power Application Architecture Adaptation using SMT solvers

Adéquation Algorithme Architecture automatisée par solveur SMT

Dirigée par: Robert de Simone et François Verdier

Soutenue le 27 Février 2017 à Sophia Antipolis
Devant un jury composé de:

Mme Florence Maraninchi	Prof.	INP Grenoble	*Rapporteur*
M Xavier Thirioux	MCF	ENSEEIHT	*Rapporteur*
M Rolf Drechsler	Prof.	DFKI Bremen	*Examinateur*
M Dumitru Potop-Butucaru	CR	INRIA Paris	*Examinateur*
M Robert de Simone	DR	INRIA Sophia	*Directeur*
M François Verdier	Prof.	UNS	*Directeur*

Contents

1 Background on model-based, platform-based design and AAA **15**
1.1 Platform based design . . . 15
1.1.1 The adequation problem . . . 16
1.1.2 Motivations to the use of models . . . 16
1.2 Application modeling . . . 16
1.2.1 Jobs, Agents, Processes and Tasks . . . 17
1.2.2 Graph models . . . 18
1.2.2.1 Dataflow Process Networks . . . 18
1.2.2.2 Structured graphs . . . 19
1.2.2.3 Conflict freeness / functional determinism / latency insensitivity . . . 20
1.2.3 Relations to general purpose languages . . . 22
1.3 Application requirement modeling . . . 23
1.3.1 Preemption and migration . . . 23
1.3.1.1 non-preemptive task models . . . 23
1.3.1.2 Preemptive task models . . . 23
1.3.2 Periodicity . . . 24
1.3.3 Objective functions . . . 25
1.4 Architecture modeling . . . 25
1.4.1 Architecture modeling in real-time theory . . . 25
1.4.1.1 Resources . . . 26
1.4.1.2 Communications . . . 26
1.4.2 Computer architecture classification in practice . . . 26
1.4.2.1 Resources . . . 27
1.4.2.2 Communications . . . 27
1.4.2.3 Extracting some knowledge . . . 28
1.4.3 Power saving mechanisms . . . 29
1.4.3.1 Dynamic Voltage and Frequency Scaling . . . 29
1.4.3.2 Power gating . . . 30
1.4.3.3 Motivating example . . . 31
1.4.4 Temperature modeling . . . 31
1.5 Existing approaches to AAA . . . 32
1.5.1 Discrete Event models and test by simulation . . . 32
1.5.2 Analytic approaches . . . 32
1.5.3 Heuristics . . . 33
1.5.4 Energy concerns . . . 34
1.5.5 Structure and syntax of the result . . . 34
1.6 Automated reasoning and solvers . . . 35

1.6.1 Computer algebra systems . . . 35
1.6.2 Solvers and methods for categorized mathematical problems . . . 35
1.6.3 Using SMT solvers for automated adequation . . . 36
1.6.3.1 Linear programming aspects of AAA . . . 36
1.6.3.2 Combinatorial optimization aspects of AAA . . . 37
1.7 Related work . . . 37
1.7.1 Application modeling . . . 37
1.7.2 Architecture modeling . . . 38
1.7.3 Provisions modeling . . . 39
1.7.4 Solving the adequation problem . . . 40
1.8 Our contributions . . . 40
1.8.1 The SymSched tool, a general presentation . . . 41
1.8.2 A running example . . . 41

2 SymSched modeling and model translation to system of constraints 45
2.1 SymSched application model . . . 47
2.1.1 SymSched task graph meta models . . . 47
2.1.1.1 Agents and Tasks . . . 47
2.1.1.2 Places and Messages . . . 48
2.1.1.3 Loop constructs . . . 48
2.1.1.4 Parallel agents . . . 50
2.1.1.5 Preemptable agents . . . 50
2.1.2 Requirements . . . 51
2.1.2.1 Periodicity . . . 51
2.1.2.2 Objective functions . . . 51
2.1.3 Application transformation to a solver input . . . 51
2.2 SymSched Architecture model . . . 53
2.2.1 Processing resources . . . 53
2.2.2 Communication resources . . . 54
2.2.3 Provisions . . . 56
2.2.3.1 Computing and communication throughput . . . 56
2.2.3.2 Power Domains . . . 57
2.2.3.3 Layout and temperature issues . . . 59
2.2.4 Architecture transformation to a solver input . . . 60
2.3 SymSched Adequation model . . . 60
2.3.1 Adequation costs (solver input) . . . 61
2.3.2 Orthogonality and composability of the models . . . 62
2.3.3 Mapping representation (solver output) . . . 63
2.3.4 Comments on expressiveness . . . 64
2.3.4.1 Limitations of the SDF model . . . 64
2.3.4.2 Data parallelism . . . 65
2.3.4.3 Pre-calculated preemption . . . 66
2.3.4.4 Periodicity . . . 66
2.3.4.5 Combining agent models and requirements . . . 68

3 Using solvers for AAA: lessons learned 69
3.1 Practical aspects of the modeling . . . 69
3.1.1 Time discretization . . . 70
3.1.2 Modeling cyber physical systems . . . 71
3.2 Scalability . . . 71
3.2.1 Independent tasks . . . 72
3.2.2 Preemptable agents . . . 74
3.2.3 Parallel agents . . . 74
3.2.4 Optimization and convergence speed . . . 74
3.3 Symmetry breaking . . . 76
3.3.1 The Platooning application . . . 76
3.3.1.1 Symmetries in the application model . . . 77
3.3.1.2 Symmetries in the architecture model . . . 78
3.4 Realistic use cases . . . 79
3.4.1 Fast Fourier Transform application . . . 79
3.4.2 Radar application . . . 80
3.5 Real use-case . . . 81
3.5.1 Cholesky matrix decomposition . . . 81
3.5.2 Experimental setup: The ARM big.LITTLE platform . . . 82
3.5.3 Practical aspects of meta programming . . . 83
3.5.3.1 Kernel cost estimations . . . 83
3.5.3.2 Controlling core pinning and core frequencies . . . 84
3.5.4 Results . . . 84
3.5.4.1 Multiobjective optimization . . . 85
3.5.4.2 Makespan optimization . . . 85
3.5.4.3 Energy optimization . . . 86
3.5.4.4 Further scalability results . . . 86
3.6 Summary on the SymSched approach . . . 88
3.7 Conclusion and future work . . . 89

Appendices 91

A Code samples 93
A.1 Stress test examples . . . 93
A.2 Calls to the GNU Scientific library . . . 93
A.3 Calls to the perf API . . . 93
A.3.1 Setting up performance counters on Odroid XU3 . . . 93
A.3.2 Using the C perf API . . . 94
A.4 A Proof of concept using z3 . . . 94
A.5 Generated constraints Cholesky big.LITTLE 1+1, 3 tiles . . . 95
A.6 Solver logs and relevant tactics descriptions . . . 96

B Application graphs and gantt diagrams 97
B.1 Tiled cholesky . . . 97
B.2 Radar application . . . 98

Résumé

Nous décrivons la méthode Symsched et l'environnement de conception AAA (Adéquation Algorithme Architecture). Cette méthodologie permet d'évaluer les compromis énergie/performance pour un système embarqué. Nous transformons les composants du problème (exigences de l'application et capacités de l'architecture) en un système d'équations et inéquations sur des variables entières pour la modélisation des aspects temporels et sur des variables booléennes pour modéliser les alternative de placement des tâches et de niveau de performance des ressources. Le modèle mathématique généré est ensuite soumis à un solveur SMT (SAT Modulo Theories). Nous étudions le passage à l'échelle de cette approche et ses compromis avec l'expressivité des modèles. Enfin nous appliquons ces outils à des problèmes d'ordonnancement sur des cas synthétiques, réalistes et réels.

Ce travail a bénéficié d'une aide de l'Etat gérée par l'Agence Nationale de la Recherche au titre du programme "Investissements d'Avenir" portant la référence : ANR-11-LABX-0031-01.

Abstract

We describe the Symsched methodology and environment for AAA design (Application Architecture Adequation). It allows to evaluate the energy/performance balance for a given embedded system. We translate the different components of the problem (application requirements et architecture provisions) in a system of equations and inequations made of integer variables for the modeling of temporal aspects and boolean variables for the modeling of admissible task mapping and resource states. We then submit this problem to an automatic search engine SMT solver (SAT Modulo Theories). We study the scalability of this methodology and its compromises with models expressiveness. We then study synthetic, realistic and real scheduling problems using this approach.

This work was partly funded by the French Government (National Research Agency, ANR) through the "Investments for the Future" Program reference ANR-11-LABX-0031-01.

Remerciements

Ce travail n'aurait pas été accompli sans le support de ma famille, mes amis et mes collègues. J'essaye dans ces quelques lignes de leur exprimer ma gratitude.

Je remercie mon directeur Robert de Simone pour sa présence, sa confiance et sa motivation, mais également son sens de l'humour sans limite. J'ai par ailleurs apprécié les remarques de mon co-directeur François Verdier et la flexibilité qu'il m'a accordé ces trois dernières années. Je remercie Florence Maraninchi et Xavier Thirioux pour avoir accepté de relire mon manuscrit et d'assister à la soutenance. Merci également à Rolf Drechsler pour avoir accepté de faire partie de mon jury. J'ai également beaucoup apprécié les discussions que nous avons eu avec Dumitru Potop-Butucaru pendant ma troisième année et je suis heureux qu'il soit venu examiner ma soutenance.

Bien entendu je n'oublie pas les collègues de l'équipe AOSTE impliqués dans cette aventure. Je pense tout d'abord à Jean-Vivien Millo que je remercie pour m'avoir très largement mis sur les rails avec une énergie débordante, malgré nos nombreux désaccords et discussions. Je pense aussi à Julien Deantoni avec qui nous avons beaucoup échangé sur des sujets scientifiques au delà de notre domaine d'investigation habituel, parfois simplement par curiosité scientifique. Merci aux autres collègues de l'équipe (y compris les éphémères) pour leur bonne humeur et leur convivialité. Pendant ces trois ans à l'INRIA j'ai également eu l'opportunité d'enseigner à l'école Polytech'Nice sous la direction de Sébastien Bilavarn. Malgré mon anxiété initiale, Sebastien en a fait une expérience agréable et très enrichissante.

Il y a un temps indispensable en dehors du labo que j'ai pris beaucoup de plaisir à partager avec mes amis et ma famille. Merci à Jonathan Viquerat pour avoir supporté mes hauts, mes bas, et les transitions. Je remercie également Camille Sulkowski pour m'avoir fait partager avec lui des moments chaleureux. J'ai également une pensée pour mes amis de l'ASSA et les souvenirs forgés dans nos aventures verticales.

Les dernières lignes et non les moindres iront à ma famille. Merci beaucoup à ma mère et mon frère de supporter mes humeurs changeantes, mes lubies et mes bêtises depuis 26 ans. Leur soutien en particulier ces trois dernières années aura été d'une aide inestimable.

Introduction

Embedded systems including consumer electronic devices are now very complex systems that may embed several subsystems and resources. For instance a smartphone is a modem, a camera, a video decoder, a GPS planer and much more. Those different resources have to synchronize, exchange data, and compute with very high data throughput. In the area of embedded systems the energy is limited, since the system is by definition not plugged into any outlet.

On the other hand, such devices are pushed to the extreme of decoding an ever increasing number of pixels in images and videos, running voice decoding, object detection, movement detection and signal processing applications, sometimes multitasking among those computationally intensive workloads. In order to satisfy the user, system architects must always offer more computing power.

The solution to raise the computing power of a machine has long been to run it faster, which in the computer science field consists in raising its frequency. It is well understood now that raising the frequency also increases power consumption roughly to the square, which raises also the temperature of the devices since this power is dissipated mainly into heat. In some domains of practical computer science (such as HPC) much effort is conducted towards cooling which is much harder in the area of embedded systems due to the physical limitations.

Another solution to raise the computing power is to distribute the work to multiple computing units. The recent and smaller transistors (expected 10nm in 2017) allow to cluster multiple resources in the same System on Chip (SoC). As a consequence, a typical smartphone CPU is not a single core system anymore (DSPs and GPUs are also often included). Those new upcoming architectures raise multiple issues that are still open research areas. Smaller transistors behave differently regarding energy consumption: recent studies show that leakage current is much higher with small transistors. As a consequence it needs to dissipate more power or to run at a lower frequency. Since the form factor of an embedded system limits the power dissipation, the latter is preffered. Furthermore, those multiple resources have to communicate in an efficient and predictable way. The typical unified and uniform memory architecture will not scale. System of chip architects still face the energy vs computing power compromise, which results in very complex, sometimes heterogeneous asymmetrical micro-architectures.

Various hardware architectures exist: embedded GPUs, heterogeneous multicores, networks on chip, and still no de-facto standard emerged. Those upcoming architectures will thus require different programming models, as the classical sequential languages does not allow to efficiently distribute the workload. Meta-programming is underused in the field of embedded systems (and HPC) although it is used efficiently in other fields such as web-design. This motivates us to investigate which models and which task scheduling and allocation policies can fit those new models.

In a typical server, desktop computer system or consumer electronics system such as smartphones and tablets, a user interacts with the device which can be modeled with sporadic job appearance. Deciding where and when to run those jobs has to be done while the system is running. This is commonly known as “dynamic or online scheduling”. On contrary some usage of those devices (such

as watching a video) does not involve user interaction as well as other embedded applications such as radars. In this case the system of jobs and resources can be analyzed at compile time, which is known as "static or offline scheduling". Because most often, very little is known about the work-load in dynamic scheduling problem, most of the platform-based design approaches focus on the latter.

The rest of this document is organized as follows:

- In the first Chapter we recall the fundamentals of model-based, platform-based design, and the Application-Architecture Adaptation (or Algorithm-Architecture Adequation) approach. We recall a number of modeling concepts involved. We do not seek here exhaustivity, but rather focus on proximity, and the description of previous efforts that were influential on our work. We end this chapter with a short, rather informal introduction to our SymSched formalism, so as to position it amongst these former references. We also introduce on an illustrative example how low-power modeling and low-power architectures raise specific issues for AAA.
- Chapter 2 is devoted to a presentation of our various models covering all aspects involved in the adequation mapping, and their (sketchy) metamodels (*applications, application requirements, architectures, architecture provisions, costs, mappings,...*) . We also describe how these models translate into formulas that will be later input to automatic solvers. In addition, we comment at places on how this global objective of producing such constraint sets of equations led us to restrict or bend the modeling styles at all levels, while keeping in mind the concern for scalability of descriptions and tractability of analysis.
- in Chapter 3 we report on how automatic approaches (mostly SMT solvers) performed for this adequation mapping assignment. We first deal with simple, rather synthetic problems, in order to better tune modeling and analysis regarding complexity. We then show the actual feasability of our approach on non-trivial examples. We consider a number of modeling extensions and/or restrictions motivated by concrete concerns (to match existing architectures), some of them still unexplored. We conlcude with a critical assessment on our results and the projections that could be drawn from then.
- The conclusion section, which also points at several remaining challenges in the definition of proper adequation methods and use of automatic solving techniques, is followed in annex by material which could not fit into the main document body (large examples, interactive solver sesssions,...)

Chapter 1

Background on model-based, platform-based design and AAA

In this chapter we recall the basics of Platform based design and Application Architecture Adequation [1] (AAA, Section 1.1). In Sections 1.2 and 1.4 we address application and architecture modeling in different fields of computer science, from theoretical scheduling to more practical programming models. Then we give an overview of the existing approaches for solving the adequation problem in Sections 1.7 and 1.5 and we detail our contributions in Section 1.8.

1.1 Platform based design

Platform based design (or Y-chart design) is the general problem of obtaining a system implementation which satisfies a set of constraints and sometimes optimizes one or multiple criteria [82], this can also include hardware synthesis [44, 6, 4, 26]. This hardware and/or software synthesis is performed based on high level (most often abstract) "components" that first need to be associated with given metrics, such that the search can actually be performed. We usually call software components the tasks, and hardware components the resources. The analysis is a high level, not cycle accurate estimation, often performed on abstract models: it does not consider implementations of the components but only some of their characteristics. It is then the responsibility of the designer to make sure that the implementation of those components will perform as expected.

In the hardware synthesis community, exhaustive verification is not often feasible and the validation is most often a simulation of random or corner cases. Platform architects sometimes refer to a similar workflow (Figure 1.2) even though following the legacy practices of hardware design they do not consider the search step and most often perform simulation, using refined hardware components.

In the software engineering community, modeling software with a graph of components is not the mainstream technique. Classical approaches do not consider any architecture model neither, the programmer must be aware of it and must decide how to use it properly. Programmers put a strong focus on the functional aspect of the application and consider that an implementation is one more layer on top of a well separated stack of software layers. On the other side the communities of real-time scheduling, program optimization and parallel compilation study structured models of the software such as activity diagrams, task graphs, state machines, or abstract interpretations of existing software. It is thus an essential concern to bring together theoretical aspects of the latter with the

[1] The methods advocating a global system view to the codesign of software and hardware, mainly in the area of embedded applications and architectures

technical aspects of the former. It requires first to identify the modeling granularity to extract a set of components, evaluate them according to relevant metrics, and finally connect or arrange them to achieve a given objective.

1.1.1 The adequation problem

Adequation mostly focuses on adapting algorithms to best fit them on existing architectures although the same models of components is used to describe both architecture and application model. The result of the adequation is an optimized (or valid according to application requirements) task distribution in space (on the different resources) and time. Adequation problems do not consider that the designer is able to add or remove resources, or to connect them differently. Because the target, and thus the number and nature of resources is not any more a variable of the problem, the adequation is a sub-problem of platform based design. Since the architectures must also be polyvalent because modifying it costs much more than modifying the software, adequation still remains an essential concern.

We call “allocation” the spatial distribution of the tasks on the resources, and we call “sequencing” the problem of sharing the available resource time in order to fit the allocated tasks. Finally, we call “adequation”, or “scheduling” the overall problem of allocation and sequencing. We call “schedule” the result of a static scheduler.

1.1.2 Motivations to the use of models

High level abstractions have long been considered a reason of impaired performance. Recent approaches to software optimization on specific problems such as FFT optimization [78] show on contrary that software optimization is not any more limited to low level implementations. Those high level approaches are welcome since low level programming is cumbersome on cutting edge complex systems. Moreover in the area of embedded and real-time systems, the specification often comes with real-time requirements and obtaining a valid implementation according to the specifications sometimes requires an optimization effort. Modeling and verification is thus a mandatory step towards a predictable and functional implementation.

The separation of concerns in the modeling approach joins Model Driven Engineering ideas [83] with a strong focus on re-usability: although the Y-chart approach strictly focuses on software generation, using models and taking apart the application implementation from its target(s) brings portability. Figure 1.1 shows the classical V-shape project development curve with a traditional approach and with an MDE approach. In the last ten years, Model driven Engineering techniques have been used in other fields such as web design with great results. It is however underused in the domains of real-time and embedded systems.

The separation of concerns also allows to run analysis separately on both models, sometimes indicating that further implementation work is not desirable since for instance the application model alone already needs to be improved (or fixed, see Section 1.2.2.3). In the case of dataflow graphs models it is usual to check that the application graph contains no deadlock and has a sufficient maximum throughput before considering any implementation generation regarding the architecture model.

1.2 Application modeling

The keys aspects of application concurrency modeling consists in being expressive enough to model different type of parallelism (task parallelism, data parallelism, and pipeline parallelism) including the ability to combine them hierarchically and sequentially and yet be simple enough to allow static analysis or automated adequation.

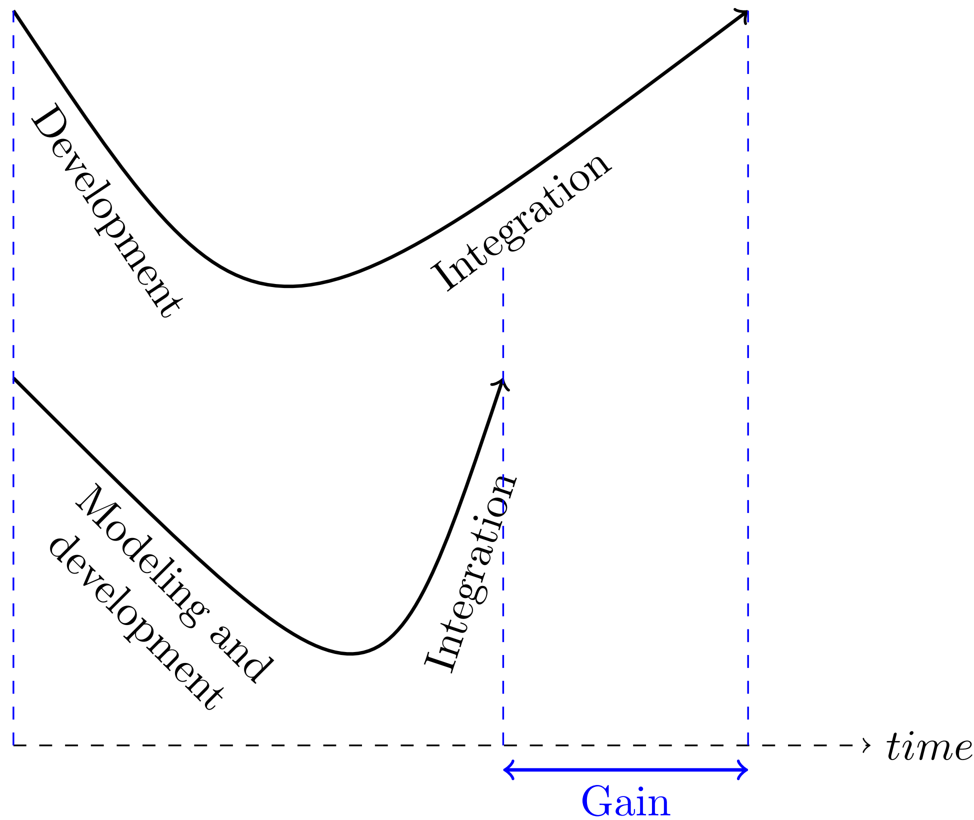


Figure 1.1: The benefits of Model Driven Engineering

In Section 1.2.1 we disambiguate some vocabulary. In Section 1.2.2 we describe different application graph models and their properties, and we try to fill the gap between theoretical graph models and programming languages in Section 1.2.3.

1.2.1 Jobs, Agents, Processes and Tasks

The definitions of jobs and tasks depend on the communities and topics. In the field of theoretical scheduling it is generally assumed that a job is made of one or many “tasks” that execute only once such as in the open-shop scheduling problem [38] (or “operations” in the job-shop scheduling problem). A confusion exists regarding the notion of “periodic task” scheduling since a “periodic task” obviously yields a task each amount of time corresponding to the period. In the community of Dataflow Process Networks (DPN), a node of the process network is sometimes called an “agent” or a “process” and can be executed multiple times [62] as long as its input ports are provisioned. Sometimes an “agent” is called a “task” [87] which can introduce more confusions (since a “task graph” for instance could then have cycles). When it comes to practical aspects of software engineering, a “process” often consists in an executing program in the context of an operating system. This process can create other processes (or threads), and communicate with them (Inter Process Communication). Agents or tasks can thus be implemented as processes (or threads), which connects scheduling problems with the software engineering field. Another confusion arises since some celebrated frameworks that are used to build parallel implementations (such as OpenMP) also use the “task” keyword in their syntax [5].

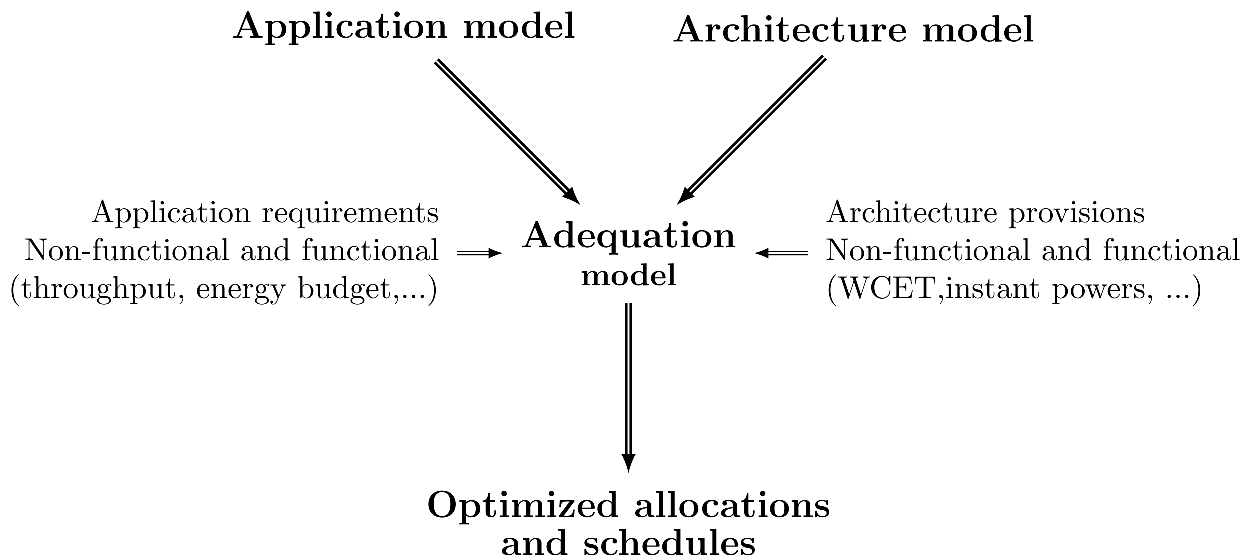


Figure 1.2: The Ychart or AAA methodology

A compiled and optimized (non running) program which executes a basic computation is sometimes called a “kernel”.

We adopt the “agent” keyword from the DPN and the definition of a non-preemptive “task” from the field of theoretical scheduling. The model of agent then defines the number, order (if they have dependencies) and costs of the tasks. A task is thus an instance or occurrence of agent. We call “kernel” the implementation of an agent. In our static scheduling approach an essential property of an agent is that even if it yields multiple tasks, we must be able to associate a cost to each of them (cf section 1.4). This means that the kernels implementing those agents must validate some properties, such as data independent behavior, predictable durations, no side effects... Typical examples kernels include dense linear algebra routines such as matrix operations, spectral methods, image or video processing operations.

1.2.2 Graph models

Many scheduling approaches directly consider Direct Acyclic Graphs of tasks as an input of the problem [93, 3]. Agents can be connected together to form different types of graphs, for instance dataflow or control flow graphs.

1.2.2.1 Dataflow Process Networks

DPN sometimes include redundant or recursive task executions which is a relevant information for the analysis. This pattern can be captured by the model. The typical example of such graph is the classic split merge pattern. We can easily imagine an allocation and scheduling algorithm for such structures, (for homogeneous and heterogeneous resources as well) by using the rates $m, n \in \mathbb{N}^2 \mid m \bmod n = 0$ defined in Figure 1.3 (left). Of course this scheduling algorithm would be specific to split merge graphs. In typical and practical problems the whole graph consists in hierarchical structured graphs or unstructured connections of structured subgraphs. StreamIt graphs (Figures 1.4) and SDF graphs (We use some in Chapter 3) are examples of such graph.

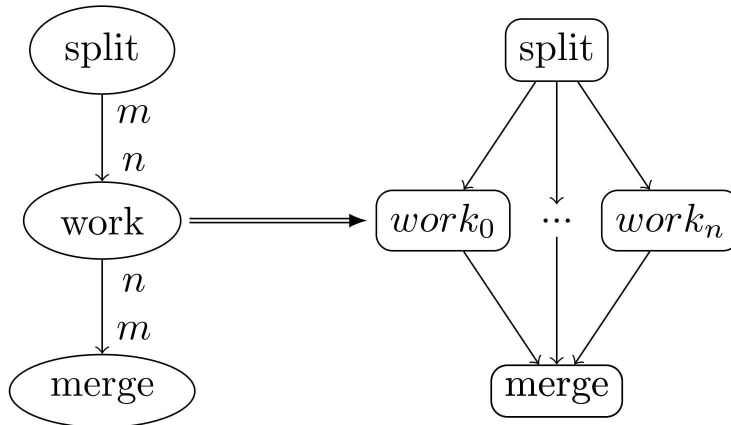


Figure 1.3: Split/merge dataflow process network and its corresponding graph of tasks

One of the most famous attempt to this is the Synchronous Dataflow model [61]. Synchronous Dataflow Graphs (SDFG) allow to model cyclic graphs of processes with input and output data rates, and analyze the correctness of the model. The original model does describe the multiprocessor scheduling of such structured graphs (apart using results from theoretical scheduling). It is however able to sequence them, and sequences can also be optimized given certain metrics such as code size (single appearance schedules) or memory consumption [8]. The classical SDF model does not include cost functions of the tasks, which is not required for sequencing but mandatory for multiprocessor scheduling. An SDFG is a bipartite graph where edges can be divided in two disjoint sets named Agents and Places. Then, the classical technique for multiprocessor scheduling of SDFG [60] is to transform them to Homogeneous SDFG (with unitary input and output rates) or "Acyclic Precedence Graph" (APG, with equal input and output rates) in order to use known heuristics (such as HEFT[93], described in Section 1.5.3) for multiprocessor scheduling of unstructured DAGs.

We refer to this model in section 2.1.1.3, and we define the main concepts here. An "agent" is a functional block that consumes and produces a static amount of data (or tokens) in its input and output buffers called "places". Weighted arcs connect agents with places. Two agents (resp. places) cannot be adjacent. The "marking" associates tokens with each place, the initial marking is the initial number of tokens in all places. We call a "period" a valid execution of agents such as every agent runs at least once and the marking takes its initial value. Tokens are used to abstract the different elements of data that are read/written by the agents. The agents abstract the tasks and should thus read and write a static amount of tokens. The Cyclostatic SDF [9] extension relaxes this requirement and allows to model actors with a cyclic behavior (the number of tokens read/written is periodic). SDF and its variants offer a concise and parametric model to describe data-independent applications, including cyclic behavior such as stream processing applications. When the model admits one or multiple cycles, finding a sufficient and minimal initial marking is a challenging problem. In this case the analysis of the graph also consists in finding an initialization and an optimized steady state.

1.2.2.2 Structured graphs

The most well known attempt that uses structured graph constructs to achieve multiprocessor scheduling is StreamIt [92], which allows to use different scheduling heuristics. The StreamIt dataflow language defines sophisticated constructs in order to manipulate the flows of data and organize them as structures such as pipelines, feedback loops, and different ways of splitting and merging data... According

to estimated metrics, different structures of the graph can be evaluated. Provided a balanced input dataflow graph and a scheduling policy, a static mapping is generated. It includes optimization such as pipelining periods of cyclic graphs. Figure 1.4 gives one of the StreamIt graph representation of the blocked matrix multiply algorithm, with four types of nodes:

Identity (blue) data is copied from the input port to the output port

Roundrobin (white) data is distributed or gathered with a round robin policy according to the rates

Duplicate (white) data is duplicated on all the output ports

Actual work (red) where the data is actually modified

In this example the majority of the nodes move data but do not modify it, which gives an idea of the vast search space regarding possible graph transformations.

Other examples of structured graphs include recursive definitions of divide and conquer algorithms such as the Fast Fourier Transform. Structured graphs can also be built from definitions of algorithms such as FFT, blocked matrix multiplication or blocked Cholesky decomposition [3]. We experiment our approach using pseudo code generated graphs in Chapter 3 (Figures 3.11, B.1, B.2).

1.2.2.3 Conflict freeness / functional determinism / latency insensitivity

An important aspect of formal models is the absence of conflicts. Conflict-freeness has a different name depending on the formal model. Conflicts are unspecified aspects of the execution, for instance when two agents compete for reading or writing. Some models introduce non determinism regarding the flow of data (or tokens) such as Petri nets [76]. Petri nets is thus not a conflict free model. The conflict resolution must be modeled with another formalism such as finite state machines. Petri nets can be sub-classed into conflict-free models such as marked graphs [22]. Kahn Process Networks (KPN [50]) or synchronous dataflow graphs[61] are also conflict free: in the SDF model the places connect only one producer to one consumer, thus no two nodes can compete for tokens. We limit our investigations to conflict free models only and we detail the methodology to check conflict freeness regarding the SDF model thereafter.

In an SDF graph, the analysis of the input and output rates of the agents allows to check that no agent will starve or no place will grow infinitely. In SDFG this can be checked by solving the "balance equations" which consists in finding the vector of non null integers X in Equation 1.1. We call X the "repetition vector". If such vector exists then the graph is "balanced". Let $\mathcal{A}$ be the set of agents and $\mathcal{P}$ the set of places. $\mathbf{card}(\mathcal{A})$ (resp. $\mathbf{card}(\mathcal{P})$) is the number of agents (resp. places). The topology matrix Γ is a matrix of $\mathbf{card}(\mathcal{A})$ columns by $\mathbf{card}(\mathcal{P})$) lines. $r(a,p)$ is the rate or weight of the arc connecting the an agent $a \in \mathcal{A}$ to a place $p \in \mathcal{P}$. It is positive if the place is an output, negative if it is an input, and zero if the agent and place are not connected.

$$\begin{aligned} a_i &\text{ is the } i^{th} \text{ agent in } \mathcal{A} \\ p_i &\text{ is the } i^{th} \text{ place in } \mathcal{P} \\ \Gamma &: \gamma_{i,j} = r(a_i, p_j) \\ X = (x_1 ... x_{\mathbf{card}(\mathcal{A})}) &: \Gamma.X = 0 \end{aligned} \tag{1.1}$$

Another analysis consists in finding the maximum throughput of the graph which is limited when the graph admits one or multiple cycles. When the graph has multiple cycles, the throughput of the graph depends on the throughput of the slowest cycle and determining the minimum and optimal marking (regarding throughput) is a challenging issue [67].

