Software Compilation Techniques for Heterogeneous Embedded Multi-Core Systems

Rainer Leupers, Miguel Angel Aguilar, Jeronimo Castrillon, and Weihua Sheng

Abstract The increasing demands of modern embedded systems, such as high-performance and energy-efficiency, have motivated the use of heterogeneous multi-core platforms enabled by Multiprocessor System-on-Chips (MPSoCs). To fully exploit the power of these platforms, new tools are needed to address the increasing software complexity to achieve a high productivity. An *MPSoC compiler* is a tool-chain to tackle the problems of application modeling, platform description, software parallelization, software distribution and code generation for an efficient usage of the target platform. This chapter discusses various aspects of compilers for heterogeneous embedded multi-core systems, using the well-established single-core C compiler technology as a baseline for comparison. After a brief introduction to the MPSoC compiler technology, the important ingredients of the compilation process are explained in detail. Finally, a number of case studies from academia and industry are presented to illustrate the concepts discussed in this chapter.

Rainer Leupers
Institute for Communication Technologies and Embedded Systems, RWTH Aachen University,
ICT cubes, Kopernikusstrae 16, 52074 Aachen, Germany
e-mail: leupers@ice.rwth-aachen.de

Miguel Angel Aguilar
Institute for Communication Technologies and Embedded Systems, RWTH Aachen University,
ICT cubes, Kopernikusstrae 16, 52074 Aachen, Germany,
e-mail: aguilar@ice.rwth-aachen.de

Jeronimo Castrillon
Center for Advancing Electronics Dresden, TU Dresden,
Georg-Schumannstr. 7A, 01187 Dresden, Germany,
e-mail: jeronimo.castrillon@tu-dresden.de

Weihua Sheng
Silexica GmbH,
Lichtstr. 25, 50825 Köln, Germany
e-mail: sheng@silexica.com

1 Introduction

1.1 MPSoCs and MPSoC Compilers

The current design trend in embedded systems show that heterogeneous Multiprocessor System-on-Chip (MPSoC) is the most promising way to keep on exploiting the high level of integration provided by the semiconductor technology and, at the same time, matching the constraints imposed by the embedded systems market in terms of performance and power consumption. Looking at today's smartphones, it is clear to see that they are integrated with a great number of functions, such as camera, personal digital assistant applications, voice/data communications and multi-band wireless standards. Moreover, like many other consumer electronic products, many non-functional parameters are evenly critical for their successes in the market, e.g., energy consumption and form factor. All these requirements need the emergence of heterogeneous MPSoC architectures. They usually consist of programmable cores of various types, special hardware accelerators and efficient Networks-on-Chips (NoCs), to execute a large amount of complex software, in order to catch up with the next wave of integration.

Compared to high-performance computing systems in supercomputers and computer clusters, embedded computing systems require a different set of constraints that need to be taken into consideration during the design process:

- *Real-time constraints:* Real-time performance is key to the embedded devices, especially in the signal processing domain, such as wireless and multimedia. Meeting real-time constraints requires not only the hardware being capable of satisfying the demands of high-performance computations, but also the predictable behavior of the running applications.
- *Energy-efficiency:* Most mobile devices are battery powered, therefore, energy-efficiency is one of the most important factors during the system design.
- *Area-efficiency:* How to efficiently use the limited chip area becomes critical, especially for consumer electronics, where portability is a must-to-have.
- *Application Domain:* Unlike in general-purpose computing, embedded products usually target at specific market segments, which in turn ask for the specialization of the system design tailored for specific applications.

With these design criteria, heterogeneous MPSoC architectures are called to outperform the previous single-core or homogeneous solutions. For a detailed discussion on the architectures, the readers are referred to Chapter [15]. MPSoC design methodologies, also referred as *Electronic System-Level* (ESL) tools, are growing in importance to tackle the challenge of exploring the exploding design space brought by the heterogeneity [53]. Many different tools are required for completing a successful MPSoC design, or a series of MPSoC product generations, such as the Texas Instruments Keystone family [73]. The *MPSoC compiler* (or *Multi-Core Compiler*) is one important tool among those, which is the main focus of this chapter.

First of all, what is an *MPSoC Compiler*? The large majority of the current compilers are targeted to single-core, and the design and implementation of special compilers optimized for various core types (RISC, DSP, VLIW, among others) has been well understood and practiced. Now, the trend moving to MPSoCs raises the level of complexity of the compilers targeting these platforms. The problems of application modeling, platform description, software parallelization, software distribution, and code generation for an efficient usage of these platforms, still remain as open issues both in academia and industry [17]. In this chapter, *MPSoC Compiler* is defined as the tool-chain to tackle those problems for a given (pre-)verified MPSoC platform.