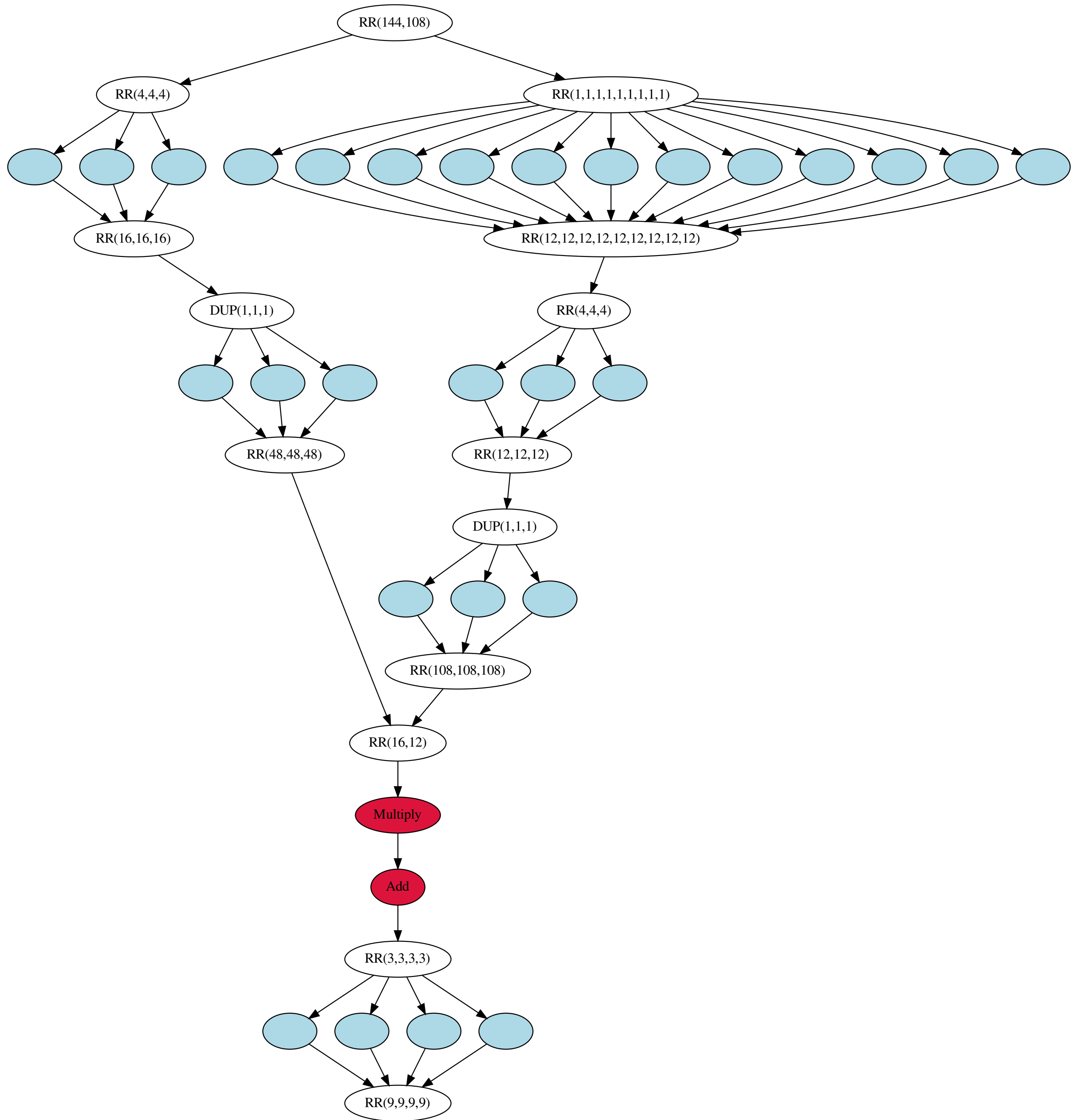


Figure 1.4: StreamIt graph of the blocked matrix multiply algorithm

StreamIt graphs are also conflict-free and balanced. In Figure 1.4, when $144 + 108$ tokens enter the graph, one complete iteration of the graph is executed down to the output node and the marking of the StreamIt graph comes back in its initial state.

Process networks thus usually define the two properties:

- "liveness": there exists a sequence where every agent fires infinitely often, which means that there is no deadlock or livelock (and there is no data "underflow" in the places)

- “safety” (or “boundedness”): the application can be executed with a finite memory capacity (and thus there is no data “overflow” in the places)

Those structured graph models can thus be used to find the set of correct partial orders for the execution of agents. Then, the classical analysis regarding sequencing and scheduling consists in extracting a complete order that is optimized regarding a criteria (size of the code, memory requirements, makespan, ...). A typical policy for the simulation of process networks consists in firing an actor as soon as its input ports are provisioned (ASAP). Live process networks admit a periodic sequence of firings that can be described for each node with a K-periodic binary word [67].

Even though some structured graph models also define languages (StreamIt does), they are still not commonly used for software engineering purposes. Domain Specific Languages (DSL) focus on a specific topic in order to improve the clarity and the efficiency of the programs (such as Halide[79] which targets image processing). They provide a textual syntax which is used to describe instances of the previously described formal models. In the next section we consider the more pragmatic approach to multiprocessor scheduling and we try to related them to those formal models. In essence the Domain Specific Languages encapsulate the behavior (Section 1.2.1) and the structure (Section 1.2.2) within the same model and thus may fill the gap between formal models and classical sequential languages.

1.2.3 Relations to general purpose languages

In the compilation area, scheduling mostly refers to instruction scheduling which means organizing the instructions such that the micro-architecture executes them faster. With the upcoming parallel architectures scheduling may also include automatic parallelization. On contrary in the HPC area, scheduling mostly refers to load-balancing, thus only to “allocation”. HPC runtimes are most often designed to process a large amount of data as fast as possible (with a best effort policy). Most often the predictability of the execution is not the main concern of those tools. It however includes constructs that allows the distribution and reduction of data as like as DPN. MapReduce[32] is an example of a programming model (and implementation) for processing large amounts of data-parallel tasks on a large amount of resources.

Sequential languages such as C and FORTRAN require that the programmer gives a synthetic execution order of the routines or instructions. Since those languages do not provide parallelism or concurrency in their syntax, the granularity of the task, their core affinity and their scheduling is either statically assigned or delegated to an external tool (or runtime scheduler through code annotations or API calls). Some specific constructs can be automatically analyzed and transformed to formal model for further analysis. Static Affine Nested Loop Programs (SANLP) can be transformed to process networks[95]. A SANLP is a set of statements surrounded with nested loops. Input and output data is a multidimensional array. The loop bounds must be immediate values and data read or write is allowed only to affine combinations of the indexes. Dependency analysis allows to build a graph of the statements. Polyhedral optimization approaches [77] manipulate this intermediate representation model of loop nests to find an optimized order to traverse the space of statements, and to tile them for better parallelism. Typical transformations such as loop tiling and unrolling also exist in general purpose language compilers, and the exploration in this state space is still a difficult topic [54, 7].

It’s not easy to extract significant information from existing sequential or parallel implementations (which is known as “automatic parallelization”) but those languages provide a sufficient backend to implement the result of a preliminary analysis based on previously described models. Early answers to the growing amounts of computing resources consists in annotating sequential application description to parallelize them.

Regarding practical aspects, the languages and APIs that are used to leverage parallelism implementations often match the hardware memory organization: shared or distributed. Multithreading matches the former: multiple threads can share memory areas. The conflicts and memory bottlenecks have to be carefully examined by the programmer. Message passing matches the latter, processes cannot share a memory area, but they are able to send and receive messages. Both approaches have many implementations such as POSIX threads, OpenMP, MPI. We use some of those tools in the experiments of Chapter 3.

1.3 Application requirement modeling

Scheduling is a very large topic with plethora of problems, cost models, and publicated algorithm of different nature. The real-time scheduling community often focuses on finding heuristics with interesting properties (regarding schedulability or performance optimization). The analytical approach to scheduling is sometimes referred as “operational research” although this domains also covers different topics (including scheduling). This section describes some modeling elements of the real-time scheduling theory. Extra functional requirements may be added to the application model aside from its former introduction in previous sections. Requirements can deal with performance (data throughput) or real-time criteria. In the current section we focus mainly on performance aspects and task models. We describe preemptive task models in Section 2.1.1.5, periodicity requirements in Section 1.3.2 and objective functions in Section 1.3.3.

we investigate architecture provisions and energy aspects in Section 1.4. In real-time scheduling theory one usually also provide a high level cost function and architectural platform which description is deferred to section 1.4.1.1.

1.3.1 Preemption and migration

“Preemption” is the ability to start a task on a resource, suspend it before it completes and resume it on the same resource. “Migration” is the ability to start a task on a resource, suspend it before it completes and resume it on a different resource. Theoretical scheduling algorithms often target either only preemptive tasks or non-preemptive tasks (and do not consider mixing both task models).

1.3.1.1 non-preemptive task models

We call “Task” the non-preemptive task model in theoretical scheduling. A typical set of variables associated to non-preemptive tasks in scheduling includes:

$$\begin{aligned} start, stop &\mid duration = stop - start \\ appearance &\mid start \geq appearance \\ deadline &\mid lateness = stop - deadline \end{aligned} \tag{1.2}$$

Figure 1.5 gives a graphical definition of those variables. When the lateness is negative it is sometimes referred to as “laxity”.

1.3.1.2 Preemptive task models

Figure 1.6 gives an example where a task is preempted once (in two tasks). Figure 1.7 shows an example where a task is migrated once from resource R_0 to R_1. In each case the total duration of the task is the sum of the intervals: $duration = preempt - start + stop - resume$. However preemption

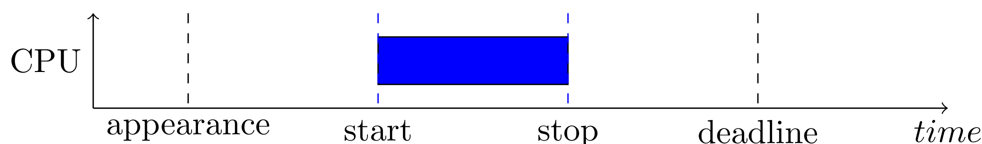


Figure 1.5: Model of a non preemptive task

and migration have a cost. This cost can be modeled. For instance when an operating system running multiple processes does preempt one of them, it must save and restore its state (which we call a "context switch").

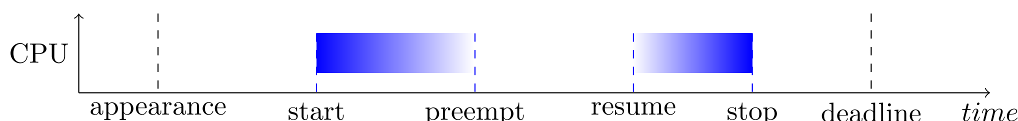


Figure 1.6: Model of a task preempted once

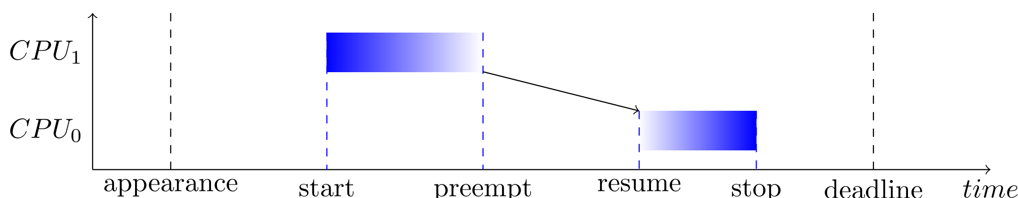


Figure 1.7: Model of a task migrated once

In a preemptive task scheduling problem, the tasks can be preempted and migrated infinitely often. In a practical multiprogramming environment, a process can be preempted or migrated each time the online scheduler runs. This scheduler runs periodically (typically 100Hz to 1kHz). The state space of such problem is finite but very large (depending on the duration of the tasks). This also means that the problem cannot be encoded only using the variables defined in Equations 1.2.

One of the main approaches in preemptive scheduling focuses on periodic task systems and consists in using the rates of the tasks in order to determine if the system is schedulable. Another approach which is not specific to periodic tasks consists in modeling the active time of each task with a binary vector whose length equals the number of time slots (TDMA: Time Division Multiple Access). A third approach consists in transforming the preemptive problem in a non-preemptive one by preempting each task a given number of times (pre-computed preemption).

In the latter the question then arises as to find the smallest amount of preemptions and migrations (or the largest timestep) that still provides the optimal solution in the sense of preemptive task scheduling. This problem is known as "limited preemption" [16] or "precomputed preemption" [18] and we partially address it along with preemptive task modeling in section 2.1.1.5.

1.3.2 Periodicity

Some events are intrinsically periodic because they are related to physical periodic events (such as periodic signals). Sometimes it is necessary for functional reasons to sample or generate a signal at

a given throughput. Another common example of periodic systems is a streaming application that admits a periodic sequencing of the tasks.

We use the confusing notion of “periodic task” (see Section 1.2.1) (calling them periodic jobs or periodic agents is also confusing in the sense of theoretical scheduling and process networks). The definition of a periodic task usually includes a period t_{period}, a duration $t_{duration}$ and a deadline (relative to the period) $t_{deadline}$. When $t_{deadline} = t_{period}$ the rate is defined with Equation 1.3

$$t_{rate} = \frac{t_{duration}}{t_{period}} \tag{1.3}$$

A periodic task system can be transformed to a system of independent tasks over its “hyper period”. Given a set of periodic tasks $\mathcal{T}$ of physical time period t_period, the hyper period is the least common multiplier of the periods:

$$Mc = LCM(t_{period} \mid t \in \mathcal{T}) \tag{1.4}$$

1.3.3 Objective functions

The makespan and the maximum lateness are the most studied objective functions in task scheduling optimization problem (Equations 1.5 give example objective functions), and we mainly focus on the makespan objective in this work. Graham’s classification [43] defines multiple objective functions such as:

$$\begin{aligned} \text{Makespan} : &\max_{t \in Tasks} t_{stop} \\ \text{Maximum lateness} : &\max_{t \in Tasks} (t_{stop} - t_{deadline}) \\ \text{Weighted sum of lateness} &\sum_{t \in Tasks} w_i \cdot t_{lateness} \end{aligned} \tag{1.5}$$

The lateness would typically be used to distinguish hard real-time problems for soft real-time problems: in the former, violating the deadline is not allowed, in the latter it is allowed at a defined cost. It can also be used to distinguish the criticality of some tasks, for instance by using the weighted sum of lateness as an objective function.

The notion of optimality is closely related to the notions of requirements, for instance regarding periodic task systems the main concern is schedulability of the system since the makespan is not relevant. The optimal value can also depend on the task model, for instance it is easy to find an example where a preemptive schedule achieves a better makespan than a non preemptive one.

1.4 Architecture modeling

In Section 1.4.1 we discuss the resources and communication modeling in the real-time scheduling theory. Then in Section 1.4.2 we discuss practical aspects of computer architecture organization. Finally we describe power saving techniques and energy modeling in Section 1.4.3.

1.4.1 Architecture modeling in real-time theory

In this section we describe the common resource models and communication models studied in real-time scheduling theory.

1.4.1.1 Resources

Theoretical problems typically consider three resource types: homogeneous, heterogeneous, and related (as described in Graham's classification [43]). Given two tasks t_i, t_j in $\mathcal{T}$ the set of tasks. Let $c_t^{r_k}$ be the cost of running task t on resource k. Resources k and l are:

- *homogeneous* $\forall t_j \in \mathcal{T} : c_j^{r_k} = c_j^{r_l}$
- *related* with coefficient r: $\forall t_j \in \mathcal{T} : \frac{c_i^{r_k}}{c_i^{r_l}} = r$
- *heterogeneous* $\exists t_i, t_j \in \mathcal{T} \mid \frac{c_i^{r_k}}{c_i^{r_l}} \neq \frac{c_j^{r_k}}{c_j^{r_l}}$

Other models consider an arbitrary cost function where the task costs is a function of various parameters, for instance the task starting time [19].

1.4.1.2 Communications

Graham's classification accounts only for precedence relations between tasks [43]. Later the classification has been extended to better account for communication delays [97].

- Instantaneous communications: there is an infinite amount of links, and an infinite throughput between resources
- Fixed cost model: the application graph specifies communication delays
- Fixed cost models that depends on sender and receiver allocations (such as in [93])
- The cost is an (arbitrary) cost function which may depend on sender task allocation, receiver task allocation, their distance regarding the number of links, the amount of data and other parameters such as the latency and throughput of the link.

The third model accounts for the "locality principle" described in 1.4.2.2. The fourth model consists in considering structured networks to benefit from special properties such as symmetry, minimum distance between two nodes, robustness to broken links, ... For instance some effort has been conducted towards the study of k-ary n-cubes [25] and such network has then been used to build HPC supercomputers (Blue Gene and Cray use a k-ary 3-cube) and then on-chip networks (Kalray MPPA [28] uses a torus network on chip which is a 4-ary 2-cube). When using such networks, because multiple routes exist between two nodes a routing algorithm is used.

The most complex models (fourth case) can be used for instance for network on chip modeling. Typical network (on chip) communication models include a constant cost α, a per-byte cost β and a per-hop cost γ such as Equation 1.6 for a message of n bytes traversing h hops. We expected that the same model would apply on a network-on-chip microarchitecture and experimented on the Kalray MPPA but we found no evidence of hop cost.

$$t(n, h) = \alpha + \beta \cdot n + \gamma \cdot h \tag{1.6}$$

1.4.2 Computer architecture classification in practice

When it comes to practical aspects many computer architectures compete, both regarding their microarchitecture and their memory organization. We recall some of the main microarchitectures and communication resources in Sections 1.4.2.1 and 1.4.2.2.

1.4.2.1 Resources

Finding a versatile architecture and memory organization at the chip level is tricky and there is not yet a de facto standard, which explains the current diversification. The computing resources must support be able to run ever increasing workloads (and thus be scalable) and dynamically adapt to low workloads by dynamically switching to low energy states.

A typical unrelated (or heterogeneous) resources setup is found when different computing microarchitectures are used. For instance in the integrated GPU CPU architectures such as Intel Haswell the GPU is proficient at executing vector operations. Hence the performance of the tasks on the GPU essentially depends on this criteria whereas the cost of the CPU implementation is not very sensitive to this.

In the compilation area, high level architecture modeling tools are not often involved. Acronyms are used to depict the main characteristics of the target architectures, such as:

SMP (symmetric shared memory multiprocessor): the resources are identical and symmetric about one memory. Multiple SMP can be combined, but the resulting architecture is thus not an SMP. For instance the big.LITTLE ARM architecture is made of one cluster of four identical big cores that access the main memory through the same path, and another cluster of four identical little cores.

(GP)GPU (General Purpose Graphical Processing Unit): the resources and interconnects are designed to provide high data throughput for embarrassingly parallel operations

DSP (Digital Signal Processor): an architecture that contain specialized hardware and instructions to increase instruction level parallelism

VLIW (Very Long Instruction Word): the compiler is in charge of clustering instructions that will be executed in parallel

1.4.2.2 Communications

There is no cheap, fast and large memory technology. Applications need to store large amount of data, and also manipulate or transform some of it very quickly. As a consequence every computer architecture needs to interconnect large (but slow) memories with small (and fast) memories (an example is visible in Figure 1.8). The arithmetic and logical units are the components that manipulate data, thus the small and fast memories are connected to those components. This is known as “the locality principle”. Moving data between large and small memories makes the complexity of software optimization. We describe thereafter the two memory models of computer architectures: shared and distributed memory.

The most widespread solution consists in adding hardware logic that tries to guess which data will most likely be fetch from the main memory, and try to retrieve it before the program uses it. This is the well known shared memory model architecture with sometimes multiple hierarchical levels of caches. The downside is that controlling this memory hierarchy is sometimes very complex since the software is not allowed to explicitly access the caches. When a block of data has to be fetched and is not in the cache, a cache miss occurs and the block is fetched from the main memory. Three types of cache misses can occur:

- Obligatory: The cache is empty (or “cold”), the block of data has to be fetched for the first time
- Capacity: The program runs through a larger memory area than the cache size

- Conflicts: The cache is smaller than the area it caches, thus each block in the cache maps to multiple blocks in memory. Accessing those blocks successively causes conflicts.

In an MPSoC the communication between integrated components (sensors, CPUs, DSPs, ...) usually goes through one or multiple Bus which are also arbitrated by an hardware bus arbiter, with no or little available control from software.

The opposite approach of distributed memory on contrary relies on software to leverage the locality principle. Because the regular compilers are not able to perform it, this programming model is not a very popular choice. Thus the trend in system on chip design is to integrate computing resources (heterogeneous CPUs, GPUs, DSPs, ...) with a shared memory hierarchy (Intel HD Graphics integrated GPUs, ARM Mali integrated GPUs, ARM big.LITTLE heterogeneous multiprocessing, ...). Yet some approaches try to leverage on-chip distributed memory architectures, mainly using network on chip interconnects. In a Network on Chip the resources are connected with a regular network graph, such as 2D meshes, 3D meshes, or torus (see Section 1.4.1.2). There is no de facto standard memory organization. Many attempts exist (Raw, Aethereal, Kalray) and they often come with a specific compiler that handles the communications on the network when they are not explicit in user code. In some architectures each node of the graph is itself an SMP.

"Scratchpad" memory organization is another distributed memory architecture which is organized hierarchically like caches, with the essential difference that every memory can be read/written explicitly from software. As like as network on chips it needs a specific compiler and is not often used in practice.

Modeling shared memory architectures is very hard and most often consists in modeling cache accesses or limiting them. Modeling distributed memory architectures is easier but is harder to leverage in practice since distributed memory embedded system architectures are not widespread.

1.4.2.3 Extracting some knowledge

Existing tools allow to extract information from a (running) target [52]. Hwloc extracts information about memories, memory hierarchy and resources [13]. Netloc[37] is a recent addition that allows to obtain information about the network. Those tools essentially target High Performance Computers. Most often the model is only structural such as in Figure 1.8.

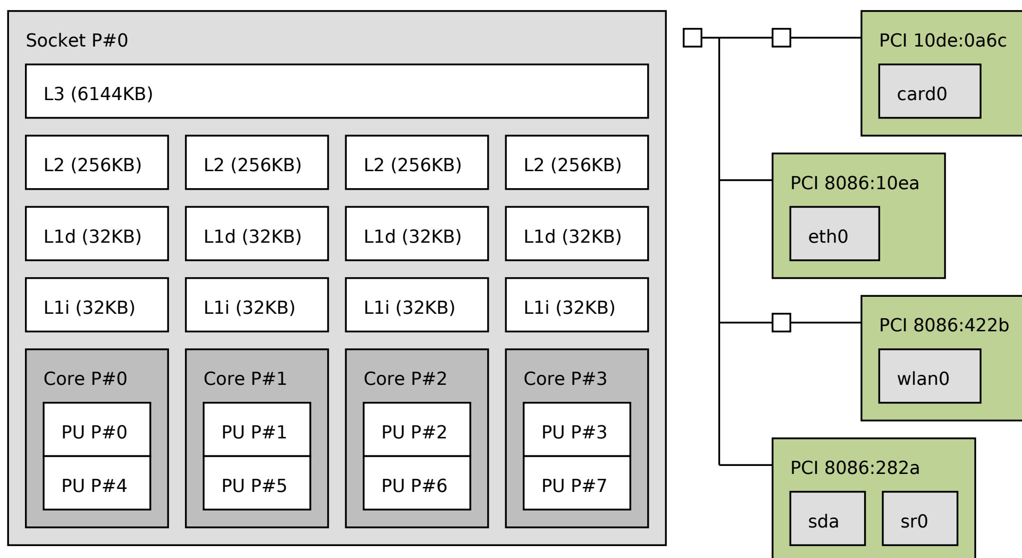


Figure 1.8: lstopo output on a laptop multicore computer

1.4.3 Power saving mechanisms

An embedded system is by definition not plugged into any power outlet, and has thus to run on battery. Moreover, most of the power it consumes is dissipated into heat which is usually not desirable. Increased battery performance and a better energy management improved the autonomy of our devices in the last decade. Energy consumption is due to transistors switching when the microarchitecture is active, and transistors leakage current when it is powered but inactive. In Sections 1.4.3.1 and 1.4.3.2 we study the main two mechanisms of energy management.

1.4.3.1 Dynamic Voltage and Frequency Scaling

DVFS is the ability of computing resources to decrease their frequency to save some energy. In most DVFS implementations the frequency can be chosen dynamically in a set of available Operating Performance Points (OPP) and not within a continuous range.

We recall the admitted high level instant power model for microprocessors [53]: given the OPP $o = (V, f)$ of a resource r:

$$\begin{aligned} P(r, o, t) &= P_{dynamic}(t) + P_{static}(t) \\ P_{dynamic} &= C_r \cdot V^2 \cdot f \cdot A \\ P_{static} &= V \cdot I_{leak} \end{aligned} \tag{1.7}$$

We define the variables of Equation 1.7 and we give their dimension:

- C_r [Farads]: the model capacitance of the CPU (roughly proportional to the number of the switching devices, i.e. the number of transistors)
- V [Volts]: the voltage of the current operating point
- f [Hertz]: the frequency of the current operating point
- A [dimensionless]: An activity factor that models the average number of bit flips that the task will cause on the resource. Although it is specific to each pair of task and resource it is often considered constant
- I_{leak} [Amperes]: Transistors leak a small amount of current when they are powered (even if they are not switching)

Regarding only dynamic power, and given a task which costs C_y cycles and runs on the resource r at OPP o: the task will last $\frac{C_y}{f_o}$, thus we can model the consumed energy with the expression 1.8. In order to operate normally, the chip must raise f_o linearly with V_o [71]. We can thus remove the dependency to f_o in $E(t)$ which means that running a task at a low frequency actually consumes less energy than running it at high frequency, even though it lasts longer. This is the purpose of frequency scaling.

$$E(t) = \int_t P(t) = \frac{C_y}{f_o} \cdot P(r, o) = C_r \cdot C_y \cdot V_o^2 \cdot A \tag{1.8}$$

1.4.3.2 Power gating

Setting the CPU to the lowest frequency does not cut all the dynamic power. Power gating is the ability to shut down a subset of the resources and thus eliminate both this remaining dynamic power and static power. When the CPU is not running, its instant power follows the same model, with a reduced activity factor (since fewer gates will flip while idling).

The instant power of the system depends on time because the resources are allowed to change their operating point in time. Given the instant power $P(R)$, the energy with a dimension of $Watts \cdot seconds = Joules$ is thus calculated for each resource r over the duration of the schedule (which means from zero to makespan M).

$$E_a(r) = \int_{t=0}^{t=M} P(r)\,\mathrm{d}t \tag{1.9}$$

$$= C_r \cdot \int_{t=0}^{t=M} V^2 \cdot f \cdot A\,\mathrm{d}t \tag{1.10}$$

Let t_o be the opp for task t, this means that each resources r running at opp o consumes:

$$E(r,o) = C_r \cdot \sum_{t \in Tasks | t_o = o} \cdot \frac{C_y}{f} \cdot V_o^2 \cdot f$$

$$E(r,o) = C_r \cdot \sum_{t \in Tasks | t_o = o} C_y \cdot V_o^2$$

Let O_r be the set of OPPs of resource r, the energy consumed by the tasks running on r is:

$$E_a(r) = \sum_{o \in O_r} E(r,o)$$

We call that quantity "active" energy consumption.

Finally let P_{idle} be the idle instant power of the resource, t_{pt} the physical time duration of task t and t_{map} the resource where t is mapped. The idle energy can also be expressed as a discrete sum:

$$E_i(r) = P_{idle} \cdot (M - \sum_{t \in Tasks | t_{map} = r} t_{pt})$$

Thus, the consumed energy of the system is:

$$E = sum(E_i(r) + E_a(r) \mid r \in R)$$

Recent studies shows that indeed the energy vs frequency function for a given task is a convex function [31]: If the frequency is very low then static power dominates and on contrary if the frequency is high dynamic power dominates. In the middle lies the most energy-efficient operating point. Validating real-time constraints however sometimes requires to run tasks at higher frequencies.

1.4.3.3 Motivating example

Lowering the frequency of a task does not always mean that the optimization criteria is degraded. For instance if we want to optimize makespan, we can find very simple application descriptions of tasks and dependencies which give evidences that energy can be optimized with a preserved makespan. Figure 1.9 is a typical problem instance with a split merge of two unbalanced workers, where slack time can be used to save energy. Generally speaking, this applies to each non critical path. This also gives evidence that the solver will be able to optimize energy if we allow more slack time from the optimal makespan schedule.

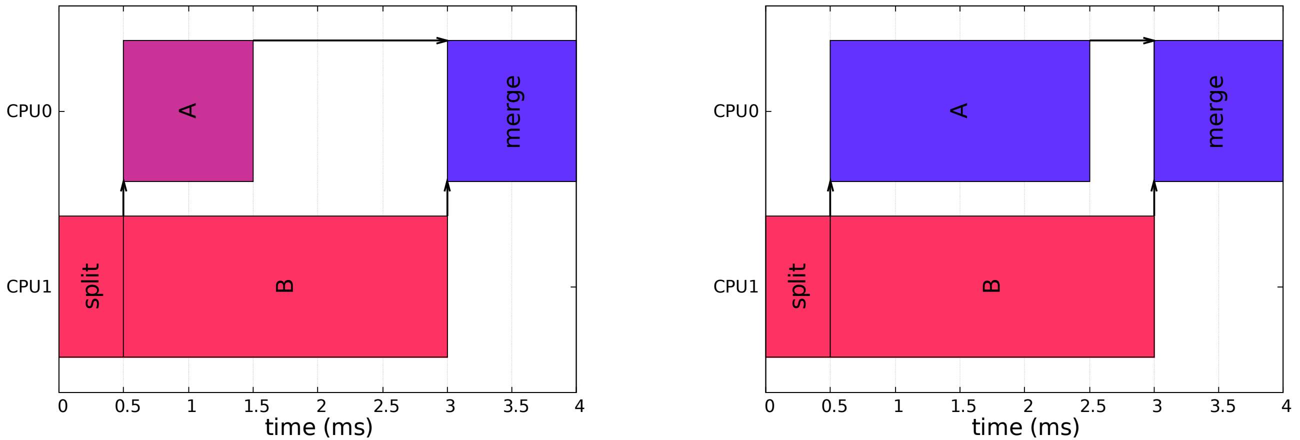


Figure 1.9: Non energy optimized vs optimized split merge pattern (red tasks run with high instant power)

When the application graph is not a simple unbalanced two tasks split/join, it is (computationally speaking) much harder to decide the allocation and the operating point in order to obtain an optimized schedule. For a few tasks with precedences it might even be hard for a human to optimize for makespan, considering only high frequency operating points. Then, the more slack time, the higher possible energy optimization, and the larger state space to explore.

1.4.4 Temperature modeling

Temperature modeling and prediction vs CPU activity is a very difficult topic. Typical models suggest that leakage power depends exponentially on Temperature [45]. As well as dynamic power, this leakage power is dissipated into heat. To prevent the runaway of this positive feedback loop (which would cause permanent failure), hardware thermal management mechanisms exist. Throttling is the typical solution, which consists in omitting cycles to allow the chip to cool down (without modifying its running frequency). Omitting cycles obviously causes a performance drop, which is hard to notice if there is no frequency change notification from the hardware. Some CPUs however will raise an event that can be captures by the software and notified to the running process.

Temperature variations are hard to predict at compile time, and the initial conditions are not always known. A naive solution consists in solving the same scheduling problem under the new constraints that high power operating points are forbidden. If we are able to find a steady state then the application can continue running using this version of the adequation. Otherwise we need to consider a lightweight version of the application (with diminished quality or features) and alternate between this and the full-featured version when the temperature comes close to the level which triggers

throttling. Hopefully, streaming applications rates (such as video decoding, typically 10 to 100Hz) are orders of magnitude smaller than temperature variations, which allows to evaluate between iterations of the application which version it will use in the next iterations in order to cool down the chip.

The heat is generated by two phenomenon in a chip. The first one is internal and is due to the Joule effect: some current passes through the chip, which releases heat. The second one is external and is due to thermal conduction of heat from the outside of the chip, or from other areas of the chip. Conduction is noticeable (see Figure 2.11), and requires "some" knowledge of the physical layout which is most often undisclosed. Both phenomena are non linear, and (in the specific case of CPU temperature modeling) cannot be safely estimated with a linear model.

The generated heat can be modeled with an RC circuit [84]. Since it is non linear, analyzing it is complex. Some modeling effort using piecewise linear functions exists [80]. We do not model heat generation or conduction, but we do some experiments in Chapter 2 to evaluate both phenomenons.

1.5 Existing approaches to AAA

All the previously listed models eventually come across the allocation and scheduling problems which consists in distributing the execution of the tasks on the resources in time. Sometimes an objective function is used to pick out the best solutions. The models of agents, messages, resources and interconnects can be sophisticated and the underlying adequation problem consisting of interval of times and resources can require a pre-processing step. We give different state of the art approaches, and an insight of their computational complexity. We mainly distinguish three approaches: simulation (Section 1.5.1), heuristic algorithms (Section 1.5.3) and analytic algorithms (1.5.2). In Section 1.6 we introduce automated reasoning tools and we motivate their use for scheduling. In Section 1.5.5 we discuss how the result of those approaches (the schedules) can be represented.

1.5.1 Discrete Event models and test by simulation

A system of resources and tasks as like as the one described in section 1.3 can be modeled and simulated with a Discrete Event Simulator (DES) such as systemC. This requires that the scheduling and allocation is specified (either statically, but most often it is a heuristic) and allows to observe one execution of the system, also known as a "trace". Although the simulation is not computationally complex, the huge number of signals existing in a complete embedded system prevents cycle accurate simulation in most of the cases.

SystemC allows to explore the different level of abstractions by successively refining the components. Regarding task scheduling and allocation though, the Transaction Level Modeling [17, 36] is most often preferred. It can be used to describe abstract models such as the one typically used in scheduling problems. However since it can only perform a simulation, the designer needs to model the scheduler or explicitly map and schedule the tasks (if the offline scheduler is an external tool). Hence regarding static scheduling, systemC might not be used to find optimized schedules, but it might be used to check if static schedules actually give the expected results on more refined versions of the resources, as an intermediate step towards actual implementation.

1.5.2 Analytic approaches

There exists a variety of offline scheduling algorithms that are often not very generic but instead match a specific scheduling problem. In specific cases, it is possible to find a polynomial complexity algorithm that is optimal according to an objective function, or that is guarantee to find a valid schedule if it exists. If we consider very simple behavioral models (for instance, no message costs,

identical machines, ...): optimizing the makespan of a DAG of arbitrary task costs on two homogeneous processors ("P2|prec|Cmax" is Graham's notation) is already NP-Hard according to Graham [43]. Most of the multiprocessor scheduling problems are actually NP-Hard. On contrary many sequencing problems admit a polynomial time optimal algorithm. Thereafter we give some example problems, corresponding algorithms and complexity.

Coffman and Graham describe an optimal scheduling algorithm for two processor systems as a polynomial time algorithm for non preemptive scheduling of a DAG of identical processing time tasks on two identical resources [20]. Thus "P2|prec,preempt|Cmax" (which is "P2|prec|Cmax" with task preemption)can also be solved in polynomial time. They extend it to an arbitrary number of resources running preemptive tasks, but the optimality holds only for two processor systems, or if the DAG is a rooted tree [72].

Some efforts focus on periodic task sets. The famous Rate Monotonic scheduling algorithm is always able to find a schedule for single resource preemptive scheduling of a system of periodic tasks under the hypothesis that the deadline is the period of the task, and preemption (and context switch) has no cost.

A large amount of problem types (and their associated algorithms) have been studied (surveys [58, 43, 19, 39]). A limitation of those approaches is that each one is specific to a task model, a resources model, a communications model, ... and a small modification of those models may result in sub-optimal, or even inconsistent schedules because the heuristics would be unable to take into account those new problem characteristics.

The adequation problem as we define it in section 1.1.1 is proved NP-Hard (minimizing the completion time of a DAG on unrelated machines [43]). NP-Hard problems can also be solved with an exponential complexity algorithm instead of using generic approaches such as constraints solvers. Finding such algorithm is feasible, but has the same drawbacks than optimal and heuristic algorithms (it is not versatile). The Resource Constraint Project Scheduling Problem (RCPSP) is a typical problem for which many specific backtracking algorithms have been proposed [14].

Another popular option consists in using a verification backend instead of defining an algorithm for a specific problem. For instance systemC models can be transformed to Petri-nets or Promela [94, 48]. This includes transforming the problem to SAT, SAT Modulo Theories (SMT) or Constraint Programming solvers which is what we do in Chapters 2 and 3.

There exists an intermediate approach regarding the resolution of NP-Hard problems. Sometimes even though the problem is NP-Hard, we can find a polynomial time "approximation algorithm". An approximation algorithm comes with a proof that its result is at most at a given distance from the optimal value. For instance we call a 2-approximation algorithm an algorithm that returns solutions optimized up to twice the optimal value. A well known example is a polynomial 2-approximation algorithm for the scheduling of independent tasks on unrelated machines [63]. In this case it is also proved that no polynomial time algorithm can perform better than a $\frac{3}{2}$-approximation. More sophisticated approaches describe an algorithm which complexity is a function of the maximum error [46] which is referred as a "polynomial approximation scheme", and is considered the most versatile polynomial time answer to an NP-Hard problem.

1.5.3 Heuristics

An heuristic trades completeness and optimality for computational complexity. The typical example is a polynomial complexity algorithm that solves an NP-Hard problem. The word "greedy" is also often employed when the algorithm does not reconsider previously established decisions. As a result it is often easy to find counterexamples where the algorithm would take a bad decision since because

of this greedy behavior, it might not be able to make short term sacrifice for long term benefit. This approach is close to simulation based approaches.

As like as optimal algorithms described in section 1.5.2, we need to build a new heuristic for each new problem, and thus there exists a variety of heuristics as well. A popular class of heuristic algorithms is "List scheduling" which consists in sorting the tasks by priority and iteratively selecting a resource for this task. A list scheduling heuristic describes how this priority is computed. For instance in Heterogeneous Earliest Finish Time algorithm (HEFT) [93] the priority of the task is determined using the computation and communication costs of its immediate successors.

ASAP (As Soon As Possible) is also often referred as a scheduling heuristic for a DAG of tasks. It consists in starting each task as soon as it is available according to the precedences. Because it ignores any other constraints such as real-time application constraints or number of resources, it is not strictly speaking a scheduling heuristic. However it can be used to provide a lower bound on the makespan of scheduling problems. The ASAP schedules of a DAG of tasks can also be found by listing the topological orderings of the graph.

1.5.4 Energy concerns

Energy modeling is not often the main purpose of automatic and assisted parallelization [11, 85, 77] although there is a renewed attention for low-power designs. Power and energy modeling involves multiple mechanisms such as frequency scaling and power gating that make both the modeling and the implementation very complex, yet necessary to obtain insightful results [47]. The energy models often introduces non linear expressions and thus raises greatly the practical complexity.

Regarding online scheduling, the current practice in consumer electronic devices in order to save energy is to give the user the ability to set a "policy" or "governor". The default policy often consists in running every core to the maximum frequency when a new process is submitted to the device (which is the well known "on-demand" power governor). When the program stops all the cores go back to a low performance (and low energy) state. Because consumer electronic devices must support dynamic task appearance, this is a good approach for a best-effort behavior regarding energy optimization. Embedded signal processing on contrary often involves a limited set of tasks that must run within a given time budget (such as in a radar processing application), hence the need for static scheduling approaches.

Offline scheduling techniques have been investigated for High Performance Computing architectures [99]. Offline energy aware scheduling heuristics have also been investigated [66].

Simulation tools such as systemC discussed in Section 1.5.1 allow estimation but no optimization [33, 35, 59], again unless further transformation of the models is performed.

1.5.5 Structure and syntax of the result

When every variable of the problem is determined a diagram is often used to present the result. When the approach includes code generation it is usually kept as an intermediate representation. One axis is time, and depending on the problem the other axis either lists resources (it is then a Gantt diagram) or tasks. System of periodic tasks, or sporadic tasks with few resources typically use the latter while other approaches typically use the former for clarity. The Gantt diagram option also allows to represent precedences and messages.

Depending on the output model plotting the result might also require a post processing step. For instance if a resource is not exclusive but allows to run a bounded amount of tasks (which is the typical model for symmetric multiprocessors) displaying the schedule as a Gantt diagram requires to

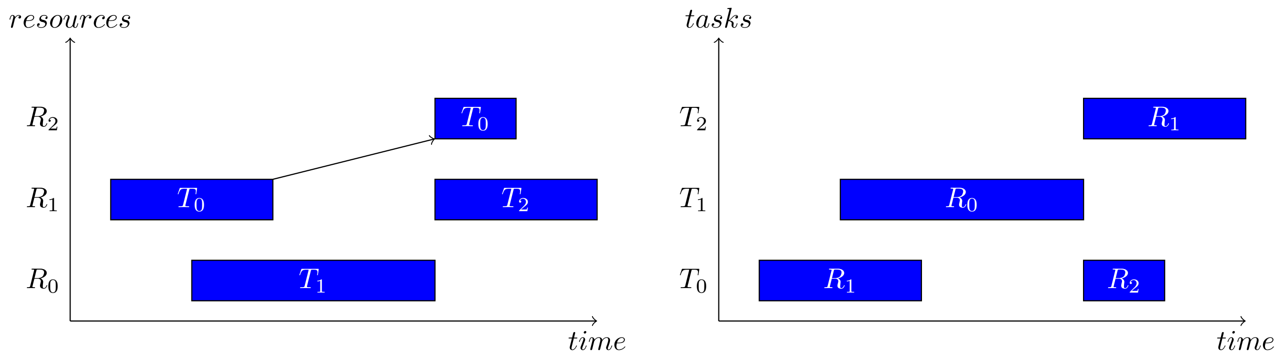


Figure 1.10: Two graphical representations of the same schedule

tile the tasks allocated to this resource which is easy (we can find a polynomial algorithm) since the start and stop dates of those tasks are known.

1.6 Automated reasoning and solvers

Automated reasoning consists in using computer programs to verify or prove mathematical problems. It thus requires models of mathematical objects and algorithms that can manipulate formulas in order to build a rationale or check a property of those models. We usually call a “Computer Algebra System” (CAS) the framework used for the modeling (Section 1.6.1), and we call “solver” the algorithms that we apply on those models. We discuss existing solvers in Section 1.6.2 and we motivate the use of SMT solvers for adequation in Section 1.6.3.

1.6.1 Computer algebra systems

A computer algebra system (CAS) is a tool which allows to declare mathematical objects such as variables, equations and systems of equations. Typical CAS include multiple numeric domains: reals, integers, complex, rationals, ... and provides algorithms to transform or simplify systems of equations. A typical transformation consists in finding the canonical form of a mathematical object. For instance the canonical form of a rational number is $\frac{a}{b} \mid b > 0$ where a and b have no common factor. When the manipulated object is a polynomial, the CAS typically allows to expand it and rewrite its terms in the descending order which is the canonical form of a polynomial. CAS also often provide solvers for specific problems. Typical examples include polynomial equation solvers, or ordinary differential equation solvers.

1.6.2 Solvers and methods for categorized mathematical problems

The problems can be categorized depending on the type of the variables, the degree of expressions, and the degree of the objective function(s). Specific tools can address automatically those particular problems. For instance linear programming (LP) encompasses the problems where all variables are positive and real: $x_1, x_2, ..., x_N \in \mathbb{R}^+$ and additional constraints are linear combinations of those variables that must be bounded such as: $\sum a_i \cdot x_i \leq b_i \mid (a_i, b_i) \in \mathbb{R}^2$. The objective function is also a linear expression of x_i that must be minimized or maximized. The canonical form for LP problems is Equation 1.11, with x a vector of variables, b and c vectors of coefficients and A a matrix of coefficients.

$$\begin{cases} \text{Maximize } c^T x \\ \text{subject to } A, b \mid (Ax - b)_i \leq 0 \\ \text{and } x \geq 0 \end{cases} \tag{1.11}$$

The simplex algorithm is a popular method for solving LP problems, but it has an exponential worst case complexity. The projective method (Karmarkar's algorithm) is a more recent algorithm which solves it in polynomial time [51].

Sometimes the problem cannot be solved in polynomial time but there still exists optimized algorithms that may perform well. The boolean satisfiability problem (SAT) is a typical example of NP-Hard problem. It is admitted that realistic problems often have a low practical complexity despite their large number of variables and expressions. Thus some "large" problem instances can be solved. Recent improvements in SAT solvers [70] also allowed to solve larger instances. Sub-problems of SAT such as Horn-SAT or XOR-SAT can be solved in polynomial time. Integer Linear Programming (ILP) is similar to LP but all the variables are integers. This problem cannot be solved in polynomial time, and two main solutions exist: cutting planes and backtracking. Cutting planes is specific to ILP problems, backtracking encompasses any combinatorial optimization problem.

Adequation problems can be described using a CAS, but it does not always fall in a category of problems that can be solved using one of those solvers. It sometimes require to combine multiple solvers. Efficiently combining theories is tricky, and it is the purpose of "SAT Modulo Theories" (SMT) solvers. Scheduling problems typically combine linear, non-linear and combinatorial expressions (implications or If-then-else constructs). Setting a set of variables and constraints on those variables is a programming paradigm known as "Constraint Programming" which differs from the usual imperative programming in which a sequence of statements change the state of a program.

Practically speaking, there is not often a clear cut between a Constraint Programming language, a specific solvers, an SMT solver, and a CAS. Most often a tool encompasses a CAS and specific solvers (Sympy, Maple, ...), or a Constraint Programming language and various solvers including SMT (Z3, Zinc/Minizinc, ...). Moreover some solvers perform satisfiability only and do not include an optimizer, yet it is still possible to do subsequent calls to the solver with decreasing (or increasing) values of the objective function until the solver finds no satisfiable instance.

1.6.3 Using SMT solvers for automated adequation

We insight and we give some evidence that solving adequation problems requires combining several theory solvers, for instance linear programming and SAT solvers.

1.6.3.1 Linear programming aspects of AAA

Since the problem mostly includes inequalities and linear equalities it is a natural idea to model it using linear programming. It is also very attractive since LP problems can be solved in polynomial time, but because we also know that most scheduling problems are NP-Hard this means that they cannot be entirely encoded with linear constraints (otherwise P=NP).

In heterogeneous scheduling problems, the duration of a task depends on which resource executes it, and if multiple operating points exist for this resource then it also depends on which power level this resource runs. This problem can be encoding using a boolean "decision variable" in both cases (it is then quadratic). It can also be encoded using an if-then-else construct. Another example non linear aspect of the problem is the "exclusive or" constraints of non-overlapping two tasks that use the same resource. Encoding such constraints is feasible using an LP solver, but requires to introduce artificial

variables and quantities that must be carefully chosen. For instance, Encoding if-then-else is possible using a “big-M” construct with a binary variable z, and a variable $x \geq 0$: `if z = 0, then x = 0` is equivalent to the linear constraint `x - M * z <= 0` where M is a large enough number. Choosing the value of M is tricky. Recent solvers (such as CPLEX) introduce specific constructs that translate to other theories.

1.6.3.2 Combinatorial optimization aspects of AAA

Task allocation is chosen in the finite set of resources, as well as the power domain of resources. Those variables can be encoded as integers, but it also makes sense to encode them as enumerations and boolean variables since they do not stand for a quantity. Satisfying a boolean formula is the purpose of SAT solvers.

We conclude that the problem is a system of equations which set of equalities and inequalities depends on boolean variables. Since combining theories is the purpose of SAT Modulo Theories solvers [73] we try to encode the problem with an SMT solver instead. Sat Modulo Theories is a very active area of research with available benchmarks[2] and competitions[3] and there exists a plethora of SMT solver implementations (Boolector, Gecode, MathSAT, Yices, Z3). Modern SMT solvers are based on a DPLL (Davis-Putnam-Logemann-Loveland) algorithm which is a complete and backtracking-based search algorithm for deciding if a propositional logic formula (in the conjunctive normal form) is satisfied or not. The basic steps of the DPLL algorithm consist in selecting a variable, assigning a value to it, and propagate this value (i.e. simplify the problem given this decision). If at some point a variable of the problem is in conflict, which means for instance it must be both True and False to satisfy different constraints), then the algorithm backtracks. The sensitive part consists in selecting interesting variables early during the search which is most often decided by heuristic algorithms. It is thus hard to evaluate how a DPLL algorithm handles specific problems in terms of efficiency, but the number of propagations and the number of conflicts are interesting metrics to get an idea if the solver is searching in the right direction.

1.7 Related work

Existing tools must often consists either in evaluating different micro-architecture (platform based design) or performing a high level compilation on an existing micro-architecture (AAA). In this section we mention different tools that target the former, and we focus on the tools that target the latter.

1.7.1 Application modeling

Some of the existing tools focus more on the theoretical aspects of application modeling, and provide theoretical models along with proofs or guarantees such as deadlock free-ness, maximum throughput, bounded memory, ... Most of those models originate from Kahn Process Networks [50] special class of conflict free Petri nets. The SDF^3 tool can generate and transform SDF graphs [88] to other graph models such as the one described in Section 1.2.2.1 and some effort has been developed regarding the memory/throughput trade-off [89]. This KPASSA tool [67] is dedicated to the simulation, analysis, and static scheduling of Event/Marked Graphs, SDF and KRG extensions. A graphical interface allows to edit the Process Networks and their time annotations (latency, ...). Symbolic simulation

[2] `http://www.smtlib.org/`
[3] `http://www.smtcomp.org/`

and graph-theoretic analysis methods allow to compute and optimize static schedules, with memory/throughput trade-offs. Most often those tools provide a graphical syntax or a simplified textual syntax, and connecting the model instances with actual implementations is difficult. The Syndex tool [85] provides a graphical and textual syntax for the modeling. The applications are described as graphs of tasks with a sophisticated modeling of periodic sets of tasks with dependencies. The set of tasks is flattened according to its hyper period as described in section 1.3.2. Task implementations can be provided as a separated set of files (in a general purpose language). Even though the optimization process is the central issue, usability aspects (such as graphical syntax, textual syntax highlighting, assisted completion, verification...) are also important assets of those tools if we wish wide adoption. GNURADIO [10] and GNURADIO companion provide textual and graphical syntaxes for building and running signal processing flowgraphs with configurable tasks, it is limited to acyclic graphs (loops are detected and they raise a runtime error). The tools that offer a graphical modeling environment most often use general purpose data structure languages such as XML or JSON to store and share models with other tools. ArrayOL [12] uses a graphical syntax to describe how signal processing tasks are connected, and a textual syntax to further describe how each of those tasks access data. ArrayOL models can also be transformed to functionally equivalent process network models such as SDF and KPN.

Other approaches for application modeling focus more on the practical aspects of generating a functional implementation [74, 92, 29, 3, 1]. The range of applications that can be described using those frameworks is often limited, at least with predictability issues. Sometimes it even focus on specific problems: Tensorflow is a very recent framework that was originally created to design and implement machine learning algorithms. Tensorflow was then extended to the modeling and implementation of arbitrary task graphs. Halide [79] is a domain specific language designed to implement image processing problems, it focuses on stencil algorithms. SPIRAL [78] is another language designed to better implement signal processing problems with a focus on linear signal processing transform and spectral functions such as the Fast Fourier Transforms. The SPIRAL language allows to express how the routines of the application can be combined or split in order to obtain different implementations. Daedalus is a framework for the Platform based design of MP-SoC based embedded multimedia applications, with such task graph transformation aspects [74]: as like as SPIRAL it allows to explore different functionally equivalent graphs by merging and splitting tasks. Depending on the nature of the tasks, combining or separating them is not always easy. Linear algebra kernels are especially well suited to those transformations. StreamIt [92] is a framework for the design of streaming systems. It provides a domain specific language that is used to instantiate structured task graphs. It allows to combine or split linear kernels. sigmaC [42] is a language and programming model that belongs to the class of process networks and focuses on the properties of deadlock freeness and bounded memory schedulability. In DAGuE (now PaRSEC [102]), the application graph can be generated from a pseudo code language. Using a pseudo code (excluding the behavioral part of the tasks) makes the transformation to a task graph easier but limits the expressiveness and makes the transformation back from a schedule to an implementation harder. YAPI is a C/C++ implementation of an extended KPN model that includes some of the benefits of theoretical models, along with its practical implementation. The COMPAAN compiler investigates the most extreme practical approach of transforming Matlab code into a YAPI model.

1.7.2 Architecture modeling

The platform based design tools level of abstractions range from very abstract component models to detailed description of the hardware with cycle accurate instruction set simulators [6, 4, 26, 96]). The majority of the AAA approaches do not provide a separated modeling language regarding the

architecture resources. The standard approach of implementing a parallel application (using general purpose languages) consists in delegating to the programmer the hard choices of the adequation problem. In this case there is no need for an architecture model. Some frameworks focus on a specific target. For instance StreamIt focuses on the Raw microarchitecture [90] (although there is also a POSIX threads backend), and sigmaC focuses on the Kalray architecture [27].

In the fields of theoretical models architecture constraints are sometimes included in the application graph (eg shared resources in Petri nets [103] or in their conflict-free submodels such as Marked Graphs [68]). In the KPASSA tool, the graphs can be modified to add further limitations that models architecture provisions. The expressiveness is however very limited. Other frameworks such as Syndex and Daedalus offer a graphical syntax to explore multiple architecture designs. Some theoretical model have also be developed for modeling architectures. Some attempts at standardizing architecture modeling exist. MARTE is an UML profile for the modeling and analysis of real time systems, and some efforts at using it in an AAA framework exist [56, 98]. The AADL language is another example of architecture analysis and design language. Those models are very expressive but also cumbersome, which may explain why they have not been widely adopted. IP-XACT [57] is a recent attempt at standardizing the interfaces of industrial IP models to promote their use in modeling, simulation, or optimization frameworks. As well as regarding application modeling, many tools use XML to store and share models, but in practice not many of them use the standard IP-XACT format.

1.7.3 Provisions modeling

Naturally, if the application model does not include functional models of the tasks, evaluating those costs cannot be automated and the designer is expected to provide them. When the user does not provide those costs, they have to be estimated using static analysis techniques or using cycle accurate simulators when they exist. Estimating the duration of a piece of code running on a specific hardware resource is a difficult problem that is known as "Worst Case Estimation Time" (or "Average Case Estimation Time"). StreamIt evaluates task costs on the Raw microarchitecture by using a cycle accurate simulator [41]. It sometimes involves generating several kernels and testing them. The same problem holds for message passing. It is especially tricky in the contexts of multi level cache architectures since moving data between caches cannot be explicit. The Syndex tool allows to specify costs for both computations and communications with an integer value. In Tensorflow the computation and communication costs are evaluated experimentally.

Some tools do not use cost models but instead generate implementations based on admitted optimization techniques (such as the data locality principle), benchmark the generated implementation and perform other optimizations based on this result until an objective is met. This feedback loop technique is referred as "auto-tuning". SPIRAL [78] performs auto tuning for linear algebra problems. The technique performs well when the generated implementation is predictable, which might not always be true in the case of multi core architectures as the experiments of Figure 2.12 suggest.

Runtime systems does not need cost functions neither since they perform dynamic scheduling. This explains for instance why the most famous shared memory runtime system (OpenMP[24]) does not provide an API for cost predictions or cost functions of OpenMP tasks or blocks. Some tools provide both dynamic and static scheduling strategies, for instance XKaapi recently allows to use the HEFT heuristic [65] instead of dynamic scheduling techniques. In this case the cost of tasks is benchmarked and updated at runtime.

1.7.4 Solving the adequation problem

Some of the previously described frameworks are only assembling/simulation tools and do not pretend to optimize an adequation problem or to schedule the tasks. In Daedalus the user can investigate multiple configurations but the search is not automated. In the Platform architect tool the user only has to specify the allocation, and then the schedule for each task is found using an heuristic. PTOLEMY [15] is an environment for simulation and prototyping of heterogeneous systems, including (but not only) embedded and real-time systems. ArrayOL [12] uses PTOLEMY. In YAPI the adequation is a concern of the system designer as well. COMPAAN/Laura and YAPI allow to perform a "workload analysis" in order to chose which tasks will be executed on synthesized hardware, but does not mention how they are chosen and if the decisions are automated. Halide focuses on the separation of functional aspects and sequencing/scheduling aspects, it provides an interface to perform the adequation, but there is not automated search or heuristic.

When the search is automated, it is most often achieved using an heuristic In Tensorflow the adequation problem is solved with an "ongoing development" heuristic. The Syndex tool makes use of greedy algorithms and specific algorithms for sets of periodic tasks (it distinguishes a transient solution and a steady state behavior that validates the real-time requirements). The Lopht tool [18] also uses a list scheduling heuristic to perform allocation and scheduling. One step of the sigmaC compiler consists in modifying the application graph so as to fit architecture provisions. We could not find how exactly the transformation is done and base on which metrics. Since sigmaC targets Kalray this information is probably undisclosed. SPIRAL provides an automated search heuristic in this design space, with some support for code generation for SMP architectures. We could not find attempts (heuristics or backtracking algorithms) to automatically transform ArrayOL specifications based on architecture resource provisions.

Since we aim at program optimization and not platform based design, we do not consider the frameworks that include hardware synthesis (such as POLIS, [6], CODEF [4], COSYN[26], COWARE[96], ForSyDe[81]) although they share some of the techniques and models that we describe thereafter. Some frameworks do not embed any solver either for allocation or scheduling, but they offer a development environment that facilitates the construction of such system, potentially including simulators, testing tools, synthesis tools. For instance the framework sometimes requires that the allocation is specified by the designer, along with the scheduling policy (it offers a set of heuristics to choose from) [64]. Because we specifically tackle the adequation problem, we don't further investigate such tools.

The SDF3 tool also provides an allocation algorithm for multiprocessor architectures [86, 87], and then applies scheduling for each processor. Considering separately the allocation and the scheduling problems is a classical approach to the problem, which has a much lower practical complexity but is not optimal regarding the adequation problem.

A recent work [91] investigates the use of SMT solvers for the makespan optimization of dataflow graphs on network on chip architectures.

1.8 Our contributions

This thesis is a deliberate attempt at studying to which extent computer algebra systems and existing solvers can be used to tackle the adequation problem in the case of joint performance/energy consumption modeling. This concretely means that we had to seek system descriptions and encodings that were as simple as possible while keeping relevant for our intent, and as fit as possible regarding the existing symbolic mathematical languages and their associated solvers.

We chose to consider a typical big.LITTLE execution platform since the problems of allocation

decisions between the big and little cores seemed both to fit our modeling and analysis purposes, and to connect our work to an important current design issue in smartphones and similar connected objects. We present our general approach that will be detailed in the following chapters, and we describe the example in very broad terms. A simple use case provides the various aspects of modeling and their counterparts in mathematical expressions such that the reader gets a first intuition of how the various notions discussed before translate in our approach.

1.8.1 The SymSched tool, a general presentation

We described in a non exhaustive way the modeling elements often found in the AAA approach. In the next chapter we shall specialize such approach for out own goal, which is to study how far general SMT (SAT (satisfiability) Modulo Theories) solvers can be used to deal with the cases including allocation alternatives depending on distinct power and performance consumption of processor models. We borrow some of the ideas and models from the real-time scheduling theory, and we give some evidence that those models are close to practical parallel embedded devices. We generate task graphs from pseudo code or other formal models such as SDF graphs, and we show how to characterize the agents regarding performance.

We implemented the SymSched tool which allows to solve the adequation problem using an SMT solver. Figure 1.11 synthesizes the workflow of the tool.

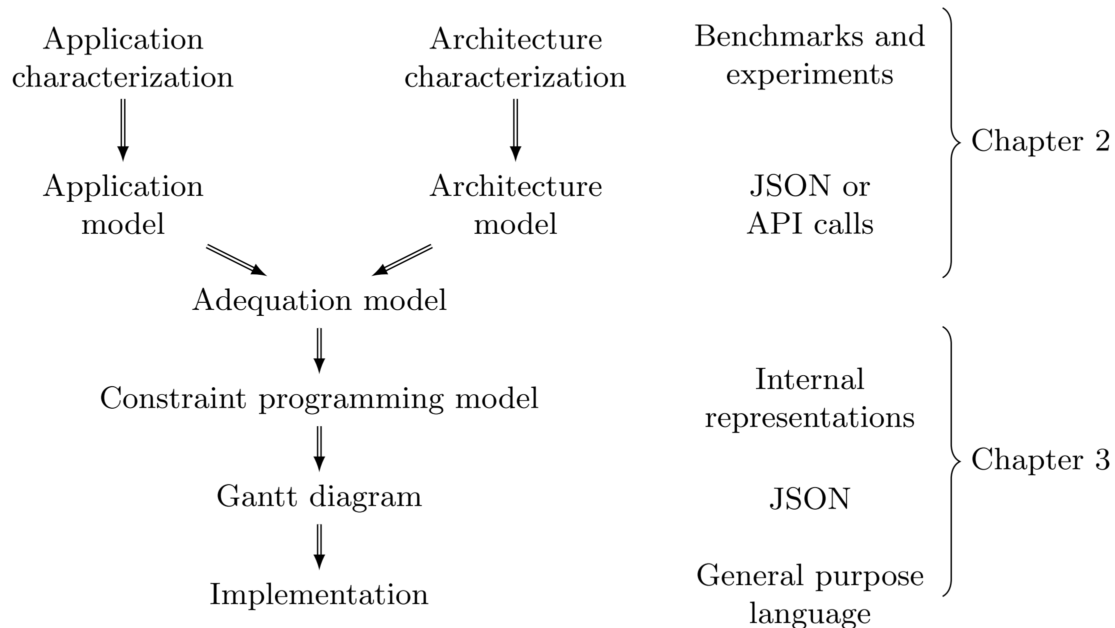


Figure 1.11: Detailed workflow of our approach

1.8.2 A running example

We illustrate our approach on a simple instance of the problem studied in section 3.5.1. In this example we study the adequation of a Cholesky factorization application on an heterogeneous MPSoC architecture made of one fast power consuming core and one slow power efficient core.

The architecture is an implementation of the ARM big.LITTLE technology (Figure 1.13) which combines the A7 ARM architecture (slow and power efficient cores) with the A15 architecture (high energy and computation efficient core). The Cholesky factorization admits a recursive definition which calls four different BLAS lvl3 matrix operation kernels (GEMM, TRSM, POTRF and SYRK). We use a pseudo-code of this recursive algorithm to build a DAG of the tasks (Figure 1.12). Then we measure the cost of each kernel on both resources, and we find out as expected that this is an heterogeneous scheduling problem (Table 1.1). In this simple example we consider that both cores run at 1GHz.

Table 1.1: Architecture provisions for the tiled Cholesky application (10^6 cycles)

	GEMM	SYRK	TRSM	POTRF
A15	11	6	7	9
A7	39	20	21	16

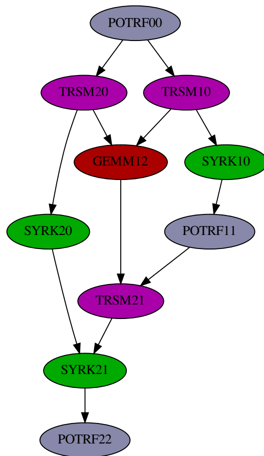


Figure 1.12: 3x3 tiled Cholesky task graph

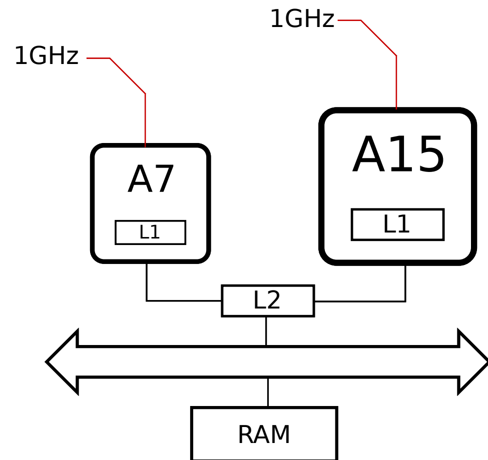


Figure 1.13: A simplified big.LITTLE architecture

Given those provisions, we encode the problem using a computer algebra system (which is the interface to the z3 solver) and we optimize with a makespan objective function.

Application constraints:

$$POTRF_{00}.stop \leq TRSM_{20}.start$$
$$POTRF_{00}.stop \leq TRSM_{10}.start$$
$$TRSM_{20}.stop \leq GEMM_{12}.start$$
$$\vdots$$

Provisions:

$$\begin{aligned} GEMM_{12}.duration = 39 \cdot (GEMM_{12}.map == LITTLE) \\ + 11 \cdot (GEMM_{12}.map == BIG) \end{aligned}$$
$$\vdots$$

Allocation constraints:

$$TRSM20.map == TRSM10.map \implies (TRSM20.stop \leq TRSM10.start)$$
$$\oplus (TRSM10.stop \leq TRSM20.start)$$
$$\vdots$$

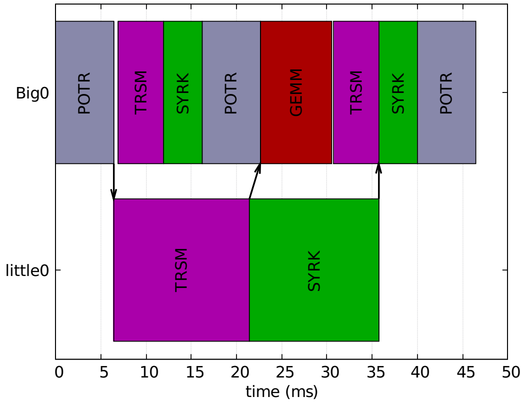


Figure 1.14: Makespan optimal solution

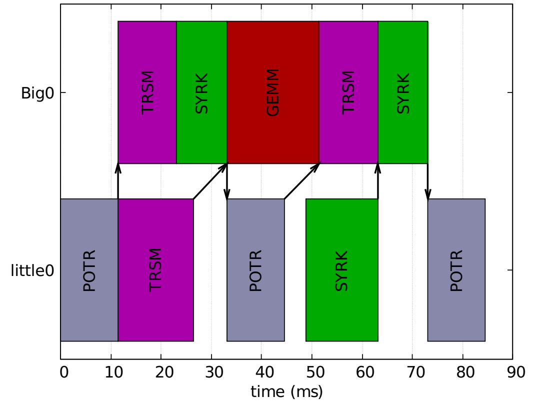


Figure 1.15: Energy optimized solution

We choose to return a JSON structure containing all the determined values such that we can later either generate code or build different types of diagrams (Section 1.5.5). We use an external tool to transform this JSON output into a Gantt diagram (Figure 1.14). Then we measure the power levels of the big and LITTLE resources when idle or active, and we add them to the existing set of equations and inequations. Then we give some slack time based on the previously obtained optimal makespan, and we obtain energy/performance trade off schedules (Figures 1.15 and 1.14). As expected, saving power requires to execute more tasks on the LITTLE core. The final step of the workflow consists in translating those schedules back to a parallel implementation. Figure 1.16 gives an example generated pseudo-code.

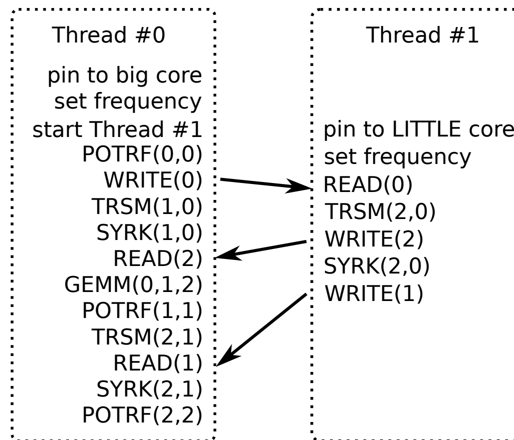


Figure 1.16: Threads, pseudo-code and message passing for the 3x3 makespan optimal solution

Chapter 2

SymSched modeling and model translation to system of constraints

In this chapter we describe the syntactic elements that will compose our modeling approach entitled SymSched (scheduling using symbolic expressions). They specialize the modeling notions encountered and motivated in the previous chapter, since the objective remains to be able to express relations (requirements and provisions) between those elements. We focus on searching a proper representation level leading to mathematical expressions then amenable to existing solvers such that the adequation search can be automated. Still we shall try to maintain the main modeling categories (Application, Architecture and Allocation) fully visible in the axiomatisation of constraints and to represent clearly the aspects of allocation costs and adequation realization as separate models. Our modeling framework will thus rely on Acyclic Precedence Graphs APG obtained by unfolding SDF models, augmented with data-parallel and preemption construction features.

Our constraint sets will consist of a mix of boolean conditions (reflecting the distinct mapping alternatives) and linear inequalities (reflecting the processing durations and energy consumption). We shall describe in turn how the logical task/message dependencies, the real-time requirements, the discrete power domains and allowed operating performance points combinations will contribute to the constraint sets. Depending on the application task graph (or on the requirements such as periodicity), an agent can run multiple times. Special notice should be made of the fact that since SMT solvers deal with a finite number of variables, our applications modeling will consider tasks that are single execution occurrences, and that otherwise iterative application models will have to be unfolded an arbitrary number of times. Then we can analyze this iteration and synthesize a system by concatenating successive optimized iterations. We call each agent run a task which is the atomic component mapped to a processing resource. This approach could be compared to the original process of finite unfolding already present in the early days of bounded model-checking when SAT-solvers were first introduced for verification of synchronous hardware.

More generally, the dedication to using hopefully efficient SMT solvers leads us to impose a number of simplification restrictions to the description of applications (as compared to the general Models of Computation described in the previous chapter). We describe these assumptions and the resulting modeling constructs for applications in Section 2.1. We justify our choices by the opportunity they offer to obtain “sensible” schedules. Still, the restricted expressiveness on the application side allows us to account, even if admittedly in an limited way, for a number of phenomena. The UML class diagram (Figure 2.1) depicts how the main elements of our model are related to each other, and some of the most relevant attributes.