It is worth mentioning that this definition of MPSoC compiler is slightly different from the term *software synthesis* as it appears in the hardware-software co-design community [28]. In this context, *software synthesis* emphasizes that starting from a single high-level system specification, the tools perform hardware/software partitioning and automatically synthesize the software part so as to meet the system performance requirements of the specifications. The flow is also called an application-driven "top-down" design flow. In contrast, the MPSoC compiler is used mostly in *platform-based* design, where the semiconductor suppliers evolve the MPSoC designs in generations targeting a specific application domain. The function of an MPSoC compiler is very close to that of a single-core compiler, where the compiler translates the high-level programming language (e.g., C/C++) into the machine binary code. The difference is that an MPSoC compiler needs to perform additional (and more complex) jobs over the single-core one, such as software parallelization and distribution, as the underlying MPSoC platform is by orders of magnitude more complex. Although, software synthesis and MPSoC compilers share some similarities, the major difference is that they exist in the context of different methodologies, thus focusing on different objectives [18].

The rest of the chapter is organized as follows. Section 1.2 briefly introduces the challenges of building MPSoC compilers, using a comparison of an MPSoC compiler to a single-core compiler, followed by Section 2, where detailed discussions are carried out. Finally, Section 3 looks into how the challenges are tackled by presenting case studies of MPSoC compilers from the academia and the industry.

1.2 Challenges of Building MPSoC Compilers

Before the multi-core era, single-core systems have been very successful in creating a comfortable and convenient programming environment for software developers. The success is largely due to the fact that the sequential programming model is very close to the natural way humans think and that it has been taught for decades in basic engineering courses. Also, the compilers of high-level programming languages (e.g., C/C++) for single-core are well studied, which hide nearly all hardware details from the programmers as a holistic tool [34]. User-friendly graphical integrated development environments (IDEs) like Eclipse [1] and debugging tools like `gdb` [2] also contribute to the ecosystem of hardware and software in the single-core era.

The complexity of programming and compiling for MPSoC architectures has greatly increased compared to single-core. The reasons are manifold and the most important ones are as follows. On the one hand, MPSoCs inherently ask for applications being written in parallel programming models so as to efficiently utilize the hardware resources. Parallel programming (or thinking) has been proven to be difficult for programmers, despite years of efforts invested in high-performance computing. On the other hand, the heterogeneity of MPSoC architectures requires the compilation process to be ad-hoc. The programming models for different Processing Elements (PEs) can be different. The granularity of the parallelism might also vary. The compiler tool-chains can originate from different vendors for PEs. All those make MPSoC compilation an extremely sophisticated process, which is most likely not anymore "the holistic compiler" for the end users. Neither the software tool-chains are fully prepared to handle MPSoCs, nor productive multi-core debugging solutions are available. The software tool-chains are not yet fully prepared to well handle MPSoC systems, plus the lack of productive multi-core debugging solutions.

An MPSoC compiler, as the key tool to enable the power of MPSoCs, is known to be difficult to build. A brief list of the fundamental challenges is provided below, with an in-depth discussion in the following Section 2.

1. *Programming Models*: Evidently the transition to parallel programming models impacts the MPSoC compiler fundamentally.
2. *Platform Description*: The traditional single-core compiler requires architecture information, such as the instruction set and latency table in the backend to perform code generation. In contrast, the MPSoC compiler needs another type of platform description including further details, such as information about the PEs and available communication resources. This information is used in multiple phases of the compilation process beyond the backend.
3. *Software Parallelization*: While Instruction-Level Parallelism (ILP) is exploited by single-core compilers, MPSoC compilers focus on a wider variety of forms of parallelism, which are more coarse-grained.
4. *Software Distribution*: An MPSoC compiler distributes coarse-grained tasks (or code blocks), while the single-core compiler performs this at instruction-level.
5. *Code generation*: It is yet another leaping complexity for the MPSoC compiler to be able to generate the final binaries for heterogeneous PEs and the NoC compared to generate the binary for a one-ISA architecture.

2 Foundation Elements of MPSoC Compilers

This section delves into the details of the problems mentioned in the introduction of this chapter. The discussion is based on the general structure of a single-core compiler, shown in Figure 1. The issues that make the tasks of an MPSoC compiler particularly challenging are highlighted, taking the single-core compiler technology as a reference.