Section 2.2 describes our architectural model which consists essentially of processing resources on

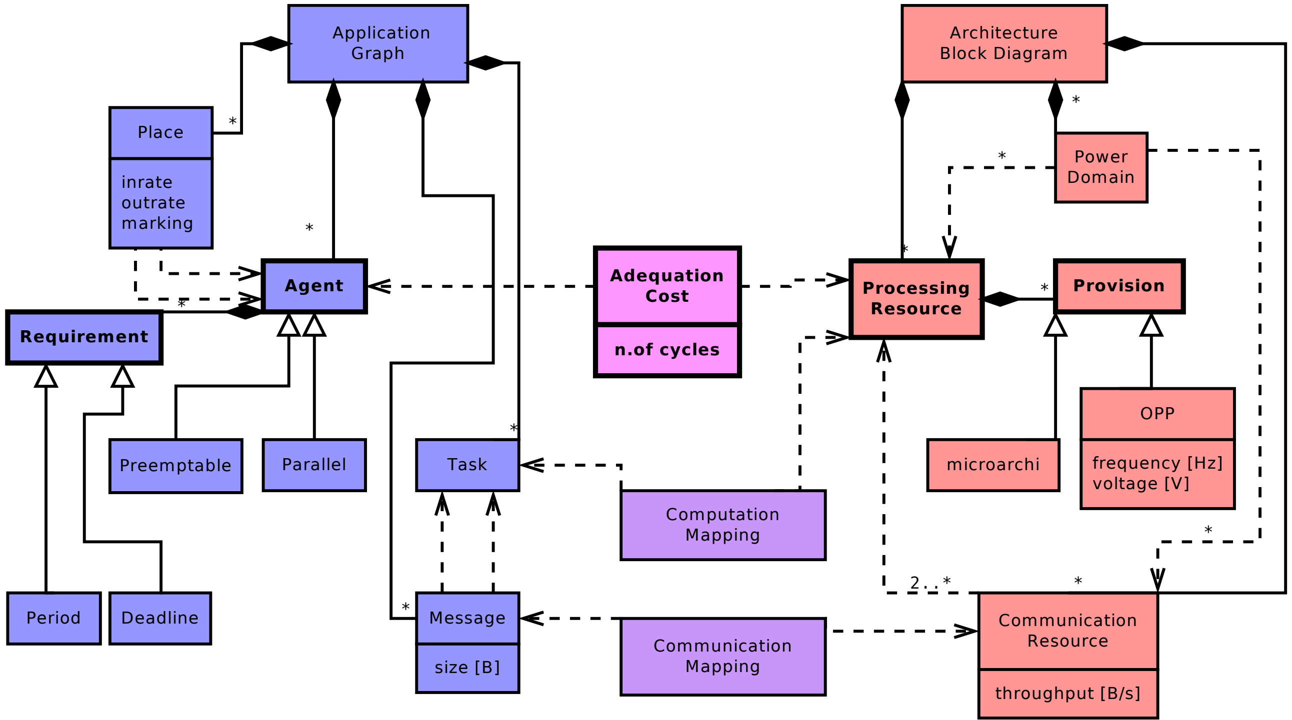


Figure 2.1: UML diagram of the SymSched tool

which application tasks are to be mapped, and communication resources which will receive application messages whenever these inter task communications and data transfers are made in between tasks allocated to distinct processing resources. Each potential such mapping will incur a cost that relates the "logical time" duration at application level expressed in processing cycles to the resulting "physical time" duration obtained when mapped to a physical architecture resource. In turn, these costs will themselves depend on parameters representing the various performance and energy levels available, corresponding to so called Operating Performance Points (the frequency and voltage tuples admissible for the resources). This general scheme of cost, and how it can be used to contribute effectively to the construction of a system of constraints submitted to an SMT solver will be further described in section 2.3. We consider also the issue of joint representation of performance/energy at this stage and distinction between complete power shutdown and sleep mode of a core in a multicore architecture can also be represented. The more general issues of representing cost functions which effectively link application-to-architecture components is dealt with in section 2.3.

Thereafter we detail the classes that we use to build a SymSched scheduling problem and their most significant attributes. Since we use a solver to determine each task start, stop, operating frequency and mapping, pre-processing the problem such that it can be described in a set of tasks, messages and precedences is a mandatory step of our approach. We try to keep the modeling as orthogonal as possible, with the blue part modeling software components, the red part modeling hardware components. Since the Adequation analysis consists in combining them we have to connect them through a cost function (light purple) and the model includes the result of our analysis in the form of computation mappings and communication mappings which are initially unknown (dark purple). The bold classes highlight the transverse classes of the problem.

Notations We use both the mathematical and the object oriented notations to navigate in an instance model of the meta-model described in Figure 2.1. For example both $t.start$ and t_{start} stand for the same variable, and $m.receiver.start$ is the $start$ attribute of the task which receives message m.

2.1 SymSched application model

In this section we describe the application models and their corresponding mathematical models. Encoding them to match our framework (mainly entering as solver input) sometimes raises theoretical or practical issues that we try to highlight on the way.

2.1.1 SymSched task graph meta models

According to the UML diagram of Figure 2.1 an application is made of instances of agents, places, tasks and messages. The computation and communication mappings however reference only tasks and messages, since they are the basic components of the solver input. We thus need to perform a model transformation from a set of agents and places which the user specifies to a set of tasks and messages which the solver uses as inputs. The set of tasks and messages is later transformed to a constraint programming problem (Algorithm 1).

To stick with our definitions of agents and tasks (a task executes only once) we use both an SDF and an APG (Acyclic Precedence Graph) meta model, and we implement the classical transformation from SDF to APG. The algorithm is similar than the one developed in the KPASSA tool [67] adapted to SDF graphs. In addition to acyclic task graphs we shall also consider (data)-parallel process constructs and the ability to specify preemptable agents (by splitting them in chunks) which we model using the "Preemptable" and "Parallel" Agent subclasses (Sections 2.1.1.4 and 2.1.1.5). We describe how each element of the models translates into equations (2.2, 2.3, 2.4, 2.5, 2.6). We motivate the modeling of data parallelism, preemption and periodicity in Section 2.3.

2.1.1.1 Agents and Tasks

Task are the basic components that allow to model acyclic dataflow graphs of tasks. The Task class is made of the previously defined attributes for non-preemptive tasks (Equations 1.2). The values of those task attributes (start, stop, duration) are timestamps or durations expressed in physical time (in seconds) and are variables of the problem (Equation 2.1). The semantics is the same than the one described in the classical models of real-time scheduling theory.

$$Task\ T : T.stop = T.start + T.duration \tag{2.1}$$

Given an SDF graph we solve the balance equations defined in 1.1 to determine how many times each agent fires for one period of the graph. Then we instantiate as many task instances and connect them according to the input/output rates of the places and their marking. The transform from SDF to APG exists in the literature [60]. Acyclic graph of tasks can be generated from another input such as pseudo-code.

Tasks can be ordered with precedences for functional reasons which translate into the inequality of Equation 2.2 or can send and receive data (Equation 2.3).

$$\begin{gathered} A, B \in Tasks \\ precedes(A, B) : A.stop \leq B.start \end{gathered} \tag{2.2}$$

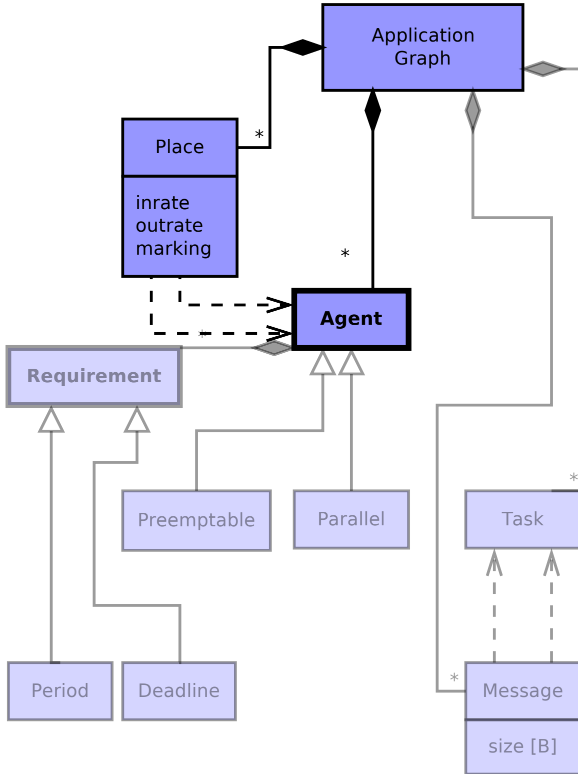


Figure 2.2: SDF meta-model

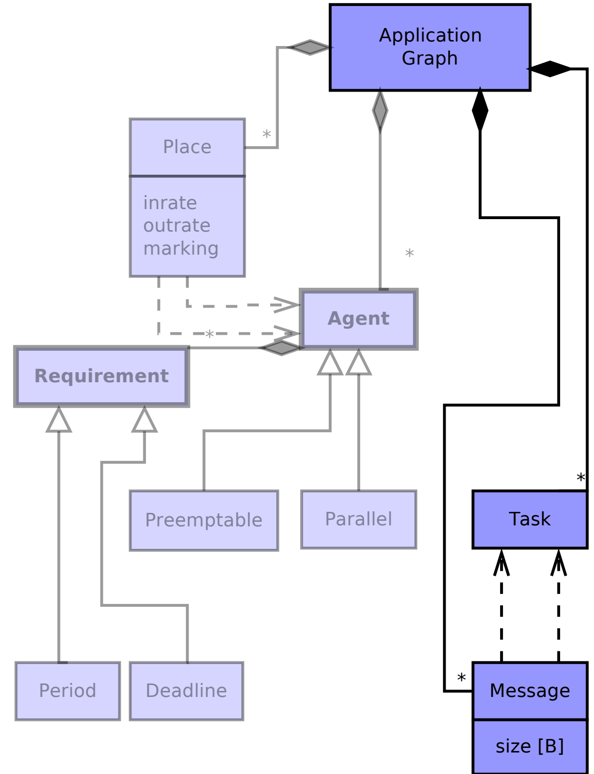


Figure 2.3: APG (or Task Graph) meta-model

2.1.1.2 Places and Messages

Messages can model additional communication costs between tasks. Messages model communication occurrences in Places in the same way a Task models an Agent occurrence. A message translates to two precedences and a time interval (which duration depends on the message size, for instance such as described in section 1.4.1.2 or in Equation 2.12). A message with size 0 is transformed to a single precedence. In this model we consider that a task can start only when all its inputs are available, or when all its preceding tasks have been executed.

$$
\begin{aligned}
&A, B \in Tasks, M \in Messages \\
&Send(A, B, M) : \begin{cases} A.stop \leq M.start \\ M.start + M.duration = M.stop \\ M.stop \leq B.start \end{cases}
\end{aligned}
\tag{2.3}
$$

2.1.1.3 Loop constructs

Using the SDF model allows to consider infinite loop constructs of streaming application models such as Figure 2.4 . The transformation to an APG in this case is still possible if the graph is balanced (see Section 1.2.2.3) but is limited to a bounded number of periods. Considering multiple iterations of the loop allows to benefit from the pipelining effect when optimizing makespan.

In the example of Figure 2.4, increasing the number of periods increases the depth of the APG graph. If the SDF graph is balanced, it has no deadlocks and the APG can be obtained by symbolic simulation of the SDFG until the desired period is reached. Note that the marking also plays an essen-

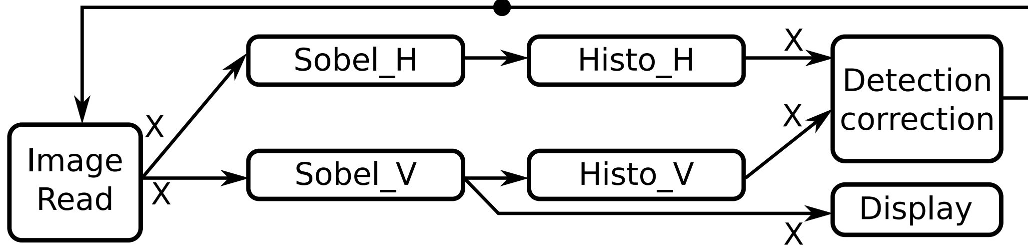
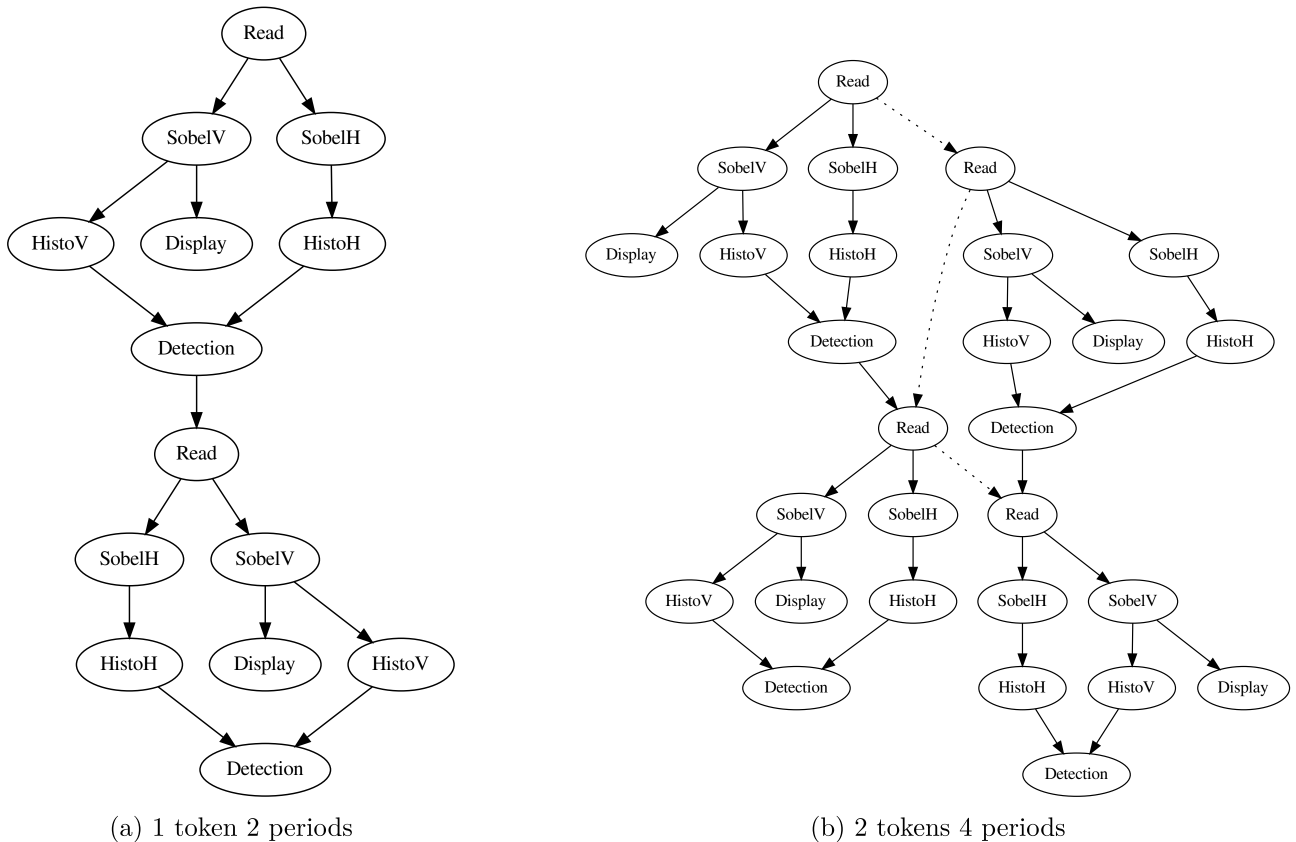


Figure 2.4: A marked and live SDF graph

tial role when considering the shape of the generated APG graph (and thus the available parallelism). SymSched allows to explore different configuration for such graphs. We generate examples in Figure 2.5. We can see in this example that buffering more tokens for the Read agent increases the potential parallelism (the breadth of the graph raises). Nodes indicate tasks and plain lines indicate messages. Graph 2.5b is actually disconnected, dotted lines order the occurrences of the Read agent.



(a) 1 token 2 periods

(b) 2 tokens 4 periods

Figure 2.5: Platooning APG graphs

Finite loop constructs can also be used when considering pseudo code as an input model instead of SDF graphs as will be shown later in the Example described in Section 3.5.1.

2.1.1.4 Parallel agents

We consider a simple split/join task graph with a total amount of work ω and a total amount of communication γ. The balanced, and optimal makespan for such problem consists in dividing both by N the number of resources: $\frac{\omega+\gamma}{N}$. Even in the case one master node holds the data (and thus has no communication overhead), we can find analytically the optimal makespan (given N resources the optimal makespan is $\frac{\omega+(N-1)\cdot\gamma}{N}$). Parallel agents allow to encode such data-parallel problem and use it in a more general context.

Parallel agents often match the problems for which the data can be partitioned and processed separately, although sometimes there might exist data dependencies (typically on the border of the partitions), or some data may be duplicated and sent to multiple workers (cellular automata, image processing filters, numerical solvers, ...). An example is the Sobel edge detection image processing algorithm where each pixel can be processed independently provided it can access the value of the 8 neighbors pixels. We implemented the ability to share the work associated to a parallel agent among multiple resources, without sophisticated data access modeling.

A parallel agent is split into a set of N_t tasks $t_i \mid i \in 0..N_t - 1$ (or "workers") with the corresponding messages and precedences. Let a parallel agent of cost $\mathcal{C}$, scheduled on an architecture of N_r homogeneous resources. The parallel agent model allows to map each task with a round robin policy when the "distributed" boolean attribute is set: $t_i.map = i \mod N_r$. The "dynamic" boolean attribute allows to use variable duration workers: when it is False, each task costs $\frac{\mathcal{C}}{N_r}$. When it is True the task durations are variables of the problem.

$$Parallel(\mathcal{C}, distributed, dynamic) : \begin{cases} \sum_{i=0}^{N_t-1}(t_i.cost) = \mathcal{C} \\ distributed \implies \forall i \in [0, N_t[: t_i.map = i \bmod N_r \\ \neg dynamic \implies \forall i \in [0, N_t[: t_i.cost = \frac{\mathcal{C}}{N_t} \end{cases} \tag{2.4}$$

2.1.1.5 Preemptable agents

We previously defined preemption and migration in the general context of theoretical scheduling (in Section 2.1.1.5). We describe our model of preemption which is a pipeline of tasks that precedes each other. A preemptable agent can receive and send messages: when it is transformed to instances, the first instance receives the incoming messages and the last instance emits the outgoing messages.

Let N_t be the number of generated instances. Again, we define attributes in order to allow different allocation policies and task models for preemptable agents: "migration" is a boolean that specifies if the pipeline is allowed to migrate its tasks among available resources, if it is False then constraint defined the additional constraint $\forall i \in [0, N_t[: t_i.map = t_0.map$ must hold. "dynamic" is a boolean that defines the cost of the generated instances. As well as for parallel agents, the cost is set to $\frac{\mathcal{C}}{N_t}$. When it is False the cost of each generated task is $\frac{\mathcal{C}}{N_r}$. Finally, since it is hard to specify a significant N_t this value can be set. If migration is allowed, allowing the task to migrate at least once on each resource is not often necessary.

$$Preemptable(\mathcal{C}, migration, dynamic) : \begin{cases} \sum_{i=0}^{N_t-1}(t_i.cost) = \mathcal{C} \\ \neg migration \implies \forall i \in [0, N_t[: t_i.map = t_0.map \\ \neg dynamic \implies \forall i \in [0, N_t[: t_i.cost = \frac{\mathcal{C}}{N_t} \end{cases} \tag{2.5}$$

2.1.2 Requirements

Application requirements can be of different nature. Most often a performance requirement is expressed in the specification as a throughput, or a bounded execution time for a given set of tasks or a given graph. Typical examples include audio or video decoding. This is achieved by bounding the makespan variable defined in 1.5.

2.1.2.1 Periodicity

Periodicity requirements demand further analysis. We define periodicity of an Agent with a period p and a deadline d. Periods and deadlines are physical time quantities. When no deadline attribute is specified, a periodic agent is transformed to a set of agents $t_i \mid i \in 0..N-1$ with appearance and deadline attributes set according to Equation 2.6. If a deadline attribute is specified, then it is set for each agent with respect to its appearance. Those are the two classical models for periodic events in theoretical scheduling.

$$Periodic(p,d) : \begin{cases} \forall t_i : \\ t_i.appearance = i \cdot a \\ t_i.deadline = (i+1) \cdot p \end{cases} \tag{2.6}$$

2.1.2.2 Objective functions

An application specification can give other requirements, in different forms such as deadlines in physical time. Most often the specification implicitly refers to an objective function. For instance a video decoding application with a frame rate requirement of 20 frames per second specifies that the decoding of one frame, thus the makespan of the graph corresponding to the decoding of one frame should not last more than 50ms. If the application performs both video and sound decoding, then maybe both problems have different throughput requirements.

When considering performance optimization only, the classical trade-off for mapping is that computation tasks would benefit from using all processing resources in parallel, except that this may introduce additional communications to map on interconnect resources, that at some point defeat the gain of processing parallelism. We add the concern for low-power consumption, so that power-efficient mappings are sought, under performance threshold (optimizing only consumption is meaningless as it leads to use systematically one resource only with the lowest consumption rate).

2.1.3 Application transformation to a solver input

We declare each attribute of task as a new variable and we aggregate the equations 1.2, including specialized agents defined in the previous sections. When additional information is known (such as the mapping of certain tasks) we can also add it to the model: the solver is proficient at propagating values, declaring variables and setting them to constant values does not increase practical complexity. Propagating values and inequalities in the z3 solver can be achieved using the two tactics “propagate-values” and “propagate-ineqs”. “Repeat” and “Then” functions keep on applying those tactics one after the other as long as no other value or inequality can be propagated.

```
1  from symsched import *
2  from z3 import *
3
4  p = Problem()
```

```
5   r = Resource(label='CPU', microarchi='arm')
6   r.add_opp(Opp('1GHz','1V'))
7   p.add_resource(r)
8
9   A = Agent(label='A', deadline='1s')
10  B = Agent(label='B', deadline='100ms')
11  p.add_agents([A, B])
12
13  p.add_cost(A, r,'1MCy')
14  p.add_cost(B, r,'3MCy')
15
16  p.flatten()
17  c = reduce(And,p.constraints())
18  s = Repeat(Then(Tactic('propagate-values'),Tactic('propagate-ineqs')))
19  c = s(c)
20  print c
```

In this example there is only one resource that runs only at one frequency. Many variables are actually determined: by propagating values and inequalities (line 18) we obtain the simple set of constraints:

```
makespan ≤ 1000
A_start >= 0
B_start >= 0
(A_stop <= B_start) != (B_stop <= A_start)
A_stop ==  A_start + 1
B_stop == B_start + 3,
A_stop <= makespan
B_stop <= makespan
```

It is "easy" to solve such problem (even if it is not automated). If we add other agents to the model such as a preemptable agent which is split into two instances of variable durations tasks (it is preempted one) such as:

```
C = Preemptable("C",1,dynamic=True, deadline="1s")
```

We obtain the expected additional constraints:

```
C_0_start >= 0
C_1_start >= 0
C_0_stop <= C_1_start
C_1_start <= makespan
C_0_duration + C_1_duration = 10
(C_0_stop <= B_start) != (B_stop <= C_0_start)
(C_1_stop <= B_start) != (B_stop <= C_1_start)
```

With the increasing number of variables and expressions, the problem becomes harder and some automation is welcome in order to find a satisfiable or optimized solution.

Algorithm 1 sums up the encoding of instances and messages. Let:

- R: the set of resources

- O: the set of operating points
- $\mathcal{T}$: the set of tasks. $t \in \mathcal{T}$ is a tuple: $(t_{start} \in \mathbb{N}, t_{stop} \in \mathbb{N}, t_{duration} \in \mathbb{N}, t_{map} \in R, t_{opp} \in O)$
- M: the set of messages (or precedences). $m \in M$ is a tuple: $(sender \in \mathcal{T}, receiver \in \mathcal{T}, size \in \mathbb{N})$
- $costs \in \mathbb{N}^{T \times R \times O}$ is a table that defines the cost of each combination of task, resource, and opp
- $timebudget \in \mathbb{N}$ is specified. $makespan$ may be the objective function.

Algorithm 1 Application constraints

Require: $timebudget$
  for $t \in \mathcal{T}$ **do**
    $t.stop = t.start + t.duration$
    $t.stop \leq makespan$
  end for
  $makespan \leq timebudget$
  for $m \in M$ **do**
    $m.start \geq m.sender.stop$
    $m.start + m.duration = m.stop$
    $m.receiver.start \geq m.stop$
  end for

2.2 SymSched Architecture model

In this section we define the behavioral and structural parts of the architecture model used in the SymSched tool (Figure 2.6) We detail in Section 2.2.1 the processing resource model, in Section 2.2.2 the investigates model of communications and in Section 2.2.3 we explain how we characterize the architectures regarding energy and computation throughput.

We define the "disjunctive" predicate (Equation 2.8) which we will use in our modeling. The predicate applies to two intervals of time (with start and stop attributes).

$$\begin{aligned} &X, Y \text{ intervals of time:} \\ disjunctive(X, Y) = (X.stop \leq Y.start) &\oplus (Y.stop \leq X.start) \end{aligned} \tag{2.7}$$

2.2.1 Processing resources

In the SymSched model a resource is an entity able to run tasks sequentially. If two tasks use the same resource they cannot overlap in time. Because in our model a task cannot be split (although a preemptable agent can be split into multiple tasks), when two tasks run on the same resource it means that one of them starts after the other stops.

We encode this using a classical "disjuncitve" constraint which we apply for each resource. Equation 2.8 gives its encoding. It is interesting to notice that this constraint makes the state space non convex, which also explains why the adequation problem is actually NP-Hard and cannot be solved alone as we suggested in Section 1.6.3.1.

$$\forall r \in R : (T_0.map == r) \wedge (T_1.map == r) \implies disjunctive(T_0, T_1) \tag{2.8}$$

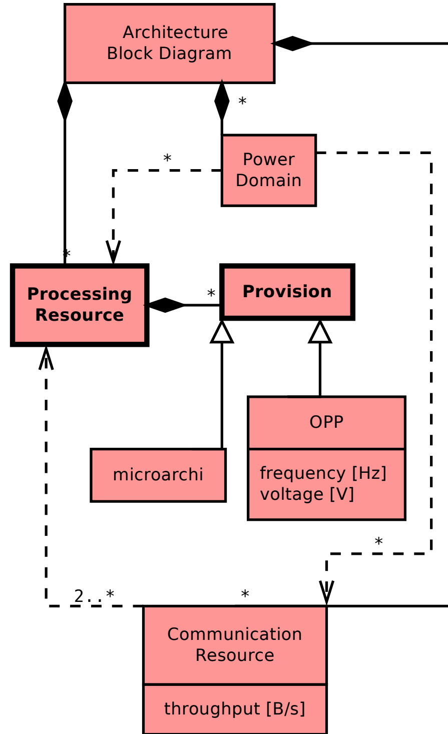


Figure 2.6: SymSched architecture model

Moreover in the SymSched model a resource includes some amount of private memory, which can only be accessed by this resource. We consider that when an task executes it accesses only the data inside this private memory. Thus before the task starts it needs to receive all its incoming messages. Although this model does not match CPU and cache typical architectures we give some evidence in Section 2.3.2 that few interference happen between memory accesses of two kernels running in parallel when they do not stress a large memory area.

Resources are associated with one or multiple OPPs (Operating points). An OPP is defined by a voltage and a frequency, it models how fast a task is executed and how much power it consumes to execute it. In our model the OPP can change when the resource is idle, but not when a task executes. However successive tasks can run at different OPPs. Thus we can define a preemptable agent that splits in tasks which run at different frequencies.

2.2.2 Communication resources

The topic of communication modeling is hard to investigate with practical experiments since the user has often very little control on the interconnect. Most of the available multicore embedded development boards implement the typical memory hierarchy that consists in one addressable memory and a set of caches. In shared memory architectures modeling the interconnect can be used to estimate cost accesses to the main memory or to mimic the behavior of on chip interconnect even though most often accessing it is not explicit. We first address Crossbar and Bus modeling as they are very generic interconnect architectures.

We model messages as a quantity of data (in Bytes) that is produced by a task and consumed by another task as described in section 2.1.1 and we consider the

Crossbar A crossbar models a N to N communication device with no interference and with a fixed

throughput

Bus A bus models a N to N communication device with fixed throughput that must be used exclusively

Routes A route models a 1 to 1 communication path which uses multiple communication resources

The Crossbar model generates no additional constraints. The Bus model generates a disjunctive constraints for all combination of messages (Equation 2.9). Modeling an interconnect made of links and routers requires more efforts.

$$Bus: \ \forall (m_0, m_1) \in \binom{M}{2} : disjunctive(m_0, m_1) \tag{2.9}$$

Distributed memory architectures also exist but programming them requires more effort. They have long been used to connect different computers and not on-chip resources. Such computer architecture also exists on chip and is known as a "Network on Chip" or a "Massively Parallel Processor Arrays". Modeling communication costs in a distributed memory architectures is essential and allows further optimization by giving the ability to the solver to overlap computation and communication costs. However since distributed memory is not the mainstream architecture in the embedded system areas we did not fully develop this model. We however experimented with the notion of routes which we describe thereafter.

If the architecture has only one route for each different pair of resources then the task allocations is sufficient to map a message to a route. If the architecture has multiple links and routers, an additional decision variable is necessary. In our experiment we used a precomputed table of conflicting routes (function $\mathcal{C}$) to determine if two routes share a communication resource. Let $\mathcal{R}$ be the set of routes and M the set of messages and $m.r$ the route used for message $m \in M$. The modeling of the interconnect consists in Equation 2.10.

$$\begin{aligned} &\mathcal{C} : \mathcal{R} \times \mathcal{R} \rightarrow \{True, False\} \\ Interconnect: \ \forall (m_0, m_1) \in \binom{M}{2} &: \mathcal{C}(m_0.map, m_1.map) \implies disjunctive(m_0, m_1) \end{aligned} \tag{2.10}$$

Figure 2.7 gives an example of tasks and messages statically mapped to a 3-resources (R_0, R_1, R_2) ring architecture. In this example tasks A and D are mapped to the same resource R_0 (and thus they do not need to exchange the message through the ring interconnect), B is mapped to R_1 thus the message from A to B is mapped to the route $R_0 - R_1$, C is mapped to R_2 and the message from A to C is mapped to the route $R_0 - R_2$. An additional decision variable can be used to model clockwise and counterclockwise routing.

We were able to experiment with the Kalray MPPA platform[55] but were not able (or allowed) to control the on-chip routers. Moreover we did not find any evidence that conflicting routes (such as in Figure 2.8) degrade bandwidth which suggests that the runtime routing policy includes a bandwidth reservation mechanism. This means that while this mechanism is active there is no point taking care of overlapping routes in the adequation process. An interesting approach regarding off line scheduling using SMT solvers would consist in considering only a subset of the available routes such as only XY and YX routes. We did not investigate such problem.

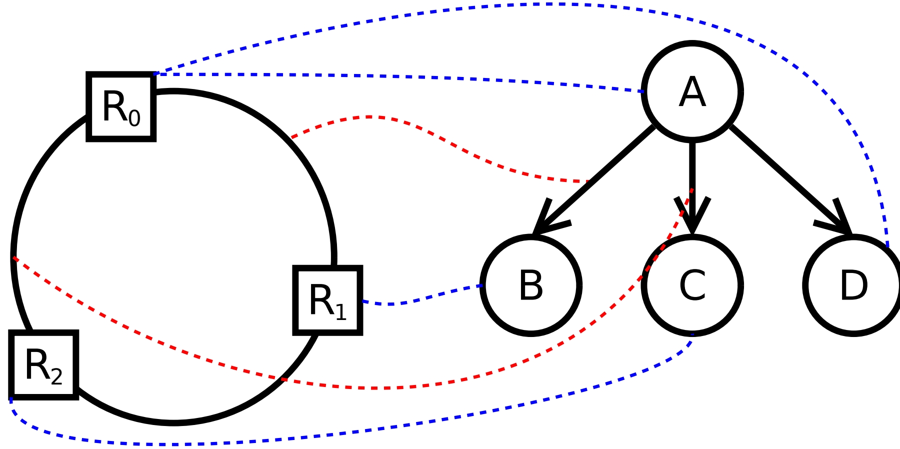


Figure 2.7: Static mapping of messages on the links of a ring architecture

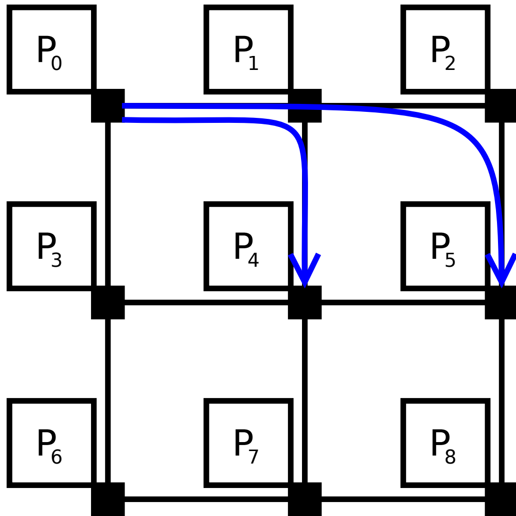


Figure 2.8: Conflicting routes on a 3x3 mesh network on chip

Regarding network on chip architectures, enumerating the possible routes to encode their conflicts using a constraint programming approach is cumbersome and we shall consider only a subset. However it is reasonable to implement it this way for simpler memory hierarchies such as hierarchical memories (Figure 2.9)

2.2.3 Provisions

A natural idea consists in trying to characterize the performance of a machine regardless which kernel it is running. In this section we discuss how we characterize the architecture regarding computing throughput, communication throughput and energy consumption.

2.2.3.1 Computing and communication throughput

In SymSched we consider fixed computation and communication throughput. A task is associated to an OPP (thus a frequency) and a resource (thus a microarchitecture). Instead of trying to find analytic cost models for both computations and memory accesses (which is an already existing and

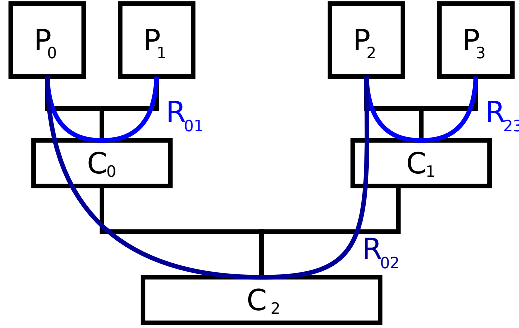


Figure 2.9: A two by two hierarchical interconnect

complex topic of research [101]) we try to consider them separately. We assume that running a data-independent kernel on a resource in a private memory area lasts a predictable number of cycles, and we model the communications that occur in shared memory areas separately. We give some evidence of this orthogonality of the models in Section 2.3.2.

In a first attempt at characterizing resource microarchitecture we executed some benchmarks to measure Flops (floating point operations per second) or MIPS (million instructions per second) on the example platform ARM big.LITTLE described in Section 3.5.2. The Dhrystone benchmark [100] for instance measures MIPS [1]. On the Cortex-A7 and Cortex-A15 architecture it reports (resp.) 1.9 DMPIS/MHz and 3.5 DMPIS/MHz. We could thus consider that those resources are related regarding performance with a ratio of $\frac{3.5}{1.9}$. Further experiments show on contrary that the ratio varies depending on the kernel (see Table 3.4) which puts into perspective using such benchmarks to compare micro-architectures. Because of this heterogeneity, other benchmarks have been developed such as Whetstone. Dhrystone focuses on integer operations, while Whetstone does both integer and floating point operations.

Memory benchmarks also exist both for shared memory and distributed memory architectures. They can be used to estimate the performance of memory operations or message passing [34] depending on message size. We did no such experiment.

2.2.3.2 Power Domains

Regarding power estimations, because not many platforms are equipped with instant power sensors it is generally hard to investigate. The Odroid XU3 has power sensors for each A15 core, and for the integrated GPU chip which allowed us to run some experiments. The coarse precision and low sampling rate[2] of the sensors where however a limitation for further experiments. However this allowed us to obtain realistic instant power values for our adequation problem. Figure 2.10 shows the instant power consumption of an A15 ARM Core vs its frequency when running a CPU intensive kernel, and when idling.

Table 2.1 gives the measured values. We artificially stressed the A7 and A15 cores using the “stress” and “taskset” utilities (Figure 2.10). We observe that the A7 cores have smaller instant power for a given operating frequency. We remind that a low instant power is not a guarantee of low energy consumption since a computation kernel can perform differently depending on the microarchitecture.

[1] Dhrystone MIPS

[2] set to 263808μs in the kernel and cannot be configured without building it

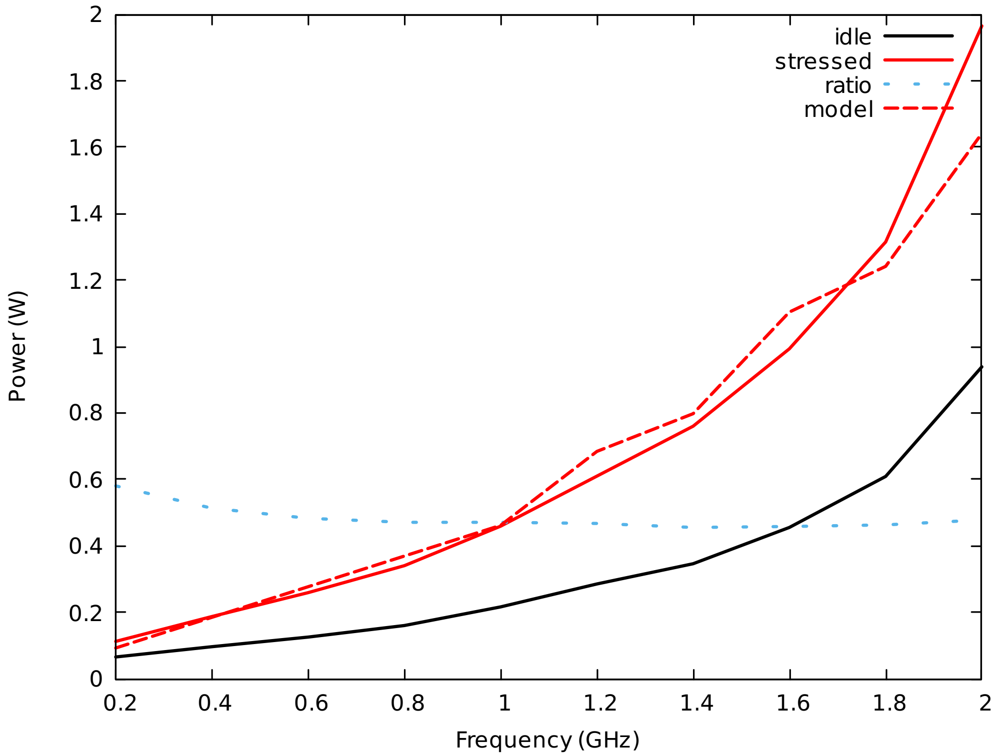


Figure 2.10: Instant power vs frequency on the Odroid XU3

In Section 1.4.3 we described the typical physical models for the DVFS and Power Gating energy/performance trade-offs mechanisms. In this section we show how we encode those models in a computer algebra system. We declare two additional variables of the problem: “makespan” and “energy”. The “makespan” variable is simply defined with the formula of Equation 1.5. Defining the “energy” variable requires more efforts which we detail thereafter.

A resource can have multiple frequency domains (or Operating Points, OPP). For each operating point and resource we define the “uptime” and “downtime” attributes. They represent the amount of time an OPP or a resource is active, and are used to estimate energy consumption. When a resource is not used at all it can be turned off. Let R be the set of resources, in order to model this energy bonus we use an if-then-else construct (Algorithm 3). The z3 solver provides such construct, which could have been replaced with a boolean variable.

We do not model operating points for other devices such as interconnects and memories since they

Table 2.1: Measured voltages and instant power values for A7 and A15 cores on the Odroid platform

	frequency	200	400	600	800	1000	1200	1400	1600	1800	2000
big	Voltage	0.91	0.914	0.914	0.926	0.946	1.026	1.054	1.116	1.193	1.319
	P_{idle}(W)	0.071	0.100	0.130	0.164	0.221	0.291	0.352	0.462	0.616	0.940
	P_{active}(W)	0.132	0.204	0.270	0.353	0.472	0.623	0.775	1.008	1.327	1.975
LITTLE	Voltage	0.925	0.925	0.963	1.038	1.11	1.185	1.285	1.285		
	P_{idle}(W)	0.016	0.022	0.033	0.048	0.087	0.121	0.180	0.180		
	P_{active}(W)	0.0277	0.040	0.058	0.086	0.148	0.207	0.281	0.284		

have strong practical limitations (see Section 3.5.3.2). We obtain the values of instant powers for each operating point, and for idle resources using either the models described in Sections 1.4.3.1 and 1.4.3.2 or the experimental instant power levels (such as Figure 2.10).

2.2.3.3 Layout and temperature issues

Temperature fluctuations are non linear and we find as expected that the temperature variations can be modeled with an RC circuit (see Section 1.4.4). The conduction phenomenon and the heat due to power dissipation are in the same order of magnitude for two resources integrated in the same chip. For instance using the Odroid XU-3 board and Exynos chip, it is easy to show that generating heat in an area of the chip causes a temperature elevation in other areas of the chip. Figure 3.10 shows the block diagram of the Exynos Chip. There is no available layout information, but we know the GPU and all cores are integrated in the Exynos5422. The chip has five temperature sensors, one on each A15 core and one on the GPU. In the experiment of Figure 2.11 we load two out of four A15 Cores using a stress utility (the two other cores are idle) and we sample the temperature for each A15 core and the GPU (also idle). The temperature raises as expected for the two active A15 cores, and reaches the same value. The temperature of the two idle A15 cores also increases significantly (about $15°C$), as well as the GPU temperature even though both are idle.

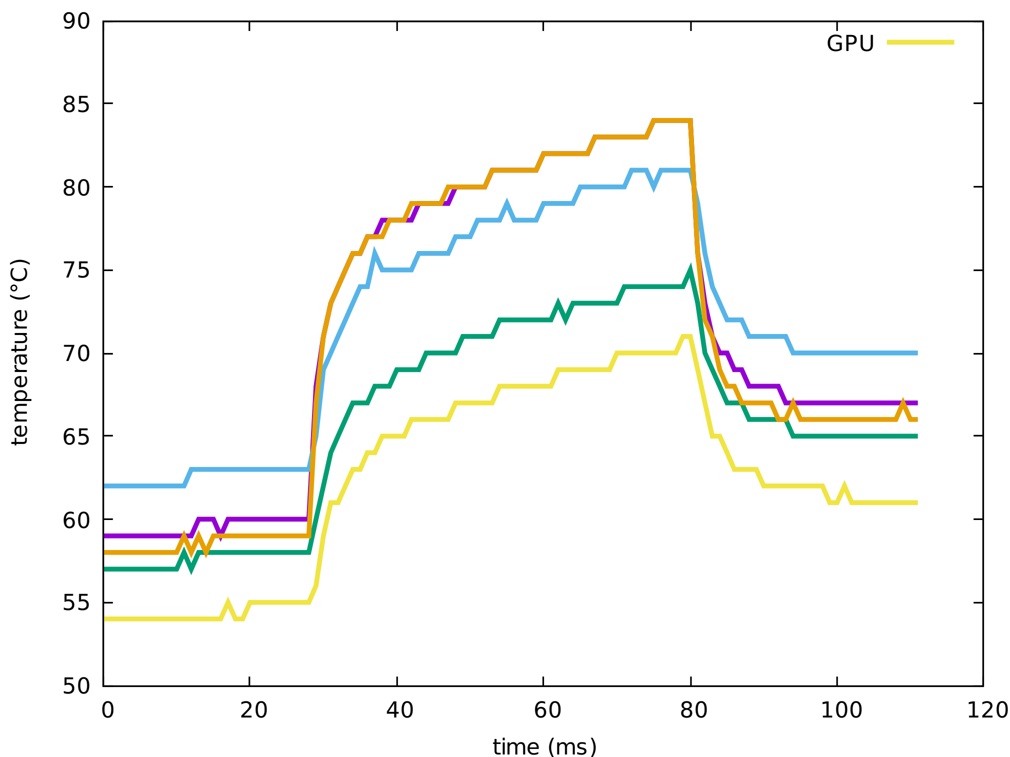


Figure 2.11: Temperature elevation of the integrated GPU and four A15 cores in the Exynos chip when only two A15 cores are active

Temperature variations are slow and are the consequence of very complex phenomena. We do not consider temperature in the SymSched tool. The naive approach described in 1.4.4 is still applicable. If we consider a setup where the resources are on separate chips, conduction is weak compared to heat generation, and piece-wise linear models could be used. We did not experiment with such computer

architecture organization and thus we did not investigate such models.

2.2.4 Architecture transformation to a solver input

The architecture adds new constraints on the already existing application constraints. We describe here how to transform them to a set of mathematical expressions that will be aggregated with the constraints obtained in section 2.1.3.

A first part of architecture constraints consist in transforming the task number of cycles and the message sizes to durations. We compute the duration of each combination of task, resource and operating frequency and we store it into a table. We do the same for communication resources and messages. We investigated mainly the Bus and Crossbar interconnects. When an interconnect is used, we add the constraints defines by Equations 2.9 and 2.10.

Algorithm 2 Architecture provision constraints

Require: $\mathcal{T}$ set of tasks
Require: M set of messages
Require: *costs* table of task costs
  for $t \in \mathcal{T}$ **do**
    $t.duration = costs[t][t.map][t.opp]$
  end for
  for $m \in M$ **do**
    $m.duration = \frac{m.size}{m.map.throughput}$
  end for
  for $(u, v) \in \binom{\mathcal{T}}{2}$ **do**
    $u.map == v.map \implies disjuncitve(u, v)$
  end for

Energy modeling requires to introduce intermediate variables. We use *OPP.uptime* as the cumulated time where this operating point is in use, and *Resource.uptime* as the cumulated time using this computing resource. Then we can calculate the downtime for each resource, which gives us the idle and active energy consumption for each computing resource.

In order to model power gating, we add the constraint that when uptime is zero, then the resource consumes no energy (instead of idling) since we consider this resource is shutdown.

2.3 SymSched Adequation model

The adequation part of the SymSched meta model introduces “Cost” and “Mapping” classes. In both cases these provide links between the application and architecture models. Costs provide problem inputs in the form of possible allocations depending on resources and operating points. Mapping represent the result in allocation and scheduling as computed by a solver which SymSched outputs as text. This output can be visualized in the shape of a Gantt diagram. After we introduce the cost notions and the relevant generated constraints (Section 2.3.1) and Mapping models and representation (Section 2.3.3) we turn back to a number of specific issues involved with the expressiveness of such model.

Algorithm 3 Energy related variables and constraints

Require: $\mathcal{T}$ set of tasks
Require: O set of OPP
Require: R set of Resources
  for $o \in O$ **do**
    $o.uptime = \sum_{t \in \mathcal{T}}(t.duration \mid t.map == r, t.opp == o)$
    $o.energy = o_p * o_{uptime}$
  end for
  for $r \in R$ **do**
    $r_{uptime} = \sum_{o \in O}(o_{uptime})$
    $r_{downtime} = makespan - r_{uptime}$
    if $r_{uptime} == 0$ **then**
      $r_{energy} = 0$
    else
      $r_{energy} = \sum_{o \in O}(o_{energy}) + r_{idlepower} \times r_{downtime}$
    end if
  end for

2.3.1 Adequation costs (solver input)

We consider that each task has a cost which depends on allocation decisions. Since we need to model different performance states for the resources, and heterogeneous resources we cannot model task costs with a quantity in physical time. Instead a cost is an amount of cycles. This amount of cycles depends on the microarchitecture of the executing resource, thus we use an attribute to distinguish homogeneous and heterogeneous resources. For instance Cortex-A7 and Cortex-A15 ARM cores are heterogeneous. The physical time cost corresponding to a task thus depends on the microarchitecture and the operating frequency. For each combination of task, frequency and microarchitecture we measure this cost and we store it in a table. Let R be the set of resources and O the set of operating points. For each task we add the constraint of Equation 2.11 which consists in two boolean decision variables reified from the two equality expressions. This equation translates the selection of a value in the "cost" array in Algorithm 2.

$$t.duration = \sum_{r \in R} \sum_{o \in O} (\underbrace{(t.opp == o) \times (t.map == r)}_{\text{Boolean decision variables}} \times cost[t][r][o]) \tag{2.11}$$

When the operating frequency can be finely tuned it is interesting instead to encode the division instead of calculating the cost for each frequency, but there might be technical limitations since solvers are not very efficient at manipulating rational expressions.

In the case of complex communication system consisting in multiple routes with different throughput the same can be achieved for message passing provided we know message sizes and route throughputs, according to Equation 2.12.

$$m.duration = \begin{cases} 0 \text{ if } m.sender.map == m.receiver.map \\ else: \sum_{r \in Routes} (m.route == r) \times \frac{m.size}{r.throughput} \end{cases} \tag{2.12}$$

2.3.2 Orthogonality and composability of the models

In the previous sections we explained why and how we use orthogonal models of application and architecture. Adequation modeling consists in figuring out how those models are coordinated. In this section we look for the limits of the approach and we give the hypothesis that must hold to ensure that the cost functions are valid.

"WCET" (Worst Case Execution Time) is a difficult topic that we don't fully tackle here. In our adequation approach we address the case where the cost of agents do not have much variability which we carefully check for instance in section 3.5.1. We consider the case of an embedded system, where the resources are usually not shared to multiple users, and the user or designer is able to control which process the system runs and when. We consider only "data-independent" kernels, which means that they run the same sequence of instructions regardless its input data since no branch in the program depends on this input data. We consider that even when they run on a shared memory architecture if such kernel can fit all their input in the cache then we can predict the amount of obligatory (see Section 1.4.2.2) cache misses.

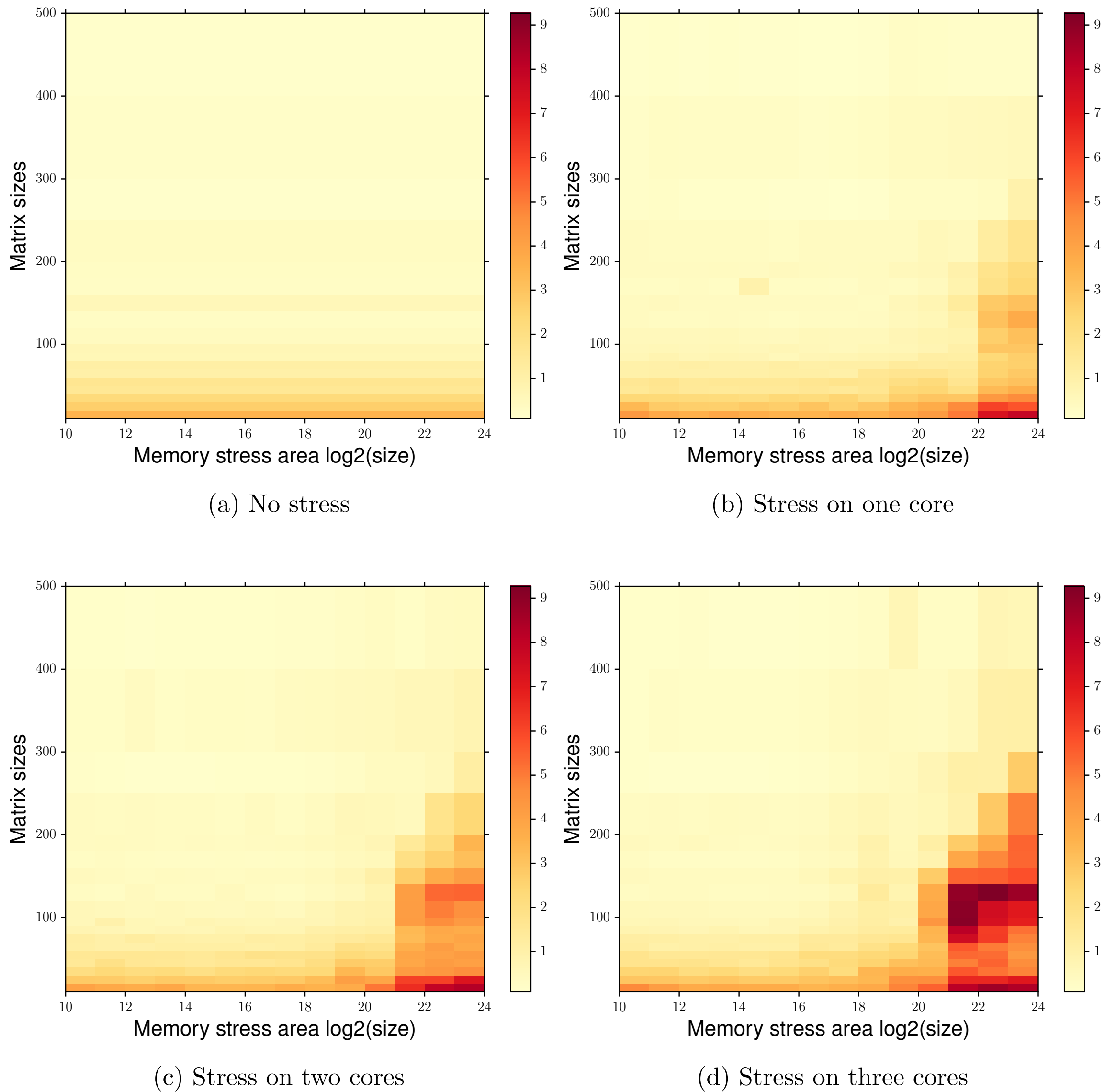
The most flagrant example of "data-dependent" kernels are the sparse matrix operation algorithms, where their dense counterpart are however "data-independent". For instance we expect that a matrix multiplication routine running on the same microarchitecture requires the same amount of instructions for a fixed matrix sizes whatever values the matrix constrains. This hypothesis would probably not hold in the case of a sparse matrix multiplication algorithm, because then the algorithm would probably behave differently depending on the sparsity of the matrices. In that case we could also over-approximate the cost of the matrix multiplication kernel, which is the purpose of the WCET metric.

An important part of the characterization step consists in determining the Average Case Estimation Time (ACET), and verifying that its standard deviation is small. It is a mandatory step in order to dimension the agents such that the implementation of the resulting schedule behaves as expected. We make the hypothesis that on a shared memory machine, two kernels that access a "large" memory area interfere more (regarding caches) than two kernels that access a "small" memory area. We give evidence that this hypothesis holds on a four cores cluster of the Odroid architecture (described in Section 3.5.2) . Core#0 runs a matrix multiplication kernel (double precision) with different matrix sizes, and we sample the number of cycles that it needs to complete. Then we compute the mean, standard deviation and coefficient of variation when Cores#1..3 are stressed with a memory intensive kernel. Figures 2.12 allows to identify that we can expect a low coefficient of variation if the memory area is no larger than about 2^{18} Bytes for this architecture. This experiment suggests that obligatory and capacity cache misses can be estimated whereas conflicting cache blocks are much harder to predict.

Embedded MPSoC often run an operating system even when there is not much user interaction (mostly for runtime scheduling purposes). Most often the system is not shared among multiple users (unlike High Performance Computing clusters). Most simple "real-time" or "embedded" operating systems consist in a set of files that a compiled with the application. General purpose operating systems on contrary offer a broad list of services that have to be carefully prioritized or shut down in order to obtain predictable runs.

Techniques that are used for other purposes such as security can be counterproductive regarding the predictability, and have to be disabled. For instance recent work [75] on the Linux ASLR (Address Space Layout Randomization, which is a counter measure for buffer overflow attacks) reports a 10% average performance penalty for CPU intensive SPEC CPU2006 benchmarks (unfortunately the study does not include variability measurements).

Appendix A details the tools and library calls that we used to obtain the results of Figure 2.12.



(a) No stress

(b) Stress on one core

(c) Stress on two cores

(d) Stress on three cores

Figure 2.12: Coefficient of variation vs memory stress

2.3.3 Mapping representation (solver output)

We used application equations (2.2, 2.3, 2.4, 2.5, 2.6), architecture equations (2.8, 2.9) and cost definitions of equation 2.11 to obtain a mapping. SymSched serializes the objects from the model defined in Figure 2.1 (Example thereafter) such that it is easy to interface it with another tool for instance for plotting purposes or code generation purposes. We do use a modified version of an existing python package [3] which transform this JSON into a Gnuplot file that we can use to generate several output formats.

```
1  {
2      "Resources": {
3          "140446157431296": {
```

[3]The Shed: `http://devel.se.wtb.tue.nl/trac/shed`

```
            "cumulative": 1,
            "label": "CPU0"
        },
        "140446157432016": {
            "cumulative": 1,
            "label": "CPU1"
        },
        "140446157847568": {
            "cumulative": 1,
            "label": "Bus"
        }
    },
    "Tasks": {
        "T140446157475920": {
            "color": "0x6432ff",
            "label": "Sobel_H",
            "map": 140446157431296,
            "opp": "2 W",
            "precedences": [
                "T140446157845712"
            ],
            "start": "15040.0000000000 us",
            "stop": "27000.0000000000 us"
        },
        "T140446157476432": {
            "color": "0xff3264",
            "label": "Histo_H",
            "map": 140446157432016,
            "opp": "5 W",
            "precedences": [],
            "start": "18960.0000000000 us",
            "stop": "27000.0000000000 us"
        },
        ...
}
```


2.3.4 Comments on expressiveness

So far we have set up our SymSched framework, with models that all try to remain simple and at the same time preserve the essential features needed for the subsequent mapping. In particular we paid special attention to the modeling of *alternative* allocations (with their respective costs), while the impact of these alternatives on power consumption was also a modeling concern.

The main objective, being limitations to build a constraint system that had the best chances to scale down to solver's capacity in complexity, led us to a number of choices and reflections upon various phenomena and how they could be handled. The task division for preemptability or the data-parallel constructs were examples of this. We now comment on the various steps that led us to the current forms, and on some restriction/extension variants that were considered (sometimes postponed, sometimes discarded, or partially included in our framework).

2.3.4.1 Limitations of the SDF model

We can identify two limitations of the SDF model for practical adequation.

The first one is than the nature and the data size of tokens is undisclosed which does not allow to associate a size to tokens, and thus to messages. Figure 2.4 gives the typical examples: increasing X

consists in splitting the image in smaller cells, thus the size of the output token from the Image Read agent is the size of the image divided by X. The tokens going from Sobel to Histogram agents are also probably image cells, but the nature of the Detection correction inputs and outputs are unspecified.

The second limitation is that the transformation to an APG can yield an exponential number of tasks and messages. According to previous work (StreamIt [92]) the input/output rates of the places are not very often unbalanced. Yet StreamIt graphs make an extensive use of split merge patterns. It is thus a natural idea to think about a better encoding for this specific graph structure.

Task parallelism and data parallelism are very similar notions, yet data parallelism suggests there is an amount of enumerable objects that must be processed. Instead of modeling one task for each of those objects and transform the whole set of tasks to constraints we investigate a different encoding with parallel agents.

2.3.4.2 Data parallelism

As explained in section 2.1.1.4 SymSched defines a model of data parallelism which consists in splitting some agents into chunks (which we consider as tasks). Given a set of agents, if one of them is data-parallel and can be split into tasks, the minimum number of chunks which allows to obtain the minimum makespan is not known. Distributing one task on one resource looks like a natural option since using more tasks than resources will result in sequenced tasks on some of the resources, which is then equivalent to a preempting this generated task on this resource. Therefore using as many tasks than resources can be interpreted as an "optimal" decision if we consider the non preemptive scheduling problem. We pass it as an optional argument, which by default is equal to the number of resources. The duration of each chunk is then another concern, which can be parameterized in our approach as described in the parallel agent model. Figures 2.13 and 2.14 give example schedules of a static and a dynamic parallel agent. In those examples we consider that all the resources are homogeneous and have only one operating point. In those examples distributing one chunk to each resource and allowing variable durations allow to evenly distribute the workload.

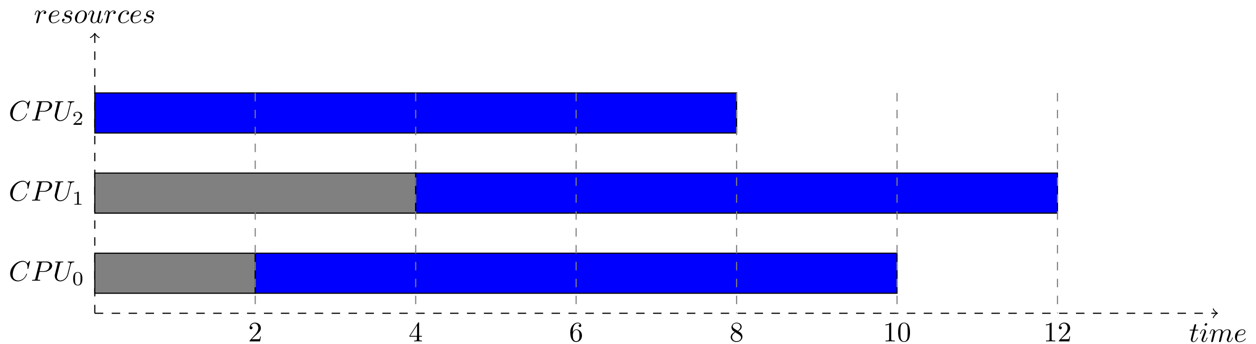


Figure 2.13: Mapping of a parallel agent (blue, cost=24) combined with two regular tasks (gray) and no messages with static task costs

From a practical point of view, this is very similar to the "static" or "dynamic" OpenMP scheduling policies, apart that since OpenMP annotations do not define task costs (or code sections costs), "dynamic" OpenMP scheduling does the balancing at runtime whereas our approach allows to compute a balanced schedule at compile time. We describe the model assuming homogeneous resources. In the case of heterogeneous resources, the generated constraints (Equation 2.4) needs to be modified in order to balance the workload. Evenly distributing the costs in an application made of multiple agent

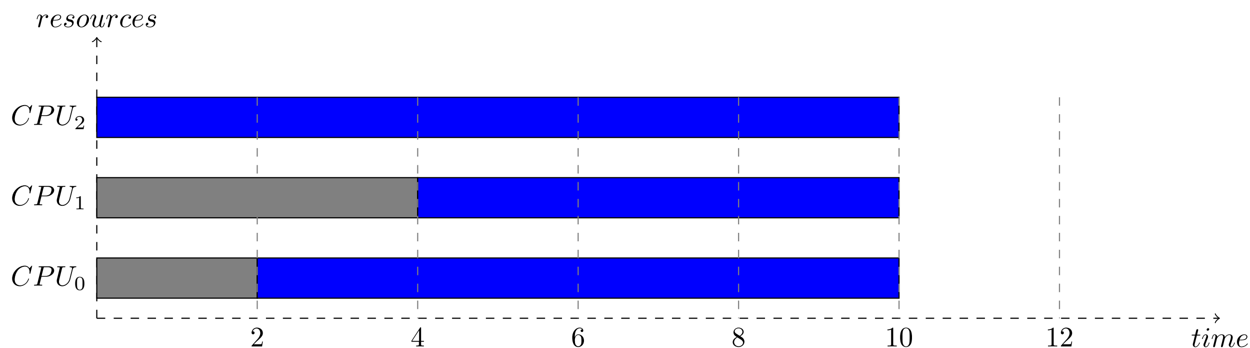


Figure 2.14: Mapping of a parallel agent (cost=24) combined with two regular tasks (and no messages) with variable task costs

types requires that the durations of its instances are a variable of the problem. To our knowledge no such approach exists in the literature for offline scheduling purposes.

Parallel agents can receive or send messages. The input and output precedences are transformed to an input and output precedence for each instance and the messages sizes are split evenly according to the cost of each instance. In many cases this model is not sufficient since some data may be completely or partially duplicated and sent to multiple workers. This requires data dependency analysis on an implementation (or pseudo code) of the application, which we do not model.

2.3.4.3 Pre-calculated preemption

In practice a preemptive task can be migrated to multiple resources. A migration requires to send data (the internal state of the task), which means there is a cost for each migration. Since we can obtain a makespan of the scheduling problem in polynomial time by using an heuristic (such as ASAP), we can bound the number of potential preemptions and migrations. This bound is still very large, but it allows to model the preemptive task as a pipeline made of a finite number of successive tasks. However unless the duration of the task is in the order of the migration cost, this (yet finite) model will not scale at least using exhaustive search techniques. As well as regarding parallel tasks, the smallest number of instances which maintains optimality is still an open question. This problem is sometimes referred as pre-calculated preemption [18].

2.3.4.4 Periodicity

Figures 2.15 and 2.16 give two example schedules of periodic agents described in Section 2.1.2.1 on a two processor architecture. Figure 2.15 gives an example where the period equals the deadline, and Figure 2.16 gives an example with $period > deadline$. In those examples the agent instance is transformed to 4 task instances. An additional attribute of periodic agents allows to specify if the deadline is relative to each task appearance, or if it is specified as an absolute value. The latter case is sometimes specified for instance in signal processing when a given amount of data arrives periodically and can be buffered (and processed later).

Bounded time periodic task scheduling problems Because we need to bound the scheduling problem to apply the further transformations (see Section 2.1.3), the periodic tasks are not transformed to an infinite number of tasks. Instead we find a macro-period and expand each periodic task according

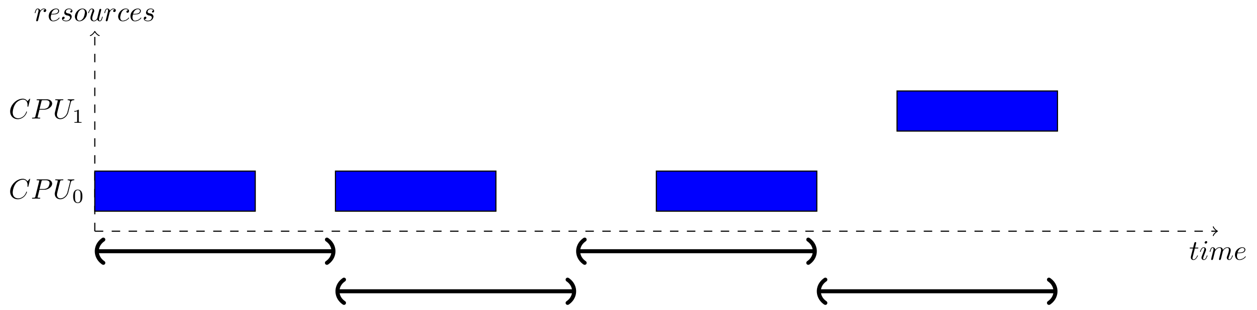


Figure 2.15: Admissible schedules of a periodic task with *period* = *deadline*

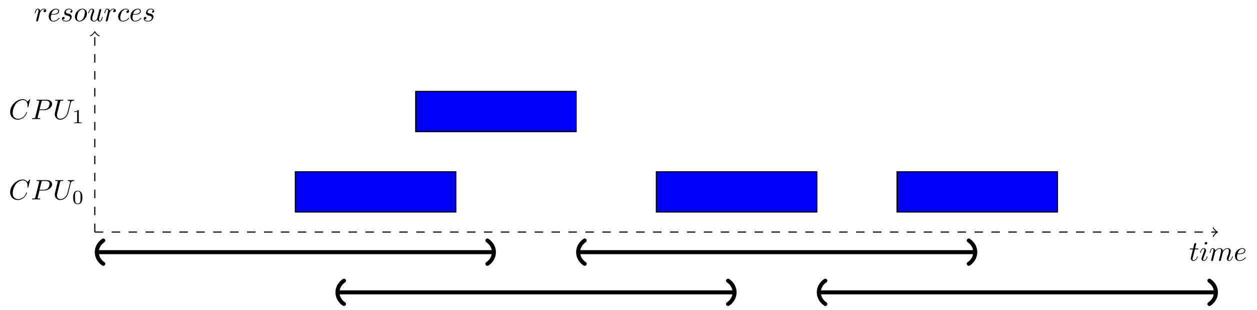


Figure 2.16: Admissible schedules of a periodic task with *period* ≠ *deadline*

to it. A satisfiable instance of the scheduling problem on this macro-period is a valid steady state for the system. Let T be the set of periodic tasks in the problem. The smallest macro-period is the Least Common Divisor of the periods and we expand each periodic task to the proper number of tasks:

$$\begin{aligned} M_c &= LCM(task.period \mid task \in T) \\ task_{\#instances} &= \frac{M_c}{task.period} \in \mathbb{N} \end{aligned} \tag{2.13}$$

The approach will fail with very uneven periods, which in practice is a corner case example that can be better described using other models such as structured task graphs (Section 1.2.2.2). If the problem includes other tasks, its bound must be adjusted regarding their deadlines to the correct number of periods. For instance if another (non periodic) task has a deadline greater than one macrocycle, the bound is set to the minimum number of macrocycles which is larger than this deadline.

Preempting periodic agents In periodic task problems we can choose a number of preemptions that makes sense. The typical example is a periodic task scheduling problem where a frequent lightweight task has to execute often while a large time consuming task has to complete. Then it makes sense to preempt the time consuming task based on the ratio of the largest and smallest periods of the tasks which recalls a TDMA scheduling approach. This approach is sometimes referred as "pre-computed preemption", it illustrated in Figure 2.17 where the time consuming task i with large period is preempted twice such that a task j with a smaller period can execute. Since $\lfloor \frac{p_i}{p_j} \rfloor = 2$ preempting task i by two is sufficient to find satisfiable schedules.