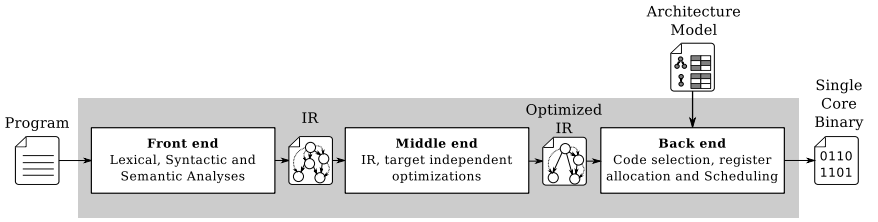


Fig. 1: Coarse View of a Single-core Compiler

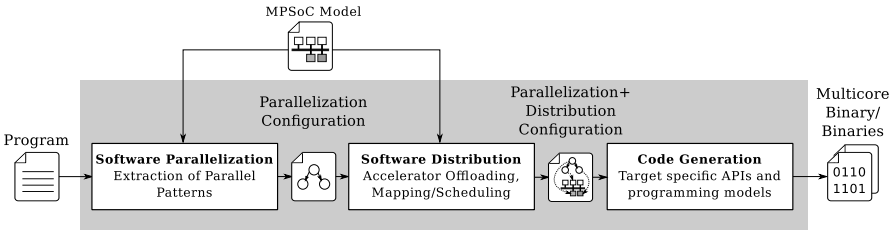


Fig. 2: Coarse View of a MPSoC Compiler

A single-core compiler is typically divided into three phases: the *front end*, the *middle end* and the *back end*. The front end checks for the lexical, syntactic and semantic correctness of the application. Its output is an abstract *Intermediate Representation* (IR) of the application, which is suitable for optimizations and for code generation in the following phases of the compiler. The middle end, sometimes conceptually included within the front end, performs different analyses on the IR. These analyses enable several target-independent optimizations that mainly aim at improving the performance of the posterior generated code. The backend is in charge of the actual code generation and is divided into phases as well. Typical backend steps include *code selection*, *register allocation* and *instruction scheduling*. These steps are machine dependent and therefore require a model of the target architecture.

MPSoC compilers are also divided into phases in order to manage complexity. The overall structure of the single-core compiler (in Figure 1) will suffer some changes though, as Figure 2 shows. In general, an MPSoC compiler is also divided into three phases: *software parallelization*, *software distribution* and *code generation*. Throughout this section more details about these phases will be provided, to help understanding the differences between single-core and MPSoC compilers.

2.1 Programming Models

The main entry for any compiler is a representation of an application using a given programming model, as shown in Figure 1. A programming model is a bridge that provides humans access to the resources of the underlying hardware platform. De-

```
1: function S = fir(coeff, X)
2: S = coeff * X';
```

(a) Matlab

```
1: float coeff[N] = {0.7, ...}
2:
3: float fir (float *ptr) {
4:  int i;
5:  float sum = 0;
6:  for (i=0; i<N; i++)
7:   sum += coeff[i]*(*ptr++);
8:  return sum;
9: }
```

(b) C

```
1: X fract coeff[N] = {0.7, ...}
2:
3: fract fir (Y fract * Y ptr) {
4:   int i;
5:   accum sum = 0;
6:   for (i=0; i<N; i++)
7:    sum += coeff[i] *
8:             (accum)(*ptr++);
9:   return sum;
10: }
```

(c) DSP-C

Fig. 3: FIR implementation on different programming languages

signing such a model is a delicate art, in which hardware details are hidden for the sake of productivity and usually at the cost of performance. In general, the more details remain hidden, the harder the job of the compiler is to close the performance gap. In this sense, a given programming model may reduce the work of the compiler but will never circumvent using one. Figure 3 shows an implementation of an FIR filter using different programming languages representing different programming models. This figure shows an example of the productivity-performance trade-off. On one extreme, the Matlab implementation (Figure 3a) features high simplicity and no information of the underlying platform. The C implementation (Figure 3b) provides more information, having types and the memory model visible to the programmer. On the other extreme, the DSP-C implementation (Figure 3c) has explicit memory bank allocation (through the *memory qualifiers* `X` and `Y`) and dedicated data types (`accum`, `fract`). Programming at this level requires more knowledge and careful thinking, but will probably lead to better performance. Without this explicit information, a traditional C compiler would need to perform complex memory disambiguation analysis in order to place the arrays in separate memory banks.

In [11], the authors classify programming models as being either *hardware-centric*, *application-centric* or *formalism-centric*. Hardware-centric models strive for efficiency and usually require a very experienced programmer (e.g., Intel IXP-C [51]). Application-centric models strive for productivity allowing fast application development cycles (e.g., Matlab [65], LabView [57]), and formalism-centric models strive for safeness due to the fact of being verifiable (e.g., Actors [30]). Practical programming models for embedded MPSoCs cannot pay the performance overhead brought by a pure application-centric approach and will seldom restrict programmability for the sake of *verifiability*. As a consequence, programming models used in industry are typically hardware-centric and provide some means to ease programmability, as will be discussed later in this section.