The approach allows to find satisfiable schedules for periodic task scheduling problems by considering only a limited number of preemptions in the case non-preemptive scheduling fails. Example of

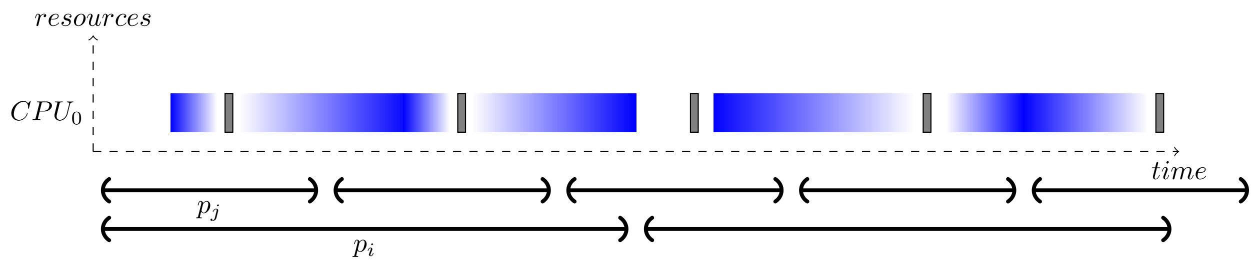


Figure 2.17: Example pre-computed preemption

Figure 2.17 cannot be scheduled on a single processing resource without preemption.

Sporadicity A sporadic task is an aperiodic task which generates tasks with a minimum and/or maximum inter-arrival time. Sporadic tasks typically model user, or environment interaction. A minimum inter-arrival time makes sense for instance in user interaction problems: humans need a minimum amount of time to react to the device outputs, and they expect that the device reacts as fast as possible. Scheduling such tasks (either on-line or off-line) requires a non-zero minimum inter-arrival time. When this condition holds then on-line schedulers can be evaluated against such problems, but because the arrival is known at runtime, off-line schedulers cannot. The state of the art for modeling such tasks consists in considering the worst case scenario, where the sporadic task is modeled with a periodic task whose period equals to the minimum inter-arrival time [49] which can be achieved using the previously described models of period tasks (Section 2.1.2.1).

2.3.4.5 Combining agent models and requirements

We do not investigate all the combinations allowed by this meta model. We prevent using some of them that does not make sense, such as adding dependencies to periodic tasks.

We do not define precedences or messages to and from periodic tasks since it can become tricky to model a meaningful system when two periodic tasks with different (and non multiple) periods are ordered. When on contrary the periods are equal or multiple of each other, it is not necessary to model this feature since the application then falls into the category of streaming applications for which a model of rates (such as synchronous dataflow) described in Section 2.1.1.3 is the most appropriate.

Other combinations such as parallel and periodic agents on contrary make sense. Using preemptable agents in an SDF graph, with a pre-computed amount of preemptions is also allowed although choosing a meaningful amount of preemption for each agent can be difficult.

Chapter 3

Using solvers for AAA: lessons learned

In the previous chapter we described how the various modeling aspects associated with the (low-power) adequation problem can be translated into a set of formulas, including linear inequalities and propositional logic constraints equivalent to boolean decision variables. In order to use an SMT solver we must first encode those equations in a Computer Algebra System (such as Maple, Mathematica, Sympy, ...) and then invoke the back-end SMT solver. In particular the time related variables could be either integers, rationals or reals. The solver then attempts to recognize for the users which theory may be applied for resolution, given the syntactic types of formulas involved (Example solver log in Section A.6) Therefore the actual shape of the constraint set resulting from translation does matter a lot. We experimented mainly with two environments: Z3 and the Minizinc G12 solver.

Before we focus in the rest of the chapter on the central topic of actual resolution, we mention a few expansion steps that are to be performed on our set of constraints, to express them as recognizable celebrated theories in SMT libraries. For instance, the careful reader would have already noticed that some of our synthesized formulas from the previous chapters were defining rational values, by using ratios in between quantities. Restoring true integer-valued formulas suggest identifying common divisors. We comment on some of these issues in section 3.1.

In section 3.2 we test the power of actual SMT solvers on simple synthetic examples expressing the kind of constraints as generated by the AAA adequation problem. At this stage the specifications do not attempt to be realistic, but only to exercise the various elementary features identified as important modeling constructs in chapter 1, especially on the applicative side. The purpose is mostly to challenge tractability of the resolution techniques. Again, we consider a few issues made to improve the scalability and applicability of SMT solvers, when they be expressed at this level.

Realistic applications (and one actual implementation) are introduced in Sections 3.4 and 3.5, especially considering the modeling of architectural platforms: low-power allocation alternatives between big and LITTLE cores are introduced. On the application side we consider 3 task graphs: one for Cholesky matrix decomposition algorithm, one for a platooning application, and the FFT convolution with its radix stages. Of course we still have to keep the set of constraint formulas simple to make the problem tractable, but on some meaningful use cases we can also monitor (and comment on) the quality of the result, where some form of optimality for a mapping (physical allocation + temporal scheduling) solution is intented, across a number of simplifications and transformations.

3.1 Practical aspects of the modeling

Computer Algebra Systems such as Sympy or Maple allow reasoning on those mathematical expressions, simplify them and solve some of them. Sometimes the problem is simple enough (tractable)

and we can find a closed form expression. Most of our problems are on contrary untractable, and this simplification step consists in finding a canonical form of the problem that suits a specific theory. Since the solvers often target specific problems the computer algebra systems often use them as a back-end. In this section we show how to express the scheduling problems in a computer algebra system.

3.1.1 Time discretization

It is generally admitted that the timestep in a scheduling problem is part of the specification, and is not something that is up to the static scheduling algorithm. Naturally, The largest dt results in the smallest state space. It's thus worth trying to find a large timestep. In SymSched the timestep can be set by the user but because of the nature of the problem, we also show how we find a coarse timestep with interesting properties.

Let a single task t of t_c cycles and a single resource r with three operating points op_0, op_1, op_2. The resource can run this at one of the admitted frequencies f_0, f_1, f_2 and we define $df = gcd(f_0, f_1, f_2)$. The frequencies corresponding to the operating points are multiplied from a base frequency. For instance in our Odroid setup (Section 3.5.2) available frequencies range from 200MHz to 1.6GHz by steps of 100MHz. Thus the task durations are defined according to Equation 3.1 (there is only three possible task duration in this problem) and $\frac{t_c}{df} \in \mathbb{Q}$ is the the GCD of the task durations. The GCD of the task durations is thus an interesting timestep since it introduces no time discretization error in the duration of the tasks.

$$\begin{aligned} f_i &= df \cdot k_i \mid k_i \in \mathbb{N} \\ t_d(op_i) &= \frac{t_c}{f_i} = \frac{t_c}{df \cdot k_i} \end{aligned} \tag{3.1}$$

Figure 3.1 gives an example of an agent and a resource with the three frequencies: 1GHz, 1.2GHz and 1.7GHz. Assuming the agent costs 1MCy on this resource we can calculate the GCD (Equation 3.2):

$$dt = \frac{1MCy}{100MHz} \cdot GCD(\frac{10}{17}, \frac{5}{6}, 1) = \frac{1}{102} ms \tag{3.2}$$

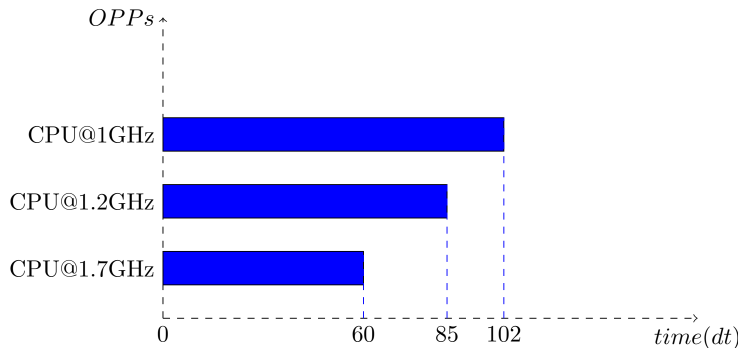


Figure 3.1: Example time discretization on frequency scaling

The same methodology applies when multiple tasks are involved, or multiple homogeneous or heterogeneous resources with different operating points. Since in the SymSched model the resource frequencies are (by construction) multiple of each others the time discretization is often coarse. When the cost of two tasks are different by orders of magnitude the smallest tasks are most often ignored.

Furthermore since those costs are approximated the problem specification often rounds them to the same order of magnitude. This means that the task costs are also likely multiples of a large number of cycles which can be factored. When another duration has to be introduced in the model such as migration costs, in our approach it is not considered for the calculation of dt (and rounded to the nearest value).

Eliminating those common factors suggests that the problem would be better encoded as a set of rational variables and a set of intervals. We considered specifying the problem using a rational variables, but we could not find a solver able to reason on rational variables. The standard SMT-libv2 language does not define it [21]. The Z3 CAS defines a rational type that can be used for numeric constants only (and not for variables). It is possible but cumbersome to encode a variable using two integers. Some approaches use real variables, which allows to avoid this pre-processing but also looses the intrinsic rational nature of the problem. To our knowledge there is no extensive quantitative experiment on which encoding (variable types and SMT solver tactics) is the most effective for task scheduling problems. We experimented with the default Z3 SMT solver tactic and both types (integers and reals). When it is not specified, z3 chooses the tactic based on the input variables and expressions (which may differ depending on the version of z3). On instances of independent task problems (such as described in section 3.2.1) we found that using real variables generates one order of magnitude more conflicts (and the search lasts about twice as much time) as when integer variables are used.

3.1.2 Modeling cyber physical systems

Adequation problems combine different physical dimensions such as time, energy and also dimensionless quantities (number of cycles, number of bytes, ...). Moreover, designing such system requires to express quantities with various magnitudes, for instance microseconds to seconds. Human interaction with such tool is easier when the interface allows to use a system of units. Many computer algebra systems come with this feature. The Z3 interface does not since it focuses on expressing problems for the solver which manipulates only unit-less quantities but it is easy to interface it with a package that implements physical quantities [1].

Such tool can also exhibit design errors in the models, such as comparing instant power and energy or missing some aspects of the models. A typical example is when building the instant power models according to Equations 1.7:

```python
from pint import *
ur = UnitRegistry()
V, f, i_leak, A = ur("1V"), ur("1GHz"), ur("1mA"), 1
P = V**2 * f * A + V * i_leak # raises a DimensionalityError
c_r = ur("10nF")
P = c_r * V**2 * f * A + V * i_leak
```

Since the solver manipulates only dimensionless variables we have to transform undetermined and determined variables to integers. We do it using the common divisor obtained in Section 3.1.1. The same can be achieved for instant power levels, since the resources uses only a finite number of instant powers values.

3.2 Scalability

In this section we study the practical complexity of constraint programming on synthetic instances of the adequation problem. Section 3.2.1 gives an extensive evaluation regarding independent tasks. We

[1] We used the python Pint library: `https://pint.readthedocs.io/`

did no extensive study of the scalability regarding the models of parallel and preemptable agents, but sections 3.2.2 and 3.2.3 give some insights. Section 3.2.4 highlights the differences between satisfiability and optimization using SMT solvers and motivates the use of optimizers for such problems.

3.2.1 Independent tasks

A natural experiment consists in studying how many independent tasks the solver is able to handle, and how many resources. This also gives an idea of the scalability of Parallel agents regarding the number of generated tasks. In the case of homogeneous resources, the problem is equivalent to a bin-packing problem with a fixed number of bins.

We define a tightness criteria: in the case of homogeneous resources it is equal to the makespan of the sequential schedule divided by the number of resources. In the case of heterogeneous resources it is the average of sequential schedules for each resource. We also performed experiments on randomly generated direct acyclic graphs of tasks with some results in [40] but interpreting the results is complex since most applications are in fact made of an unstructured graph of structured sub-graphs such as Preemptable and Parallel agents. We thus focus on the problem of independent tasks. We run a large number of experiments and we highlight some interesting correlations in Figures 3.2 and 3.3.

For each experiment:

- Solver search timeout is set to 60s
- The number of tasks is uniformly distributed in 1..100
- The number of resources is uniformly distributed in 1..32
- We consider no interconnect model
- Each problem is either homogeneous or heterogeneous, with randomly chosen task costs (in 2..20MCy, with one operating point for each resource at 1GHz)

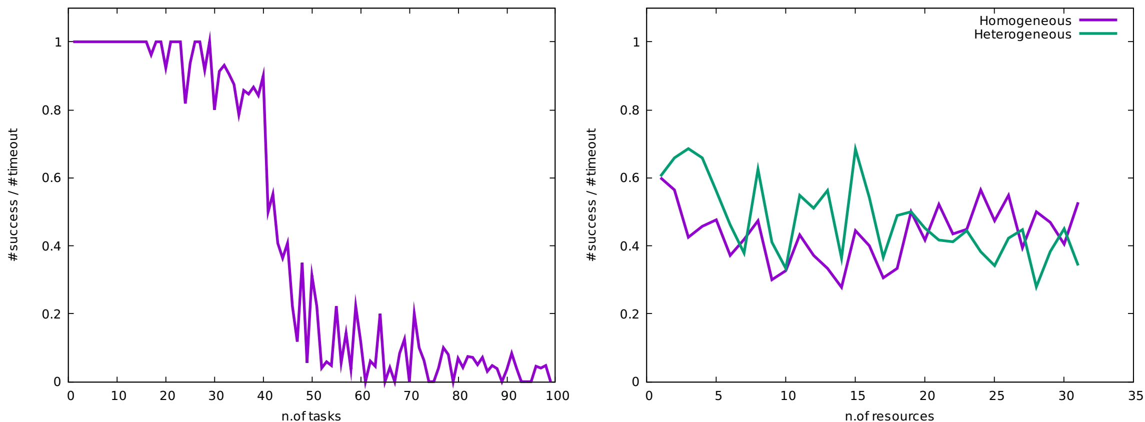


Figure 3.2: Practical complexity: success rate vs number of tasks (left) and nature of resources (right)

In our previous experiments [40] we found that the tightness parameter is essential regarding the ability of the solver to return a satisfiable instance (or to determine that there is no satisfiable solution). Those additional experiments show that the solver is indeed efficient at finding a satisfiable instance

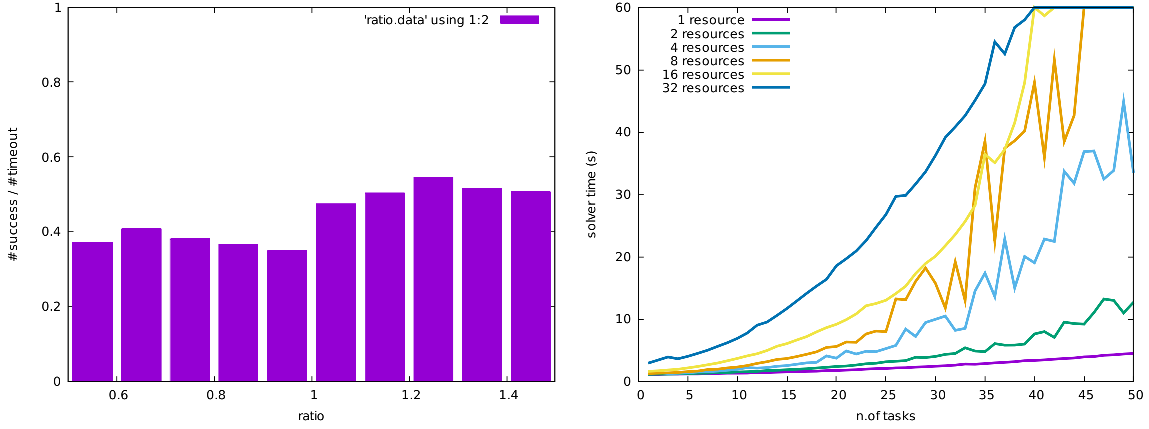


Figure 3.3: Practical complexity: success rate vs problem tightness (left) and solver time vs number of tasks (right)

when the time budget is not very tight, but not equally efficient at deciding unsatisfiability when the time budget is too small.

Most of the practical complexity for problem instances follow the same exponential curve, but as expected it is hard to guarantee that a problem will be easy to solve in practice. Some problem instances thus have unusual high or low practical complexity. We find as expected that a problem is more likely to have a low practical complexity if it has a small number of tasks and resources. Figure 3.2 shows that for a fixed timeout (of 60s) there is however a clear correlation between the success rate and the number of tasks since most of the problems including more than 50 independent tasks will fail. As expected the solver search time is exponential with the number of tasks. Figure 3.3) gives the solver time for a different number of resources and a fixed tightness. The steepness of the exponential increases with the number of resources and the tightness of the problem.

We try to compare homogeneous and heterogeneous resources regarding solver time for fixed tightness. There is no obvious difference unless that for a small number of resources, the heterogeneous case is easier. We interpret this tendency as the effect of (no) symmetry breaking in the homogeneous case (see Section 3.3). The results thus suggest that there is a possible improvement of the encoding. The choices of solving tactics could also be improved since although the 1-resource problem is not even an NP-complete problem, the solver surprisingly fails to prove some unsatisfiable problem instances.

Laziness Naturally, a better encoding exists for makespan optimization purposes on this independent tasks problem. When a resource finishes a task, if no other task is pending then all the independent tasks have been scheduled. This translates to the constraint of Equation 3.3. This resource criteria exists in the real-time scheduling theory and is known as "non-laziness". Figure 3.4 gives an example of lazy and non-lazy schedule for the same problem instance of 20 independent tasks and 4 resources.

$$\begin{aligned} &\text{set of tasks: } \mathcal{T} \\ &\forall t \in \mathcal{T} : \begin{cases} t.start = 0 \text{ or} \\ \bigvee_{\substack{j \in \mathcal{T} \\ j \neq t}} ((t.map = j.map) \wedge (t.start = j.stop)) \end{cases} \end{aligned} \tag{3.3}$$

The problem can be further simplified: the ordering of the tasks on one resource is not relevant since they are independent. Only their cumulated duration is. Then the resulting problem is equivalent

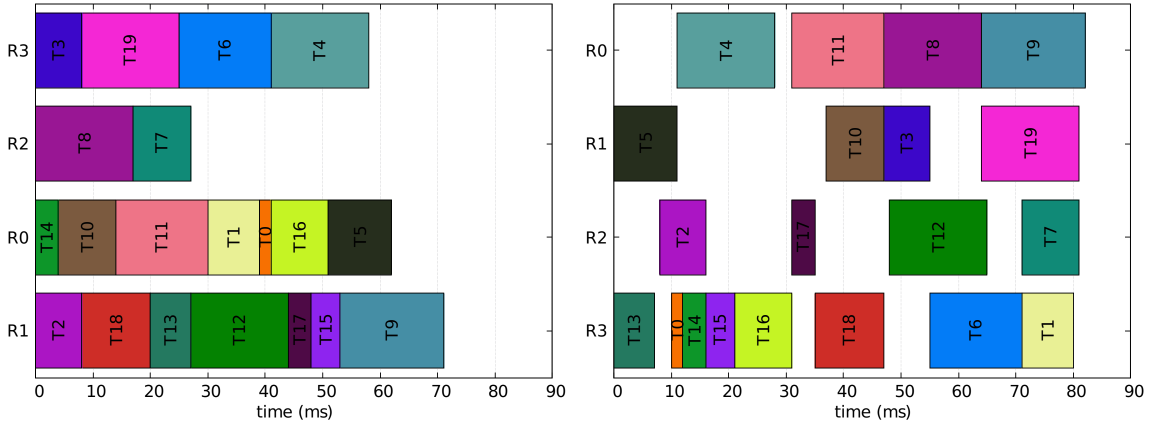


Figure 3.4: Generated non-lazy and lazy schedules for independent tasks

to a bin-packing problem with a fixed number of bins.

Those simplifications apply only to independent tasks. For instance we are not aware of a general result regarding non-laziness scheduling of direct acyclic graphs of tasks with precedences and/or communication costs. Because of the task dependencies in the application model the available parallelism during the execution of the application may be greater than the available computing power, and sometimes smaller which means even if the schedule makes maximum use of the computing resources, the resources may be sometimes idle and sometimes active. In this case we cannot use the constraint of Equation 3.3 since it does not hold for valid, and maybe optimal schedules.

3.2.2 Preemptable agents

Our previous experiments [40] report that scheduling preemptive agents is as expected more complex in practice than non preemptable tasks. In this work we use independent preemptable agents and we experiment different amount of preemption. The tasks (that result from preemptions) are able to migrate to other resources. As expected a greater number of preemptions raises practical complexity. Figure 3.5 gives the satisfiability solver time for a problem of 4 resources and 8 preemptable agents for a different number of preemptions. For instance 2 preemptions means each of the 8 agents is split into 3 tasks.

Scheduling only one preemptive task (which is of course not an NP-Hard problem) scales linearly as expected.

3.2.3 Parallel agents

Satisfiability of only one parallel agent on only one resource with limited time budget scales linearly as expected. Surprisingly, optimizing the same trivial problem scales exponentially. Further investigation shows that as expected the optimizer finds the optimal makespan very fast, and then stalls exploring equivalent sequences of tasks. Naturally, ordering those tasks (see Section 3.3) solves the issue and generally speaking, if two independent tasks are sequenced then they can be ordered.

3.2.4 Optimization and convergence speed

The experiments in section 3.2.1 show that the solver is more efficient at finding a solution when it exists than proving unsatisfiability, especially when the adequation problem has slack time (when the real-

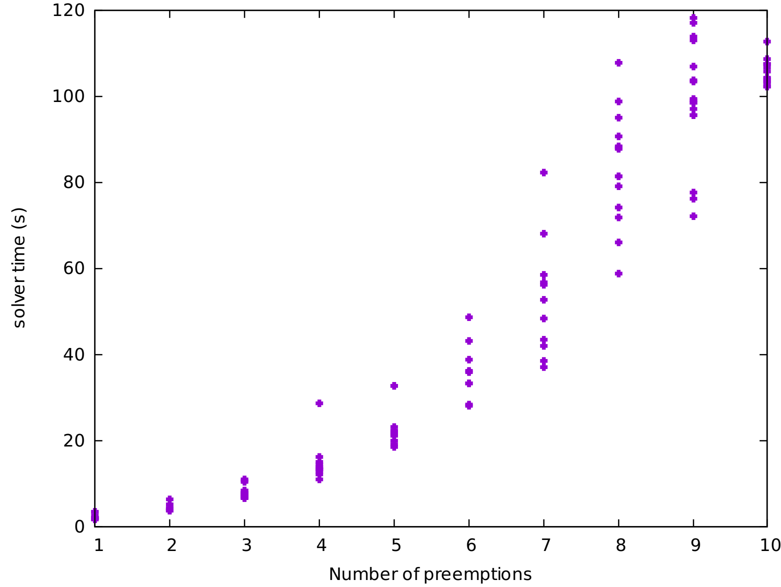


Figure 3.5: Solver time (s) vs number of preemptions for 8 pipelines, 4 resources, and tightness=1.3

time constraints are not very tight). SMT optimizers consist in successively tightening the objective function until the problem becomes unsatisfiable. We thus expect that regarding optimization (of makespan or energy), the solver quickly finds a solution which is close to the optimal, slowly converges to the optimal value and then consumes a lot of time trying to figure out that this is the optimal value. Because of the various types of adequation problems, it is hard to do an extensive study on the topic but we were able to observe this on our examples. For instance in the results presented in Table 3.7 on the optimization of Cholesky matrix factorization, we ran multiple experiments with timeouts and observed exactly this behavior (Figure 3.6).

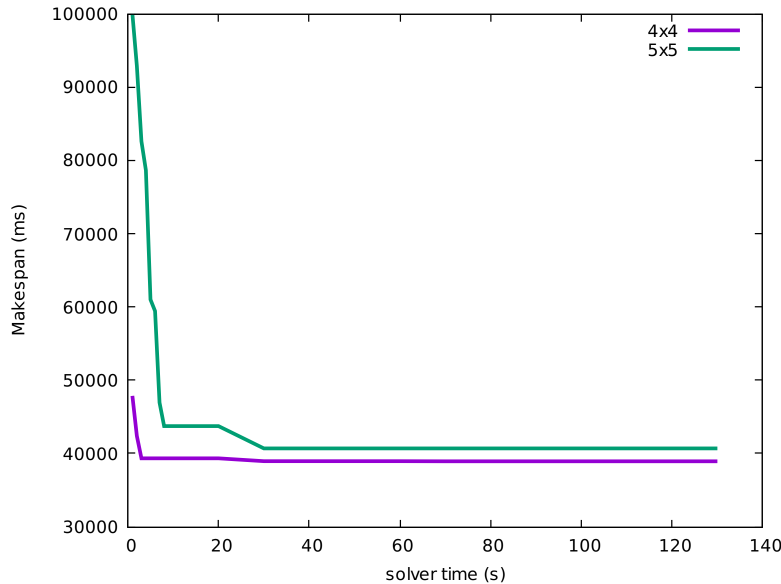


Figure 3.6: Optimizer time vs makespan in the Cholesky matrix factorization use case (4x4 and 5x5 tiles on two heterogeneous resources)

3.3 Symmetry breaking

Symmetry exists in constraint programming models whenever two elements of the problem can be swapped without modification of any other variable. Regarding scheduling problems, it appears for instance when scheduling tasks with no communication costs on homogeneous resources. In this case if a valid schedule is obtained we can swap two homogeneous resources and obtain (another) valid schedule with identical makespan (and the same start and stop values for each task). Exploring both solutions is however a waste of time. To our knowledge automatic symmetry breaking is not implemented in Z3 or in Minizinc G12 solvers. Breaking symmetry can thus be achieved either by adding new constraints to the model such that only one of the symmetric solution satisfies the constraints, or using a different modeling approach (such as using our models of parallel agents instead of instantiating many workers for data parallel agents).

In this section we try to highlight potential benefits of symmetry breaking by studying the scalability of a realistic application with and without adding symmetry breaking constraints.

3.3.1 The Platooning application

A platooning application is run by one car in order to automatically follow the car in front of it. It reads inputs from an embedded camera and controls the speed and steering of the car. This is an oriented gradient histogram computer vision application: the image is split in small areas (or cells) that are analyzed with a Sobel edge detection algorithm and evaluated with an histogram algorithm. A detection algorithm that has been trained before with a set of images and results is then able to detect vehicles and change the car speed and steering accordingly. We do not have a functional implementation of each kernel, thus we were unable to evaluate their cost. We use arbitrary costs and assume that the image processing kernels are the most intensive.

The detection algorithm also adjusts image capture parameters, which means there is a feedback loop in the model. The feedback dependency arc initially contains two tokens. Figure 3.7 gives an SDF model of this application. Since it contains a cycle we need to add initial marking to make in "live" (see Section 1.2.2.3). Each cell of the image can be processed independently and there is thus a large available parallelism. This available parallelism can be adjusted by modifying the rate parameter X. Figure 3.8 gives an example of Acyclic Precedence Graph on one period and with the parameter $X = 3$. The graph can also be unrolled on multiple periods for pipeline parallelism.

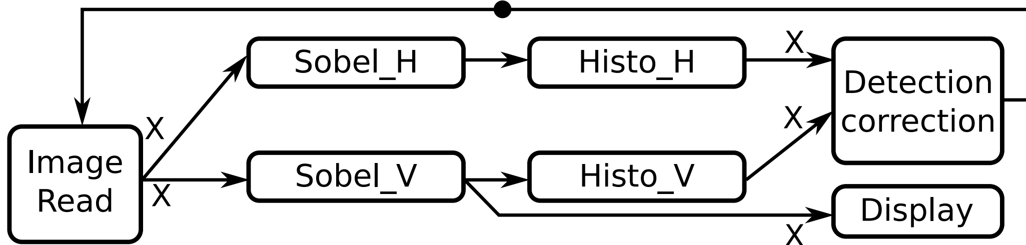


Figure 3.7: Platooning application dataflow graph with a rate parameter X

The results of Table 3.1 were obtained with the G12 solver on a Minizinc model and give the solver time and memory consumption for different values of X, considering only one period. We observe that the solver fails when we consider a graph with large breath. The other experiment consists in raising the number of periods with $X = 1$. On contrary it scales to a large number of periods (Table 3.2) but at some point exhausts the memory. We found similar results using the Z3 solver.

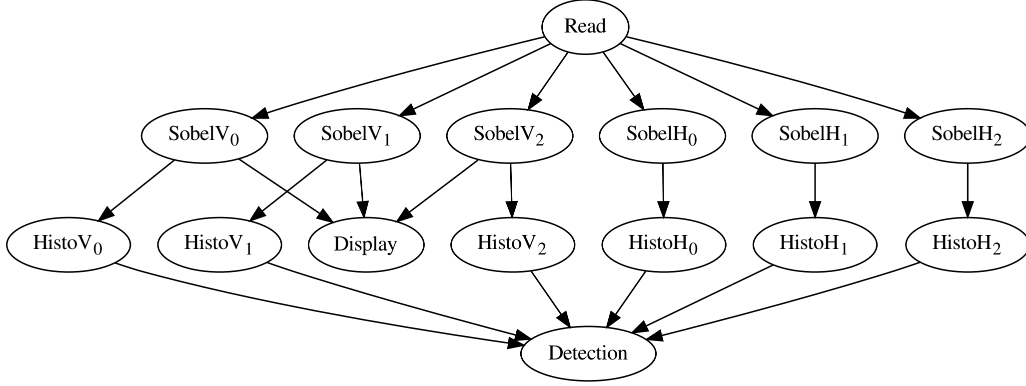


Figure 3.8: APG of the SDF graph (Figure 3.7 with X=3)

		Symmetry breaking							
X (p=1)	n.of tasks (messages)	None		Task		CPU		Both	
1	7 (7)	0.7	2	0.7	2	0.7	2	0.7	2
2	11 (14)	0.9	18	0.7	7	0.7	5	0.6	5
3	15 (21)	5.4	28	1.5	28	2.9	26	1.1	20
4	19 (28)	> 1800	80	159	58	> 1800	81	50	33

Table 3.1: CP solver optimization: *time(s), memory(MB) problem performance when raising the breadth of the graph*

3.3.1.1 Symmetries in the application model

According to the results of Table 3.1, a very symmetric problem is hard in practice. We break symmetry in the application model for the parallel tasks by ordering the start variable of $Sobel_H_X$ and $Histo_H_X$ tasks (for instance according to their index X). It is established that when the application model introduces symmetry, such as when transforming an SDF graph to its APG, the symmetry can be cancelled by ordering the tasks [91]. However we observe that because the image processing kernels operate on image cells (which can be as small as necessary), they are actually embarrassingly data parallel agents that can be split in a lot of fine grain tasks. Due to the data dependencies (e.g. from the Sobel agent to the Display agent) our Parallel agents model does not exactly fit SDF agents thus we could not use it to describe an application model without symmetry. We break symmetry by ordering the tasks, and we illustrate the use of parallel agents in another example (Section 3.4.2).

p (X=0)	1	2	3	4	5	6	7	8	9	10	11	12	13
n.of tasks	7	14	21	28	35	42	49	56	63	70	77	84	91
n.of messages	7	14	22	30	38	46	54	62	70	78	86	94	102
opt time(s)	0.4	0.8	1.2	2	3.2	4.9	7.2	11.3	17.6	21	26	28	> 300
mem peak (MB)	4	10	21	89	199	426	701	1195	1882	2936	4143	5872	> 8000

Table 3.2: CP solver optimization problem performance when raising the depth of the graph

3.3.1.2 Symmetries in the architecture model

When we add some symmetry breaking for resources by mapping the "Read" agent on one resource we observe a great improvement in memory consumption and solver time which means that the solver actually explores symmetric solutions without figuring out that they are the same. Breaking symmetry in the architecture model can be achieved either artificially by introducing carefully chosen additional constraints (such as mapping some tasks to one resource). In this Section we try to find a more generic approach which consists in modeling homogeneous resources as a quantity instead of modeling them individually. In Section 1.4.2 we described typical hardware architectures. A widely used architecture is the SMP. Since in SMP architectures the computing resources are identical a natural idea consists in modeling them as a quantity of available resources, which in the field of Operations Research is known as "cumulative" resource scheduling.

A non cumulative resource can run only one task at a time (it is mutually exclusive). A cumulative resource on contrary can run a limited number of tasks at a time. Cumulative models can also apply to communication resources modeling, when a communication media can be shared up to a limited bandwidth.

As we observed in the simple example in Section 1.8.2, we can model exclusive resources using the disjunctive predicate defined in Equation 2.8 for two tasks T_0 and T_1. Since $disjuncive(T_0, T_1) <=> cumulative([T_0, T_1], 1)$ and given a 2-cumulative resource and the three tasks T_0, T_1, T_2, we can implement a 2-cumulative constraint with the predicate "at least two tasks out of three do not overlap":

$$\begin{aligned} cumulative([T_0, T_1, T_2], 2) : \ & cumulative([T_0, T_1], 1) \\ & \vee\, cumulative([T_1, T_2], 1) \\ & \vee\, cumulative([T_0, T_2], 1) \end{aligned}$$

and we can generalize to a C-cumulative resource (at most C tasks will overlap)

$$cumulative(T_0, ..., T_{C+1}) : \bigvee_{(i,j) \in \binom{T+1}{2}} disjuncitve(T_i, T_j) \tag{3.4}$$

Generating it for every set of $C + 1$ tasks is a strong limitation even though some tasks will not overlap due to application constraints (precedences and messages). The solver is able to simplify the cumulative predicate accordingly, but avoiding those extraneous constraints can save a lot of pre-processing time.

Another approach consists in calculating for each instant the amount of active tasks associated to each resource. Let $\mathcal{T}$ be the discrete set of instants, we can formulate this using a new variable $t \in \mathcal{T}$ and using quantifiers ($\forall$):

$$disjunctive(T_0, ..., T_N) : \forall t \in \mathcal{T} : \sum_{u \in (T_0, ..., T_N)} (u.start \leq t \leq u.stop) \leq C \tag{3.5}$$

This latter approach requires either to use quantifiers constructs in the constraint programming model, or requires to instantiate a lot of new variables which is tricky in both cases.

3.4 Realistic use cases

We previously studied the case of a platooning application to highlight symmetry issues. In Section 3.4.1 we consider the celebrated Fast Fourier Transform application graph and in Section 3.4.2 we illustrate the use of parallel agents for the modeling of a data parallel application graph.

3.4.1 Fast Fourier Transform application

The graph presented in Figures 3.9 is a parallelized version of the Cooley-Tukey implementation of the integer 1D radix 2 FFT [23]. The FFT has a recursive nature, as the task graph of the FFT on 2^{n+1} inputs is obtained by instantiating twice the FFT on 2^n inputs and then adding 2^n tasks. For instance, the task graph of an 8-input FFT, provided in 3.9(middle) can be obtained by instantiating twice the task graph of the 4-input FFT, and then adding the 4 tasks of the bottom row. In each of the 3 task graphs of 3.9, nodes are tasks and arcs are data dependencies. All dependencies in an FFT transmit the same amount of data.

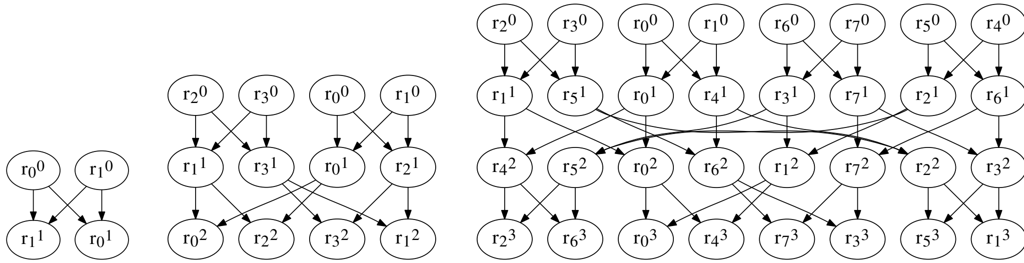


Figure 3.9: From left to right: FFT task graphs for 4, 8 and 16 inputs

In previous work we use the KRG model of computation to analyze this FFT graph running on a network on chip architecture (including models of the routers) using the KPASSA tool [68]. KRG is a model of computation (similar to SDF) that we don't fully describe here. Our analysis allowed us to encode both routing decisions, messaging and processing of the FFT into one DAG which is thus a valid partial order for the execution. In this work on contrary we focus on finding a valid complete order, on another architectural platform.

The solver was only able to produce optimal schedules for small problems (12 tasks and 16 dependencies), which is consistent with our synthetic example experiments. Note that we were unable to exploit here the symmetry breaking technique, because the FFT task graph does not use split/merge parallelism.

Table 3.3: Fast Fourier Transform CP resolution time at different levels of the recursion

FFT size (n. of inputs)	n. of tasks (messages)	opt. time (s)	mem. peak (MB)
2	1(0)	0.5	4
4	4(4)	1	4
8	12(16)	2	18
16	32(48)	> 1800	271
32	80(128)	> 1800	> 8000
64	192(320)	> 1800	> 8000

3.4.2 Radar application

The radar application is a signal processing use case based on the previously studied FFT (in Section 3.4.1). We consider a coarser problem in which the FFT is not the whole application but is one agent. The radar application consists in running the FFT on an image instead of a single signal which means each row of pixels needs to be transformed to the frequency domain, as well as each column of pixels. Application specifications typically perform this by transposing the image in between horizontal and vertical FFT (also known as "corner turn" operation [69]). We consider that the system also has to communicate with an operator and we associate an independent networking task to it. Horizontal and vertical FFT are typical data-parallel agent examples. We consider an image of size 2 megabytes and a simple architecture model made of (2, 3 and 4) homogeneous resources running at 1GHz, with the (synthetic) homogeneous costs resp. $(2, 20, 20, 3, 2, 5)10^6 cycles$ for the tuple of agents $(Read, HFFT, VFFT, MT, Write, Net)$.

The resulting schedules can be found in Annex B.2. Figure B.3 illustrates the ability of the parallel agent model to find "sensible" schedules: in the static model, the solver is not able to "align" the end of HFFT tasks (schedules on the left) whereas it is able to do so when we allow variable durations for those tasks (schedules on the right). In Figure B.4 we illustrate two communication systems model (Bus and Crossbar) and we show their effect on the final schedule: the ability of the crossbar interconnect model to overlap messages is used whereas with a bus model the messages have to be sequenced. In all the experiments but one the optimal makespan was found (10 minutes timeout).

3.5 Real use-case

In the last use case of Cholesky matrix factorization we intended to use our approach for parallel code generation. We study a tiled Cholesky matrix factorization (Section 3.5.1) running on an ARM big.LITTLE embedded platform (Section 3.5.2). A real use case raises new concerns, such as doing a correct estimation of kernel costs (Section 3.5.3).

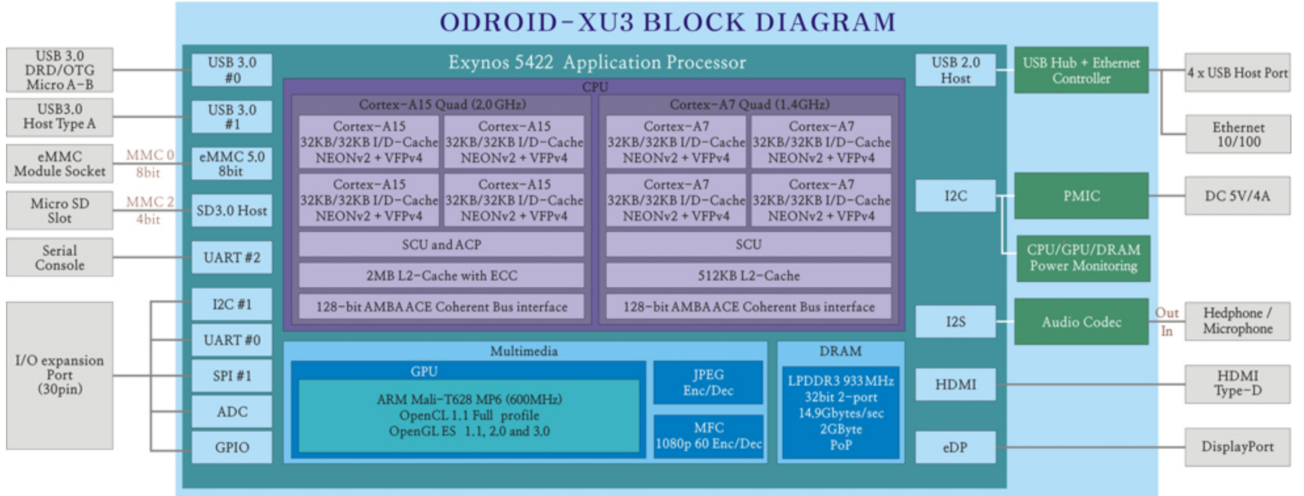


Figure 3.10: Block diagram of the Exynos 5422 Processor (source: reference manual)

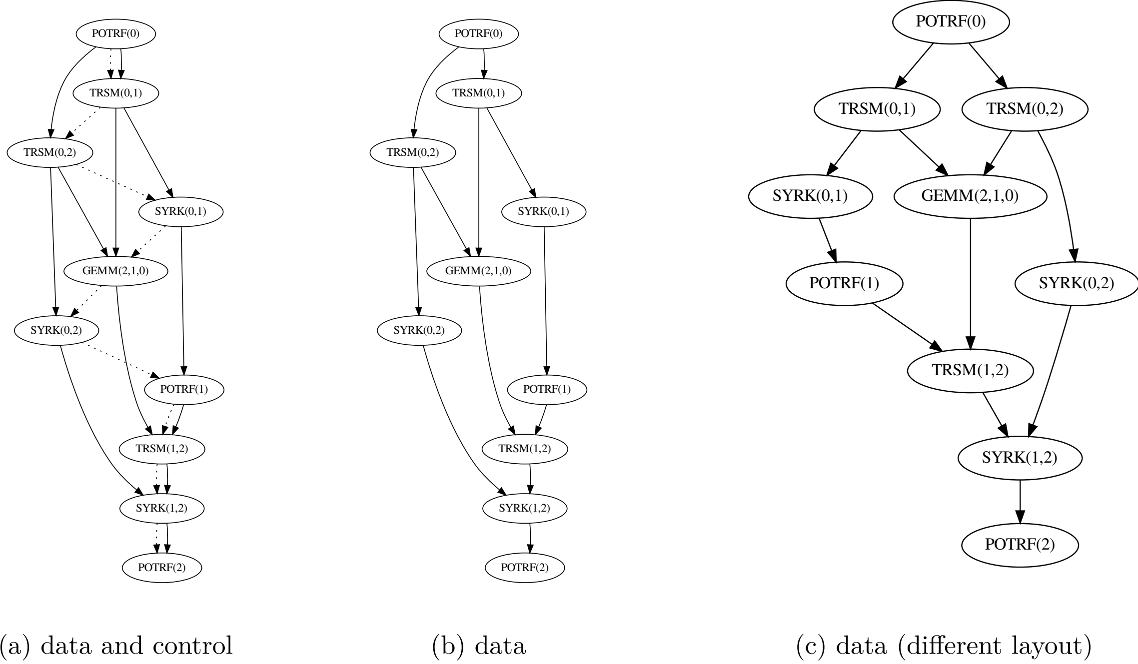
3.5.1 Cholesky matrix decomposition

The blocked Cholesky matrix decomposition defines a structured but complex graph. We can build it from the definition of the sequential algorithm 4 in the form of a SANLP. Given a number of tiles T of matrix A we use the tiled Cholesky algorithm (Figure 4) to compute the graph of dependencies between the BLAS level-3 functions (GEMM, TRSM, POTRF and SYRK). We obtain the graphs in Figure 3.11.

Algorithm 4 Pseudo-code for the tiled Cholesky algorithm

```
Require: number of tiles: T
Require: tiled matrix: A
  for k=0...T do
      A[k][k] = POTRF(A[k][k])
      for m=k+1...T do
          A[m][k] = TRSM(A[k][k], A[m][k])
      end for
      for n=k+1...T do
          A[i][i] = SYRK(A[n][k], A[n][n])
          for m=n+1...T do
              A[n][m] = GEMM(A[m][k],A[n][k],A[m][n])
          end for
      end for
  end for
```

Figure 3.11 shows three task graph models of a tiled Cholesky algorithm for a matrix of 3x3 tiles. Dotted edges in Figure 3.11a represent the sequential execution corresponding to the pseudo-code 4. We identify separately each function call depending on which tile it reads or writes, hence the numbering in the resulting graph. We add a dependency to the graph for each corresponding read and write of the input tiled matrix. Figure 3.11b shows the same graph without sequential edges, and Figure 3.11a is an isomorphism which layout better shows the available parallelism.



(a) data and control (b) data (c) data (different layout)

Figure 3.11: 3x3 tiled Cholesky factorization task graphs

Annex B.1 gives the same graph layouts for 4x4 and 5x5 matrices which as expected provide higher task parallelism as the number of tiles raises.

3.5.2 Experimental setup: The ARM big.LITTLE platform

We run all the experiments on an embedded development board (Odroid XU3) featuring a Samsung Exynos5422 ARM big.LITTLE processor of four A7 cores and four A15 cores such as the one described in Figure 3.10. The big.LITTLE technology is an heterogeneous MP-SoC architecture designed by ARM. Samsung implemented it in the Exynos5422 Chip, which was used by the Korean company “Hardkernel” to build the Odroid board. The platform runs a Linux Ubuntu 15.10 operating system with a 3.10 kernel that was patched in order to enable hardware cycle counters. In the experiments described in Section 3.5.4 we sometimes consider 1,2 or 4 cores of each (big and LITTLE). The platform is equipped with instant power sensors that allow us to sample instant power of each cluster (of big or LITTLE cores) and a performance monitoring unit (PMU) that allows to access relevant metrics such as cycle counts through the perf Linux API [30] (Annex A.3).

State of the art (dynamic) scheduling consists in clustering the resources such that the resulting architecture is homogeneous (considering a cluster as a computing resource). We can thus use three configurations (Figure 3.12)

- Only the A7 cores (A15 cores are shutdown)
- Only the A15 cores (A7 cores are shutdown)
- Use each pair of A7 and A15 core

This technique is known as In Kernel Switching (IKS, Figure 3.12). The Global Task Scheduler was then developed to handle the case of heterogeneous computing resources.

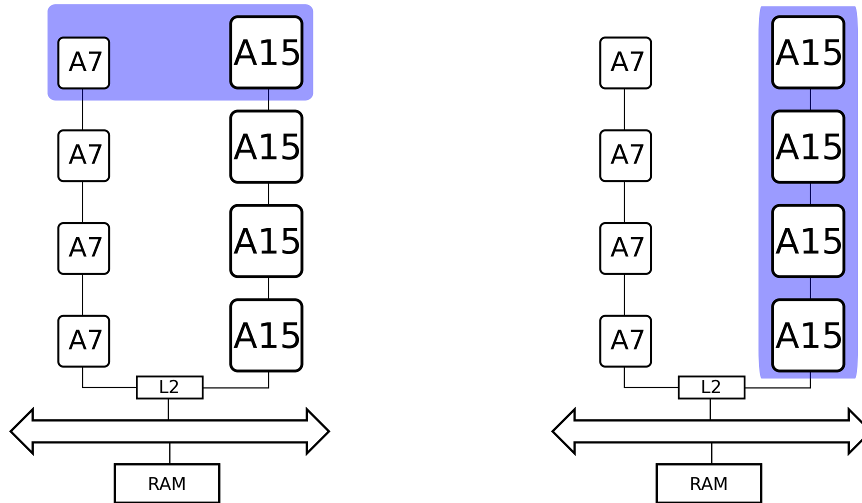


Figure 3.12: In Kernel Switching big.LITTLE configurations

The memory organization is a two levels cache hierarchy. Thus in examples involving this architecture we do not consider communications costs since we cannot do explicit accesses to caches.

3.5.3 Practical aspects of meta programming

Using our approach on a real use case first requires a work estimation of the different kernels. In StreamIt, the work estimation phase is a static analysis. Static analysis does not require actual hardware, whether benchmarking requires it. Static analysis is tricky in practice: StreamIt authors report that actual runtimes may differ with a factor two. Sometimes it it is not applicable (for instance when a StreamIt program calls an external routine). SPIRAL on contrary benchmarks a set of atomic tasks (for instance a set of FFT kernels). We also found evidence in the previous chapters that in a shared memory architecture, if the model does not account for communication and does not validate any hypothesis on the atomic tasks, interferences on the memory hierarchy may occur.

3.5.3.1 Kernel cost estimations

We estimate the cost of the task with their number of cycles when executed on the target Odroid-XU3 system. In this Section we examine and challenge the low-variability hypothesis. Figure 3.13 shows the number of cycles when a single threaded BLAS (from the OpenBLAS benchmark) is executed on the target, along with the Violin error plot and the extreme values. We observe a very stable execution for each of the BLAS kernel we use. We use the coefficient of variation as the variability metric (standard deviation over mean), the results are presented Table 3.4 and Figure 3.13. We perform those measurements by reading the hardware counters through the "perf_events" Linux kernel interface [30]. We use the costs defined in Table 3.4 in the next sections in order to obtain parallel schedules for this

Table 3.4: Costs of the BLAS kernels (10^6 cycles)

		GEMM	SYRK	TRSM	POTRF
A15	mean	11	6	7	9
	c_v (%)	0.29	0.21	0.19	0.08
A7	mean	39	20	21	16
	c_v (%)	0.11	0.23	0.11	0.64
ratio		3.5	3.3	3	1.7

heterogeneous architecture. In this experiment, the costs of the BLAS kernels are in the same order of magnitude. When the problem combines very small costs with very large costs, the approach may require either to omit the small tasks or to merge them such that the costs of the jobs remain in the same order of magnitudes.

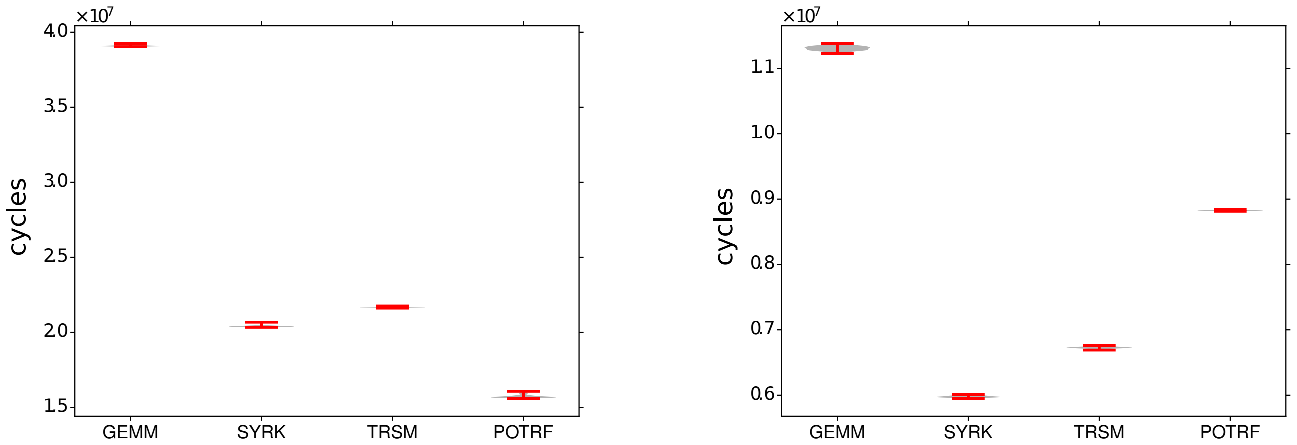


Figure 3.13: Variability of the BLAS kernels

3.5.3.2 Controlling core pinning and core frequencies

In practice, DVFS is implemented for computing resources in most of servers, laptops, embedded systems, and even microcontrollers. It is implemented in the Linux kernel we use on the Odroid platform. Interconnect and Memory frequency scaling from an operating system is a much more exotic feature that is still limited to very specific (and supported) hardware. Controlling CPU frequencies is done with the well known cpufreq tool. Controlling the frequency of other devices, including interconnects and memory is or will be implemented in the devfreq tool which is not yet available for Exynos 5422.

3.5.4 Results

Some recent experiments on static scheduling of the Cholesky factorization (using an heuristic, and on another architecture) suggest that unexpectedly static scheduling is competitive with dynamic scheduling [2]. We find similar results (Section 3.5.4.2) and additional results regarding energy related issues in Section 3.5.4.3 and scalability of the SMT solver approach (used to perform this scheduling) in Section 3.7. We give an example of generated constraints in Section A.5.

3.5.4.1 Multiobjective optimization

Since the A7 little core can run at the lowest instant power available on the platform, the schedule that minimizes energy is the one that uses only the A7 little core, whatever the resulting makespan. Regarding makespan optimization, the solver selects the highest frequencies (hence the fastest but also the highest instant powers) but there is no straightforward solution. It is thus legitimate to search which level of performance we can maintain given an energy budget, or on contrary how much energy we can save if we give the system some slack time.

3.5.4.2 Makespan optimization

As expected when the number of tiles grows, the number of tasks and task dependencies also grows. The overall number of samples is constant, thus the number of samples per tile decreases when the number of tiles increases. The costs obtained in Table 3.4 have been updated with the corresponding values for the tiles of different sizes corresponding to our experiments. We were able to solve the non trivial examples of 3x3 and 4x4 tiled Cholesky algorithm running on two heterogeneous cores optimally, and we got a sub-optimal (but still optimized) result for the 5x5 tiled Cholesky algorithm. Because the solver is very efficient at finding satisfiable solutions when there is a lot of slack time [40], even with a much larger number of tasks and messages (resp. 35 tasks and 60 messages in the 5x5 example), it is able to find a solution, and to get an optimal or optimized solution. We experiment different problems with higher theoretical complexity and we report their practical complexity in Section 3.2.

Because the solver approach also allows to implement heuristics by doing successive calls to the solver and changing the objective function, we can compare the obtained schedules to one of the greedy "As Soon As Possible" (ASAP) schedules. In order to obtain this schedule we sort the tasks by topological order and we add them in this order to the set of constraints, asking for makespan minimization: the strategy becomes greedy, hence sub-optimal but allows as expected to obtain schedules for large instances. We also compare our results to the naive sequential implementation with all tasks running on the fastest resource, at the highest frequency. In either case we obtained significant improvement. For the 5x5 tiled Cholesky algorithm graph, we obtained an optimized but sub-optimal result, still better than naive and ASAP implementations. The results also suggest that the divide and conquer strategy actually works, since the makespan decreases when the number of tiles increases (for the same matrix size). The exhaustive results are presented in Table 3.5.

Table 3.5: Results obtained with the modeling and solving (time in milliseconds)

	3x3	4x4	5x5
N.of tasks (and messages)	10 (12)	20 (30)	35 (60)
Optimal makespan	46	39	≤ 38
Sequential makespan	55	51	49
ASAP makespan	55	45	44
Energy optimal makespan	350	349	343

The synthesized implementation is compared to those predicted values and its predictability is evaluated with the same coefficient of variation used in Section 2.3.2. The results are presented in Table 3.6. Of course running those programs requires a high level of privileges since it requires access to critical hardware features, such as modifying frequencies. This is acceptable on an embedded system since there is probably only one user (sometimes there is even no operating system) but it is probably not acceptable on a shared server with no proper isolation. When the tiles are too large,

Table 3.6: Predicted vs Traced execution time for the tiled Cholesky factorization

	1big+1LITTLE			4big+4LITTLE		
	3x3	4x4	5x5	3x3	4x4	5x5
Predicted (ms)	46	39	38	37.9	24.5	17.8
Execution (ms)	53.2	40.5	39.6	38.2	25.2	18.6
Variability (%)	0.89	0.53	0.77	0.65	0.38	0.67

input data of the BLAS kernels do not fit in the data caches. This can increase variability and lower performance, as the result for the 3x3 case suggests. Adapting tile sizes to cache size such that the generated schedule restricts as much as possible the access to shared levels of the memory would be an interesting addition. The 4x4 and 5x5 results do not suffer from this issue because they compute on smaller tiles. The results in those latter cases are very close to the predicted results, which suggests a very faithful model.

3.5.4.3 Energy optimization

As well as for makespan optimization, the obtained makespans when optimizing for energy are very close to the predicted values. We now generate Pareto fronts for those models and examine the corresponding implementations. Figure 3.14 shows a subset of the schedules that are on the Pareto front for the 3x3 algorithm on a 1+1 big.LITTLE architecture. The power sensor that was used in the Section 1.4.3 in an on-board device that is connected to the big.LITTLE through an I^2C Bus. It is not very precise and allows sampling at a slow rate (not at the 1kHz rate that we would need). However, we can at least check the instant power when running different implementations, and the very precise timings allow to compare the consumed Energy.

The fastest theoretical schedules for 3 and 4 tiles report a 52.5mJ and 45.1mJ energy consumption. The instant power is sampled when the implementations run and we obtain 1.24W and 1.27W, which corresponds (according to the obtained execution times of Table 3.6) to energy consumption of 66mJ and 51mJ. We do the same for the schedules where the time budget is twice the optimal makespan, those experiments report (resp.) and 30.0 and 30.5mJ, and the implementation experiments report 45.4 mJ and 46.3 mJ. Although the model is not correct, we obtained valid implementations within a given time budget, with a significant energy optimization.

We were able to perform the experiments with a larger number of cores to fit the existing platform. Figure 3.15 shows the makespan optimal schedule for the 4x4 tiled Cholesky factorization running with all available 4+4 big and LITTLE cores. We were able to obtain optimized energy schedule by giving twice the obtained makespan as a time budget. In this case we observe that as expected fewer big cores are used (only one) and it runs some of its allocated tasks to a lower frequency.

3.5.4.4 Further scalability results

There are a number of direct extensions one can apply to the previous modeling and encoding effort, with in mind the idea of checking how the method can be pushed to its limits in terms of complexity regarding solver efficiency. Already the parametric aspect of Cholesky algorithm allows to scale up a range of applications. Modeling the multiplicity of process frequencies, or multiple cores in each compute block also increases the demand on solvers. We experiment with such extensions now, looking for limitations as they occur. As the Figure 3.10 suggests, our experimental setup allows to run up to

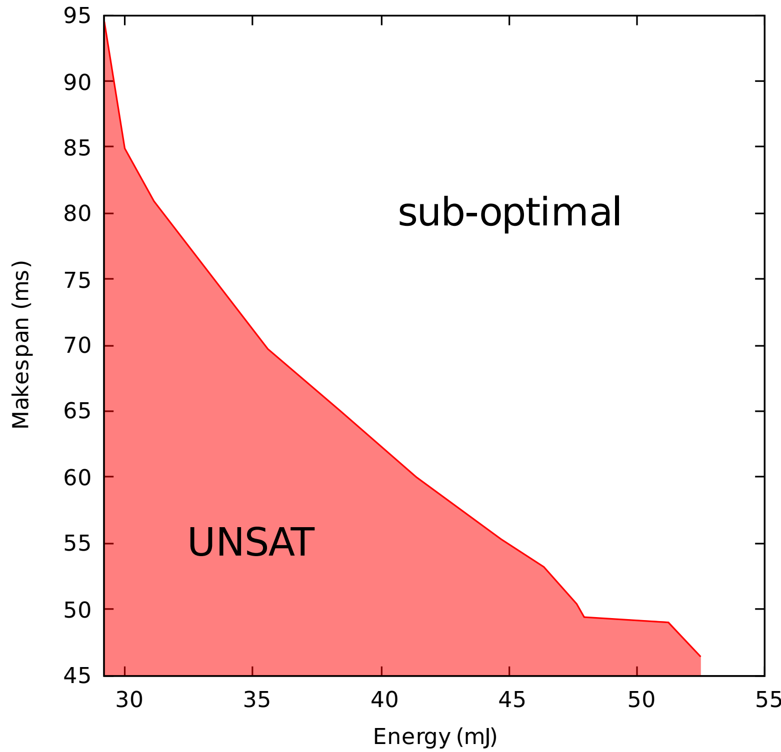


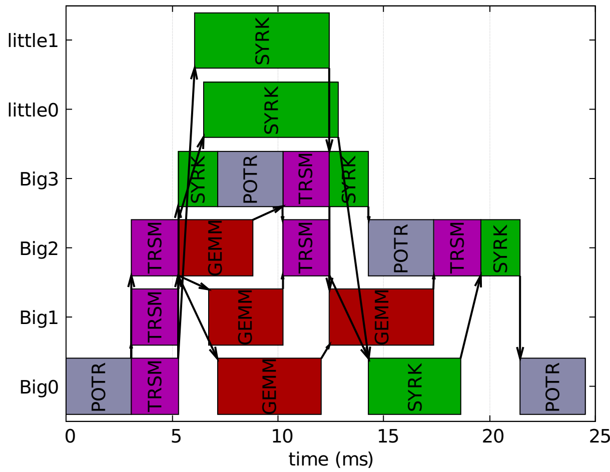
Figure 3.14: Pareto front for the 3x3 algorithm

4 big and 4 little cores. In the previous experiments we have been using only 1+1 cores. We synthesize the results regarding scalability in Table 3.7.

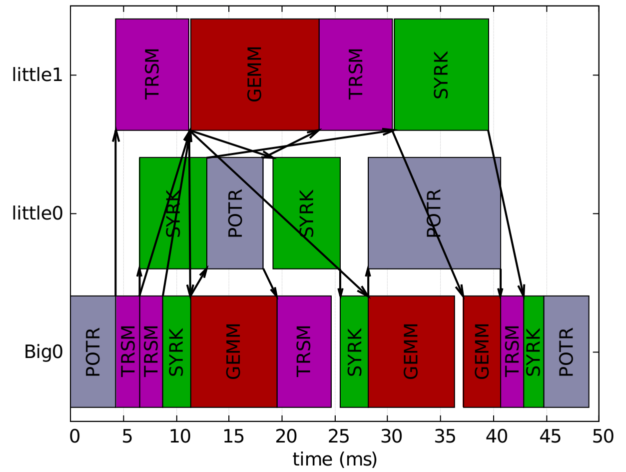
Table 3.7: Solver time vs number of cores and number of tiles

		#cores (big+LITTLE)		
		1+1	2+2	4+4
#tiles	3x3	2s	2s	3s
	4x4	1min48s	8s	11s
	5x5	>1h	10s	28s

We observe that although the scheduling complexity is on the whole related to the size of the problem (the system of constraints to solve), this connection is not as regular as it may seem, and the impact of whether regular or irregular solutions emerge can dominate the size complexity. So certainly much more remains to be done on predicting when (and how) relative dimensioning may favor regularity, and so solver efficiency, in the universe of solutions. Moreover, although in the makespan optimization problem the highest frequencies alone can achieve the optimal results, some of the optimal results can include low frequency tasks (as figure 1.3 suggests). We observed for instance an impact of the number of operating points on the solver time in the "4 tiles running on 2 little and 2 big cores" problem. 6x6 tiled problems timed out after one hour. The irregularities in the scalability suggest that the constraint programming models can be further improved.



(a) Optimal makespan



(b) Energy optimized for twice the optimal makespan

Figure 3.15: 4x4 tiled Cholesky schedules on 4+4 big.LITTLE

3.6 Summary on the SymSched approach

Through those several use cases we show that Makespan optimization scales to practical problem sizes even when large number of resources are involved. Some pathological cases won't reach the optimal value fast but most of our examples suggest it converges fast to an optimal value. Energy optimization does not scale as well but we were still able to obtain sensible results. Using SMT solvers raises a lot of technical challenges. We believe the models, the encoding and the solver tactics can still be improved to further scale. Nowadays embedded systems have a small number of resources which advocates for this approach. In very large problems using approximate solutions or heuristics is probably necessary in many cases but we believe it can be efficiently combined with exhaustive techniques by limiting it to sub-problems of reasonable sizes.

Static scheduling is as expected cumbersome because it requires to study carefully the costs of the tasks and their interactions. The ever increasing computer architecture complexity is worrying and many users consider that it is too complex now to be controlled efficiently. They thus rely to default behaviors (of the scheduler, the energy consumption policies) or legacy practices. Achieving static scheduling and validating it (such as in [2] and in this work) give encouraging insights that we can handle this complexity.

The burden of understanding how we can limit what we cannot control (such as cache memory) puts into perspective the legacy microarchitecture consisting of hardware managed multi levels of caches. The industrial tendency to other architectures and in particular to on-chip distributed memory (or delegated cache consistency to software) underlines this feeling.

Conclusion

3.7 Conclusion and future work

In the current work we have attempted to deal with the optimized mapping of embedded applications onto embedded (multicore) architectures, with as objective to minimize power consumption while preserving minimal performance requirements. We challenged the ability of automated approaches such as SMT solvers to provide good solutions to this allocation and scheduling problem. On first thought it seems that these techniques should be well adapted, since the very nature of the problem involves propositional Boolean constraints, reflecting the alternatives of task placement and operating performance points (for choice of frequency and voltage levels in processors), with linear (in)equalities regarding the real-time schedulabiliy conditions and accumulated time/energy budget, amongst other things.

Still, practical solver complexity was always lurking somewhere in our approach, and naice approaches would not simply scale to reasonable analysis time. So it soon became clear that most of the effort consisted in finding the good compromises in restricting the modeling expressiveness, while taking advantages of the relevant modeling features to improve or guarantee some form of efficiency of our tools.

Because the structure of the Boolean variables involved in our constraint sets reflect somehow the structure of possible mappings, it may look on an abstract level that he SMT resolution consists in, first proposing a given mapping, second evaluate its practical cost and check whether it is admissible; the internal wizardry of SMT solvers makes it somehow complex to understand why solvers were guided towards the proposed solution.

The approach developed here, which deals essentially with time in a symmetrical way, may be contrasted with more common approaches to scheduling and allocation, where tasks are mapped onto resources in a chronological order, "filling up" Gantt diagrams from the start at each resource lane. These alternative approaches act as symbolic simulation, performing allocation dynamically, and only recognizing on static-control systems whenever a state already encountered in the past is met again, then loop around this iteration. Consisting essentially of *greedy algorithm* approaches, where previous choices are not reconsidered and undone, despite the NP-completeness challenge of good mapping, they may sometimes use auxiliary computed values so that the choice made at a given stage takes into account the ones made in the past (such as for instance through the current workload of a task, or its current priority due to its position relative to others).

It would certainly be interesting to investigate further the comparative merits of the mechanisms involved in the SMT-solving process, and those at work in the "symbolic simulation / greedy allocation" approach. At minima, a tentative comparison of the "quality", or even nature, of mapping results obtained by each approach could be instructive. Further, trying to devise a way combining the two (and maybe refining one by the other) could be a topic of reflection for future work.

Appendices

Appendix A

Code samples

A.1 Stress test examples

Code sample to stress a 4096 Bytes memory area on cores #0 and #1 with the linux “stress” utility:

```
1  taskset -c 0 stress -m 1 --vm-bytes 4096 &
2  taskset -c 1 stress -m 1 --vm-bytes 4096 &
```

Code sample to stress CPU #0 and #4 with the linux “stress” utility:

```
1  taskset -c 0 stress -c 1 &
2  taskset -c 4 stress -c 1 &
```

A.2 Calls to the GNU Scientific library

```
1  gsl_matrix *A,*B,*C;
2  ...
3  r = gsl_blas_dgemm(CblasNoTrans, CblasTrans, -1.0, A, B, 1.0, C);
```

A.3 Calls to the perf API

A.3.1 Setting up performance counters on Odroid XU3

Obtaining those costs requires some kernel modification of the default Odroid XU3 kernel:

- Enabling compile-tile kernel options

  ```
  CONFIG_PERF_EVENTS=y
  CONFIG_HW_PERF_EVENTS=y
  ```

- Enable the Performance Monitoring Unit in the Device Tree
  (file `arch/arm/boot/dts/exynos5422_evt0.dtsi`)

- Build and install the perf tool

A.3.2 Using the C perf API

```
static long perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
                int cpu, int group_fd, unsigned long flags)
{
        int ret;
        ret = syscall(__NR_perf_event_open, hw_event, pid, cpu, group_fd, flags);
        return ret;
}
struct perf_event_attr pe;
long long cycles;
int fd,r;

memset(&pe, 0, sizeof(struct perf_event_attr));
pe.type = PERF_TYPE_HARDWARE;
pe.size = sizeof(struct perf_event_attr);
pe.config = PERF_COUNT_HW_CPU_CYCLES;
pe.disabled = 1;
pe.exclude_kernel = 1;
pe.exclude_hv = 1;
fd = perf_event_open(&pe, 0, -1, -1, 0);
assert(fd!=-1);
ioctl(fd, PERF_EVENT_IOC_RESET, 0);
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

        ...

ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
read(fd, &cycles, sizeof(long long));
printf("\%lld,", cycles);
close(fd);
```

A.4 A Proof of concept using z3

```
from z3 import Int, EnumSort, Const, set_option, Optimize, And, Implies, Xor
from pprint import pprint

Resource, (p1,p2) = EnumSort('Resource', ['p1','p2'])
a_stop, a_start, a_duration = Int('a_stop'), Int('a_start'), Int('a_duration')
b_stop, b_start, b_duration = Int('b_stop'), Int('b_start'), Int('b_duration')
c_stop, c_start, c_duration = Int('c_stop'), Int('c_start'), Int('c_duration')
a_map, b_map, c_map = Const('a_map', Resource), Const('b_map', Resource), Const('c_map', Resource)

set_option("verbose", 10)

s = Optimize()
s.add(And(a_start>=0, b_start>=0, c_start>=0))
s.add(And(a_stop <= makespan, b_stop <= makespan, c_stop <= makespan))
s.add(a_stop == a_start + 3)
s.add(b_stop == b_start + 5)
s.add(c_stop == c_start + 7)
s.add(Implies(a_map==b_map, Xor(a_stop<=b_start,b_stop<=a_start)))
s.add(Implies(a_map==c_map, Xor(a_stop<=c_start,c_stop<=a_start)))
s.add(Implies(b_map==c_map, Xor(b_stop<=c_start,c_stop<=b_start)))
makespan = Int('makespan')
```

```
22 s.minimize(makespan)
23 s.check()
24 print s.model()
25 pprint(s.statistics())
```

A.5 Generated constraints Cholesky big.LITTLE 1+1, 3 tiles

```
Or(Not(POTRF22map == 0), Not(3 <= POTRF22opp))
Or(Not(POTRF22map == 1), Not(3 <= POTRF22opp))
Or(Not(GEMM012map == 0), Not(3 <= GEMM012opp))
...
...
Or(Not(POTRF22map == GEMM012map),
   Not(POTRF22start + POTRF22duration <=
       GEMM012start) ==
   (GEMM012start + GEMM012duration <=
    POTRF22start))
Or(Not(And(POTRF22map == GEMM012map,
           Not(POTRF22opp == GEMM012opp))),
   Not(POTRF22start + POTRF22duration <=
       GEMM012start) ==
   (GEMM012start + GEMM012duration <=
    POTRF22start))
Or(Not(POTRF22map == TRSM20map),
   Not(POTRF22start + POTRF22duration <=
       TRSM20start) ==
   (TRSM20start + TRSM20duration <= POTRF22start))
...
...

((((Not(And(POTRF22map == 0, POTRF22opp == 0, POTRF22duration == 135)) ==
  And(POTRF22map == 0, POTRF22opp == 1, POTRF22duration == 189)) ==
  ...
((((Not(And(GEMM012map == 0, GEMM012opp == 0, GEMM012duration == 165)) ==
      And(GEMM012map == 0, GEMM012opp == 1, GEMM012duration == 231)) ==
  ...
((((Not(And(TRSM20map == 0, TRSM20opp == 0, TRSM20duration == 105)) ==
      And(TRSM20map == 0, TRSM20opp == 1, TRSM20duration == 147)) ==
  ...
...
POTRF11start >= SYRK10start + SYRK10duration
SYRK21start >= SYRK20start + SYRK20duration
GEMM012start >= TRSM20start + TRSM20duration
POTRF22start >= SYRK21start + SYRK21duration
...
makespan >= 0
```

```
makespan <= 2100
POTRF22start >= 0
POTRF22start <= 2100
POTRF22start + POTRF22duration <= 2100
POTRF22duration >= 0
POTRF22duration <= 2100
POTRF22opp >= 0
POTRF22map >= 0
POTRF22map <= 1
GEMM012start >= 0
GEMM012start <= 2100
GEMM012start + GEMM012duration <= 2100
GEMM012duration >= 0
GEMM012duration <= 2100
GEMM012opp >= 0
GEMM012map >= 0
GEMM012map <= 1
10080*If(And(POTRF22map == 0, POTRF22opp == 0), 135, 0) +
  10080*If(And(GEMM012map == 0, GEMM012opp == 0), 165, 0) +
  10080*If(And(TRSM20map == 0, TRSM20opp == 0), 105, 0) +
  10080*If(And(TRSM10map == 0, TRSM10opp == 0), 105, 0) +
  ... >= 0
6050* If(And(POTRF22map == 0, POTRF22opp == 1), 189, 0) +
  ...
...
```

A.6 Solver logs and relevant tactics descriptions

```
(simplifier :num-exprs 6542 :num-asts 12695 :time 0.01 :before-memory 1.56 :after-memory 2.04)
(propagate-values :num-exprs 6542 :num-asts 12695 :time 0.00 :before-memory 2.04 :after-memory 2.04)
(solve_eqs :num-exprs 8179 :num-asts 15015 :time 0.01 :before-memory 2.04 :after-memory 2.26)
(:num-elim-vars 92)
(simplifier :num-exprs 8179 :num-asts 15015 :time 0.00 :before-memory 2.26 :after-memory 2.37)
(dt2bv :num-exprs 8179 :num-asts 15015 :time 0.00 :before-memory 2.17 :after-memory 2.17)
(elim01 :num-exprs 8319 :num-asts 18041 :time 0.01 :before-memory 2.17 :after-memory 2.41)
(cardinality-intro :num-exprs 8018 :num-asts 20084 :time 0.01 :before-memory 2.41 :after-memory 2.62)
(eq2bv :num-exprs 8018 :num-asts 20084 :time 0.00 :before-memory 2.73 :after-memory 2.73)
(simplifier :num-exprs 9770 :num-asts 21768 :time 0.01 :before-memory 2.73 :after-memory 2.97)
(optimize:check-sat)
(smt.simplifier-done)
```

```
simplify : apply simplification rules.
dt2bv : eliminate finite domain data-types. Replace by bit-vectors.
elim01 : eliminate 0-1 integer variables, replace them by Booleans.
lia2card : introduce cardinality constraints from 0-1 integer.
eq2bv : convert integer variables used as finite domain elements to bit-vectors.
```

Appendix B

Application graphs and gantt diagrams

B.1 Tiled cholesky

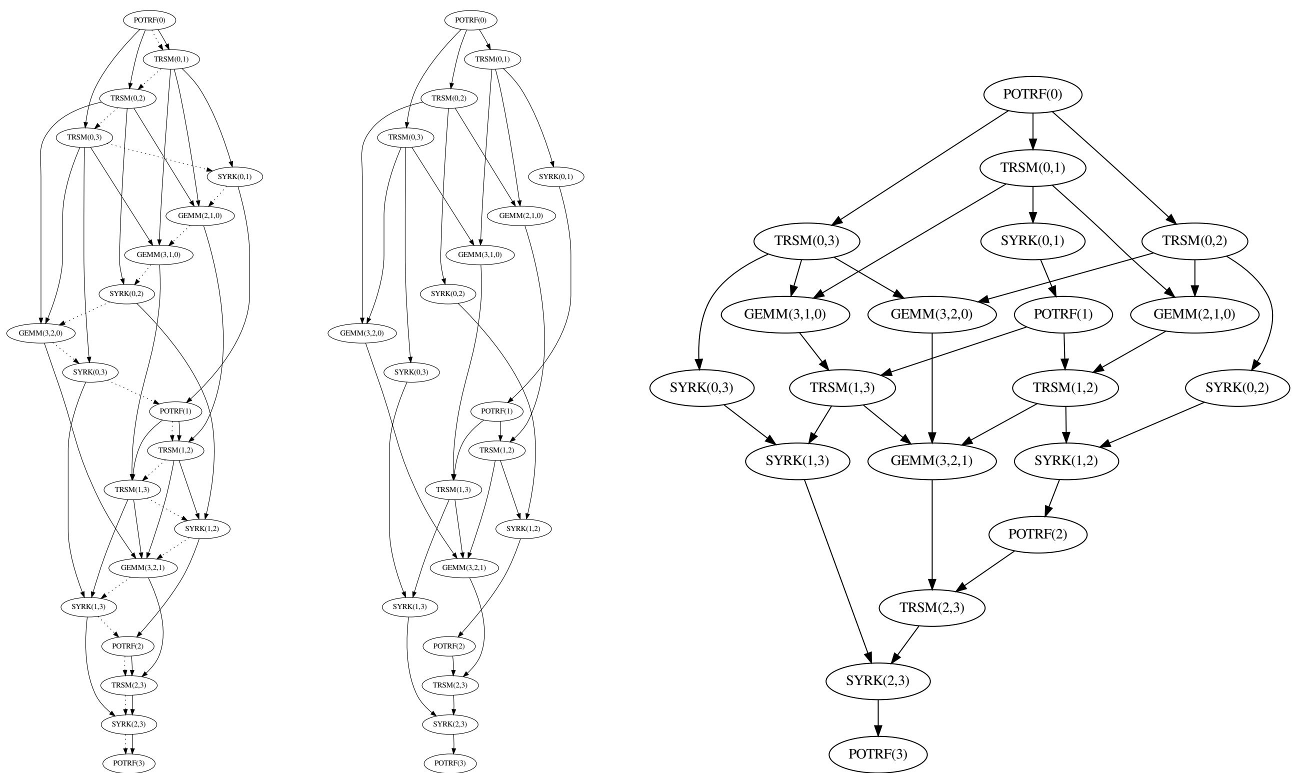


Figure B.1: 4x4 tiled cholesky factorization task graph

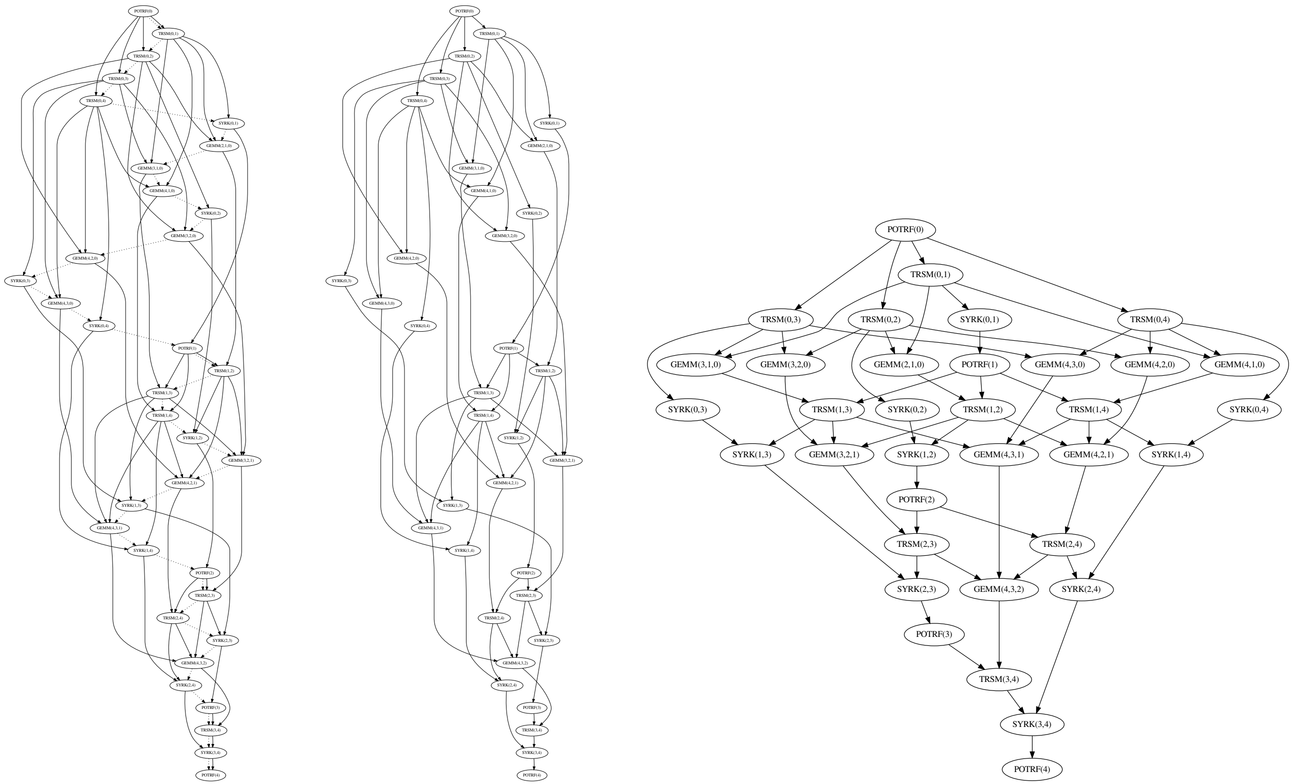


Figure B.2: 5x5 tiled cholesky factorization task graph

B.2 Radar application

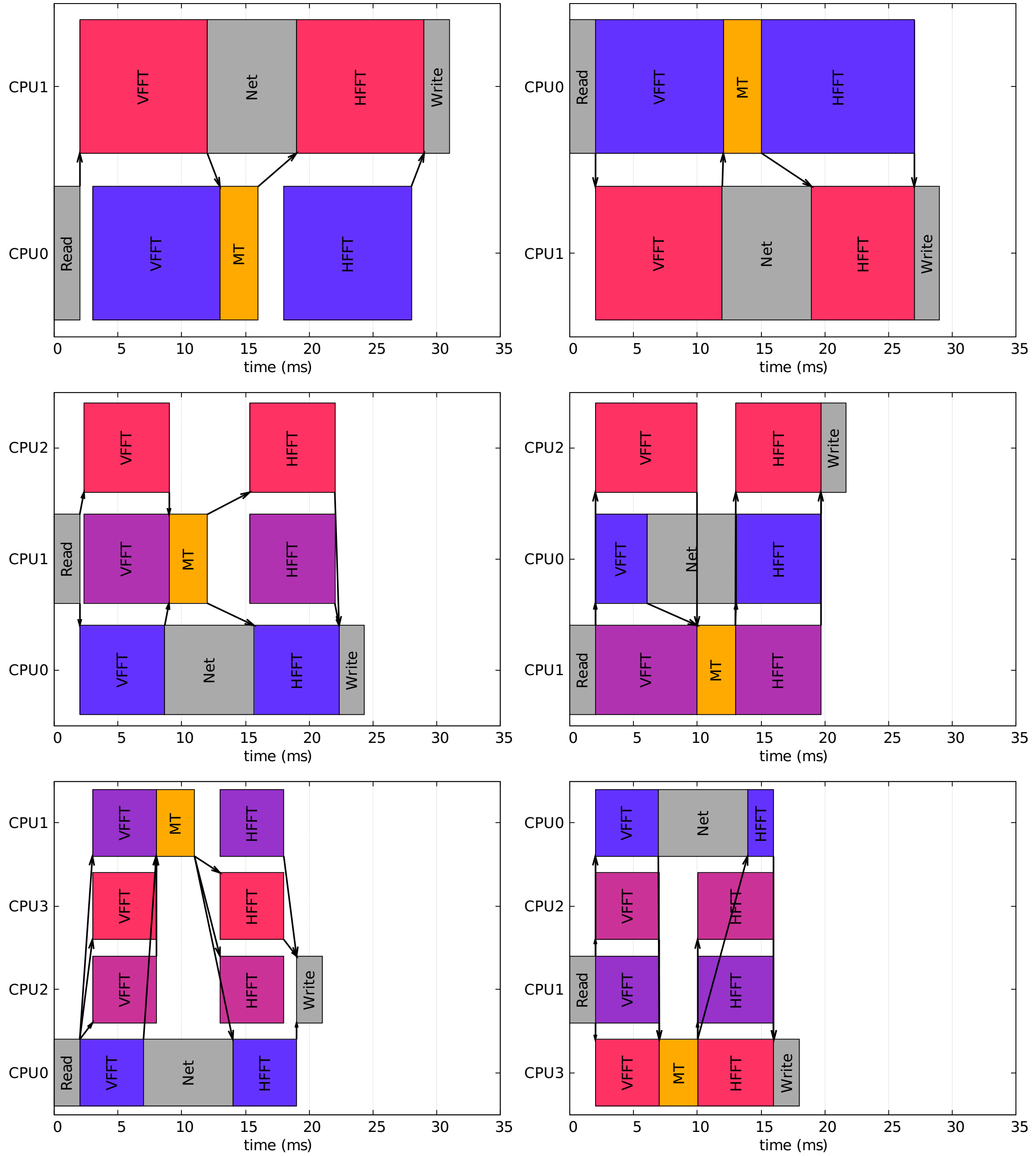


Figure B.3: Radar application Gantt diagrams: static (left) vs dynamic (right) agent models

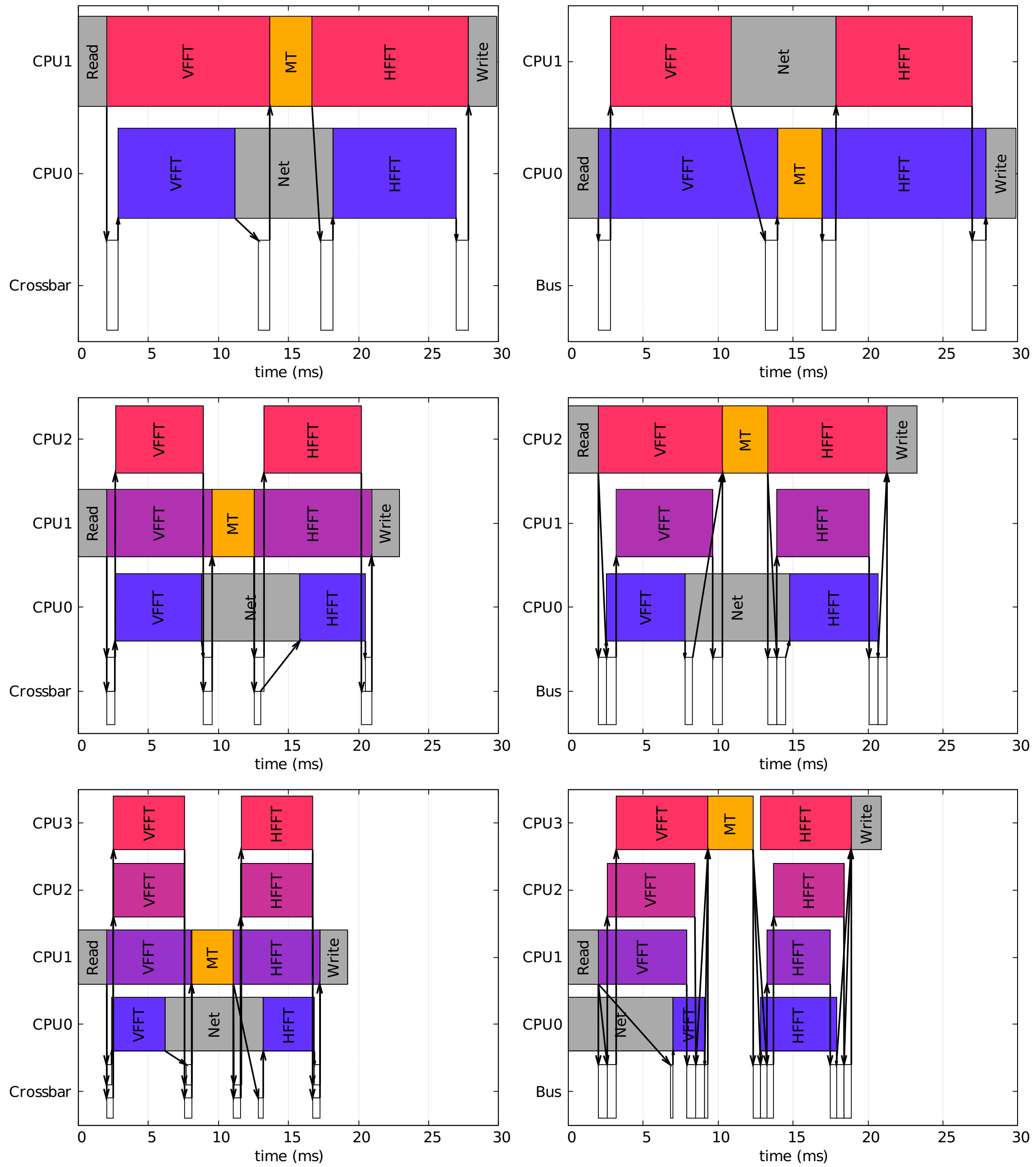


Figure B.4: Radar application Gantt diagrams dynamic agent models: Bus vs Crossbar interconnect models

List of Figures

1.1 The benefits of Model Driven Engineering . . . 17
1.2 The Ychart or AAA methodology . . . 18
1.3 Split/merge dataflow process network and its corresponding graph of tasks . . . 19
1.4 StreamIt graph of the blocked matrix multiply algorithm . . . 21
1.5 Model of a non preemptive task . . . 24
1.6 Model of a task preempted once . . . 24
1.7 Model of a task migrated once . . . 24
1.8 lstopo output on a laptop multicore computer . . . 28
1.9 Non energy optimized vs optimized split merge pattern (red tasks run with high instant power) . . . 31
1.10 Two graphical representations of the same schedule . . . 35
1.11 Detailed workflow of our approach . . . 41
1.12 3x3 tiled Cholesky task graph . . . 42
1.13 A simplified big.LITTLE architecture . . . 42
1.14 Makespan optimal solution . . . 43
1.15 Energy optimized solution . . . 43
1.16 Threads, pseudo-code and message passing for the 3x3 makespan optimal solution . . 43

2.1 UML diagram of the SymSched tool . . . 46
2.2 SDF meta-model . . . 48
2.3 APG (or Task Graph) meta-model . . . 48
2.4 A marked and live SDF graph . . . 49
2.5 Platooning APG graphs . . . 49
2.6 SymSched architecture model . . . 54
2.7 Static mapping of messages on the links of a ring architecture . . . 56
2.8 Conflicting routes on a 3x3 mesh network on chip . . . 56
2.9 A two by two hierarchical interconnect . . . 57
2.10 Instant power vs frequency on the Odroid XU3 . . . 58
2.11 Temperature elevation of the integrated GPU and four A15 cores in the Exynos chip when only two A15 cores are active . . . 59
2.12 Coefficient of variation vs memory stress . . . 63
2.13 Mapping of a parallel agent (blue, cost=24) combined with two regular tasks (gray) and no messages with static task costs . . . 65
2.14 Mapping of a parallel agent (cost=24) combined with two regular tasks (and no messages) with variable task costs . . . 66
2.15 Admissible schedules of a periodic task with $period = deadline$ . . . 67
2.16 Admissible schedules of a periodic task with $period \neq deadline$ . . . 67

2.17 Example pre-computed preemption 68

3.1 Example time discretization on frequency scaling 70
3.2 Practical complexity: success rate vs number of tasks (left) and nature of resources (right) 72
3.3 Practical complexity: success rate vs problem tightness (left) and solver time vs number of tasks (right) 73
3.4 Generated non-lazy and lazy schedules for independent tasks 74
3.5 Solver time (s) vs number of preemptions for 8 pipelines, 4 resources, and tightness=1.3 75
3.6 Optimizer time vs makespan in the Cholesky matrix factorization use case (4x4 and 5x5 tiles on two heterogeneous resources) 75
3.7 Platooning application dataflow graph with a rate parameter X 76
3.8 APG of the SDF graph (Figure 3.7 with X=3) 77
3.9 From left to right: FFT task graphs for 4, 8 and 16 inputs 79
3.10 Block diagram of the Exynos 5422 Processor (source: reference manual) 81
3.11 3x3 tiled Cholesky factorization task graphs 82
3.12 In Kernel Switching big.LITTLE configurations 83
3.13 Variability of the BLAS kernels 84
3.14 Pareto front for the 3x3 algorithm 87
3.15 4x4 tiled Cholesky schedules on 4+4 big.LITTLE 88

B.1 4x4 tiled cholesky factorization task graph 97
B.2 5x5 tiled cholesky factorization task graph 98
B.3 Radar application Gantt diagrams: static (left) vs dynamic (right) agent models . . . 99
B.4 Radar application Gantt diagrams dynamic agent models: Bus vs Crossbar interconnect models 100

List of Tables

1.1 Architecture provisions for the tiled Cholesky application (10^6 cycles) 42

2.1 Measured voltages and instant power values for A7 and A15 cores on the Odroid platform 58

3.1 CP solver optimization: *time(s), memory(MB) problem performance when raising the breadth of the graph* 77
3.2 CP solver optimization problem performance when raising the depth of the graph . . . 77
3.3 Fast Fourier Transform CP resolution time at different levels of the recursion 79
3.4 Costs of the BLAS kernels (10^6 cycles) 84
3.5 Results obtained with the modeling and solving (time in milliseconds) 85
3.6 Predicted vs Traced execution time for the tiled Cholesky factorization 86
3.7 Solver time vs number of cores and number of tiles 87

Glossary

AAA Application Architecture Adequation. 3–5, 14, 15, 18, 32, 36–39, 41, 69, 101

APG Acyclic Precedence Graph. 19, 45, 47–49, 65, 77, 101, 102

BLAS Basic Linear Algebra Subkernels. 42, 81, 83, 84, 86, 102, 103

CAS Computer Algebra System. 35, 36, 71

DAG Direct Acyclic Graph. 33, 42, 79

DPN Dataflow Process Network. 17, 18, 22

DVFS Dynamic Voltage and Frequency Scaling. 29

FFT Fast Fourier Transform. 16, 20, 79, 80, 83

HEFT Heterogeneous Earliest Finish Time. 19, 39

JSON JavaScript Object Notation. 38, 43, 63

KPN Kahn Process Networks. 20, 38

KRG K-periodically Routed Graph. 37, 79

MPPA Massively Parallel Processor Array. 55

MPSoC Multi Processor System on Chip. 28, 41, 62

OPP Operating Performance Point. 29, 56

SANLP Static Affine Nested Loops Programs. 22, 81

SAT boolean SATisfiability problem. 33, 41

SDF Synchronous Data Flow. 4, 18–20, 37, 38, 41, 45, 47–49, 64, 76–79, 101

SDFG Synchronous Data Flow Graph. 19, 20, 48

SMT SAT Modulo Theories. 14, 33, 40, 41, 45, 69, 71, 72, 84, 88, 89

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar. Are static schedules so bad? a case study on cholesky factorization. 2015.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] M. Auguin, L. Capella, F. Cuesta, and E. Gresset. Codef: a system level design space exploration tool. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, volume 2, pages 1145–1148. IEEE, 2001.

[5] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[6] F. Balarin. *Hardware-software co-design of embedded systems: the POLIS approach.* Springer Science & Business Media, 1997.

[7] M. A. Bergach. *Adaptation du calcul de la Transformée de Fourier Rapide sur une architecture mixte CPU/GPU intégrée.* PhD thesis, Université Nice Sophia Antipolis, 2015.

[8] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(2):151–166, 1999.

[9] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, 1996.

[10] E. Blossom. Gnu radio: tools for exploring the radio frequency spectrum. *Linux journal*, 2004(122):4, 2004.

[11] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[12] P. Boulet. *Array-OL revisited, multidimensional intensive signal processing specification.* PhD thesis, INRIA, 2007.

[13] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[14] P. Brucker, S. Knust, A. Schoo, and O. Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 107(2):272–288, 1998.

[15] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. 1994.

[16] G. C. Buttazzo, M. Bertogna, and G. Yao. Limited preemptive scheduling for real-time systems. a survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, 2013.

[17] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24. ACM, 2003.

[18] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens. From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. *Leibniz Transactions on Embedded Systems*, 2(2):01–1, 2015.

[19] T. E. Cheng, Q. Ding, and B. M. Lin. A concise survey of scheduling with time-dependent processing times. *European Journal of Operational Research*, 152(1):1–13, 2004.

[20] A. P. E. Coffman Jr and R. L. Graham. Optimal scheduling for two-processor systems. *Acta informatica*, 1(3):200–213, 1972.

[21] D. R. Cok et al. The smt-libv2 language and tools: A tutorial. *Language c*, pages 2010–2011, 2011.

[22] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(5):511–523, 1971.

[23] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[24] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[25] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, 1990.

[26] B. P. Dave, G. Lakshminarayana, and N. K. Jha. Cosyn: hardware-software co-synthesis of embedded systems. In *Proceedings of the 34th annual Design Automation Conference*, pages 703–708. ACM, 1997.

[27] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *HPEC*, pages 1–6, 2013.

[28] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[29] E. A. de Kock, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: Application modeling for signal processing systems. In *Proceedings of the 37th Annual Design Automation Conference*, pages 402–405. ACM, 2000.

[30] A. C. de Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, 2010.

[31] K. De Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In *International Conference on Parallel Processing and Applied Mathematics*, pages 793–803. Springer, 2013.

[32] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[33] N. Dhanwada, I.-C. Lin, and V. Narayanan. A power estimation methodology for systemc transaction level models. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 142–147. ACM, 2005.

[34] J. J. Dongarra and T. Dunigan. Message-passing performance of various computers. *Concurrency - Practice and Experience*, 9(10):915–926, 1997.

[35] B. Fischer, C. Cech, and H. Muhr. Power modeling and analysis in early design phases. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 197. European Design and Automation Association, 2014.

[36] F. Ghenassia et al. *Transaction-level modeling with SystemC.* Springer, 2005.

[37] B. Goglin, J. Hursey, and J. M. Squyres. netloc: Towards a comprehensive view of the hpc system topology. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 216–225. IEEE, 2014.

[38] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM (JACM)*, 23(4):665–679, 1976.

[39] M. J. Gonzalez Jr. Deterministic processor scheduling. *ACM Computing Surveys (CSUR)*, 9(3):173–204, 1977.

[40] R. Gorcitz, E. Kofman, T. Carle, D. Potop-Butucaru, and R. De Simone. On the scalability of constraint solving for static/off-line real-time scheduling. In *Formal Modeling and Analysis of Timed Systems*, pages 108–123. Springer, 2015.

[41] M. I. Gordon. *Compiler techniques for scalable performance of stream programs on multicore architectures.* PhD thesis, Citeseer, 2010.

[42] T. Goubier, R. Sirdey, S. Louise, and V. David. σc: A programming model and language for embedded manycores. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 385–394. Springer, 2011.

[43] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1979.

[44] A. Greiner, E. Faure, N. Pouillon, and D. Genius. A generic hardware/software communication middleware for streaming applications on shared memory multi processor systems-on-chip. In *Specification & Design Languages, 2009. FDL 2009. Forum on*, pages 1–4. IEEE, 2009.

[45] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Low Power Electronics and Design, 2003. ISLPED'03. Proceedings of the 2003 International Symposium on*, pages 217–222. IEEE, 2003.

[46] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

[47] S. Holmbacka, E. Nogues, M. Pelcat, S. Lafond, and J. Lilius. Energy efficiency and performance management of parallel dataflow applications. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pages 1–8. IEEE, 2014.

[48] G. J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279, 1997.

[49] D. Isovic and G. Fohler. Handling sporadic tasks in off-line scheduled distributed real-time systems. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 60–67. IEEE, 1999.

[50] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. 1976.

[51] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

[52] F. Kereki. What's in the box? interrogate your linux machine's hardware. *Linux Journal*, 2015(260):1, 2015.

[53] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *computer*, 36(12):68–75, 2003.

[54] P. M. Knijnenburg, T. Kisuki, and M. F. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.

[55] E. Kofman, J.-V. Millo, and R. De Simone. Application architecture adequacy through an fft case study. In *JRWRTC2013-7th Junior Researcher Workshop on Real-Time Computing*, 2013.

[56] A. Koudri, D. Aulagnier, D. Vojtisek, P. Soulard, C. Moy, J. Champeau, J. Vidal, and J.-C. Le Lann. Using marte in a co-design methodology. In *MARTE UML profile workshop co-located with DATE'08*, pages 6–pages, 2008.

[57] W. Kruijtzer, P. Van Der Wolf, E. De Kock, J. Stuyt, W. Ecker, A. Mayer, S. Hustin, C. Amerijckx, S. De Paoli, and E. Vaumorin. Industrial ip integration flows based on ip-xact standards. In *2008 Design, Automation and Test in Europe*, pages 32–37. IEEE, 2008.

[58] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[59] H. Lebreton and P. Vivet. Power modeling in systemc at transaction level, application to a dvfs architecture. In *Symposium on VLSI, 2008. ISVLSI'08. IEEE Computer Society Annual*, pages 463–466. IEEE, 2008.

[60] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[61] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[62] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[63] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1-3):259–271, 1990.

[64] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System level design with spade: an m-jpeg case study. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 31–38. IEEE Press, 2001.

[65] J. V. Lima, T. Gautier, V. Danjean, B. Raffin, and N. Maillard. Design and analysis of scheduling strategies for multi-cpu and multi-gpu architectures. *Parallel Computing*, 44:37–52, 2015.

[66] Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *8th International Symposium on Quality Electronic Design (ISQED'07)*, pages 204–209. IEEE, 2007.

[67] J.-V. Millo. *ORDONNANCEMENTS Periodiques DANS LES Reseaux DE PROCESSUS: APPLICATIONA LA CONCEPTION INSENSIBLE AUX LATENCES.* PhD thesis, Texas Instruments, 2010.

[68] J.-V. Millo, E. Kofman, and R. D. Simone. Modeling and analyzing dataflow applications on noc-based many-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3):46, 2015.

[69] A. Moreira. Real-time synthetic aperture radar (sar) processing with a new subaperture approach. *IEEE Transactions on Geoscience and Remote sensing*, 30(4):714–722, 1992.

[70] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[71] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[72] R. R. Muntz and E. G. Coffman Jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the ACM (JACM)*, 17(2):324–338, 1970.

[73] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[74] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of the 45th annual Design Automation Conference*, pages 574–579. ACM, 2008.

[75] M. Payer. Too much pie is bad for performance. 2012.

[76] J. L. Peterson. Petri net theory and the modeling of systems. 1981.

[77] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proceedings of the 2006 GCC Developers Summit*, page 2006. Citeseer, 2006.

[78] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[79] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[80] R. Rao and S. Vrudhula. Performance optimal processor throttling under thermal constraints. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 257–266. ACM, 2007.

[81] I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, 2004.

[82] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, (6):23–33, 2001.

[83] D. C. Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[84] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 2–13. ACM, 2003.

[85] Y. Sorel. Syndex: System-level cad software for optimizing distributed real-time embedded systems. *Journal ERCIM News*, 59(68-69):31, 2004.

[86] S. Stuijk. Predictable mapping of streaming applications on multiprocessors, 2007.

[87] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*, pages 777–782. ACM, 2007.

[88] S. Stuijk, M. Geilen, and T. Basten. Sdf3: Sdf for free. In *ACSD*, volume 6, pages 276–278, 2006.

[89] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[90] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro*, 22(2):25–35, 2002.

[91] P. Tendulkar, P. Poplavko, and O. Maler. Symmetry breaking for multi-criteria mapping and scheduling on multicores. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 228–242. Springer, 2013.

[92] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196. Springer, 2002.

[93] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[94] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A systemc/tlm semantics in promela and its possible applications. In *International SPIN Workshop on Model Checking of Software*, pages 204–222. Springer, 2007.

[95] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229. ACM, 2004.

[96] K. Van Rompaey, I. Bolsens, H. De Man, and D. Verkest. Coware—a design environment for heterogenous hardware/software systems. In *Proceedings of the conference on European design automation*, pages 252–257. IEEE Computer Society Press, 1996.

[97] B. Veltman, B. Lageweg, and J. K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel computing*, 16(2):173–182, 1990.

[98] J. Vidal, F. De Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 226–231. IEEE, 2009.

[99] L. Wang, G. Von Laszewski, J. Dayal, and F. Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 368–377. IEEE, 2010.

[100] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[101] S. J. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

[102] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 156–165. IEEE, 2015.

[103] M. Zhou and F. DiCesare. Parallel and sequential mutual exclusions for petri net modeling of manufacturing systems with shared resources. *IEEE Transactions on Robotics and Automation*, 7(4):515–527, 1991.