Orthogonal to the previous classification, programming models can be broadly classified into *sequential* and *parallel ones*. The latter being of particular interest for MPSoC programming and this chapter's readers, though having its users outnum-

bered by the sequential programming community. As a matter of fact, C and C++ are still the top languages in the embedded domain [23], which have underlying sequential semantics. Programmers have been educated for decades to program sequentially. They find it difficult to describe an application in a parallel manner, and when doing so, they introduce a myriad of (from their point of view) unexpected errors. Apart from that, there are millions of lines of sequential legacy code that will not be easily rewritten within a short period of time to make use of the new parallel architectures. Parallel programming models for heterogeneous architectures can be further classified as *host-centric* and *non-host centric*. In the host-centric approach the PEs in the platform have specific roles, either as hosts or accelerators. Here the execution is controlled by the hosts and eventually they offload computationally intensive code blocks to specialized accelerators to improve performance. In contrast, in the non-host centric approach code blocks are assigned to PEs without assuming any specific role for each of them and the control flow is distributed.

Compiling a sequential application, for example written in C, for a simple core is a very mature field. Few people would program an application in assembly language for a single issue embedded RISC processor, such as the ARM7 or the MIPS32. In general, compiler technology has advanced greatly in the single-core domain. Several optimizations have been proposed for superscalar processors [40], DSPs [49], VLIW processors [24] and for exploiting *Single Instruction Multiple Data* (SIMD) architectures [50]. Nonetheless, high performance routines for complex processor architectures with complex memory hierarchies are still hand-optimized and are usually provided by processor vendors as library functions. In the MPSoC era, the optimization space is too vast to allow hand-crafted solutions across different cores. The MPSoC compiler has to help the programmer to optimize the application, possibly taking into account optimized routines for some of the processing elements.

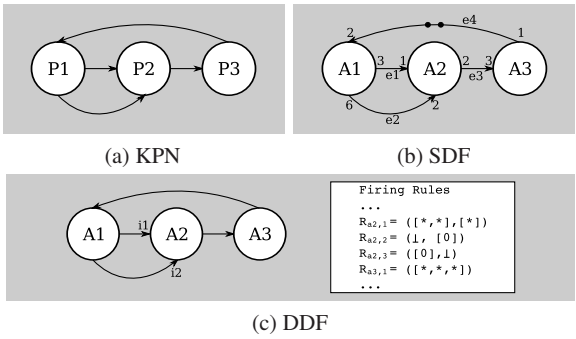
In spite of the efforts invested in classical compiler technology, plain C programming is not likely to be able to leverage the processing power of future MPSoCs. When coding a parallel application in C, the parallelism is hidden due to the inherent sequential semantics of the language and its centralized control flow. Retrieving this parallelism requires complex dataflow and dependence analyses which are usually NP-complete and sometimes even undecidable (see Section 2.3.2). For this reason MPSoC compilers need also to cope with parallel programming models, some of which will be introduced in the following.

2.1.1 Mainstream Parallel Programming Models

There are manifold parallel programming models. Modern parallel programming models are built on top of traditional sequential languages like C or C++ by means of compiler directives, libraries or language extensions. These models are usually classified by the underlying memory architecture that they support; either shared or distributed. They can be further classified by the parallel patterns that they allow to express (see Section 2.3.3). Today a great majority of the mainstream parallel programing models are industry standards, which have a solid tooling support and

are constantly evolving to satisfy the needs of developers and to exploit the new features of modern multi-core platforms. These programming models have their roots in the *High Performance Computing* (HPC) community, however, they have gained acceptance in the embedded domain [5, 41, 71, 74]. Prominent examples of these models are presented in the following:

- **POSIX Threads (Pthreads):** This is a library-based shared memory parallel programming model [69]. Pthreads is a low level approach, as the developer has to explicitly create and destroy threads, partition the workload, map the threads to cores and ensure a proper thread synchronization. The accesses to shared data (critical sections) have to be carefully designed to avoid *data races* and *deadlocks*. The protection to the critical sections can be achieved by means of mutual exclusion (mutex) or semaphores.
- **OpenMP:** This is an industry standard parallel programming model for shared memory systems based on compiler directives [3]. The use of compiler directives implies minimal source code modifications in contrast to Pthreads. Moreover, thread management in OpenMP is performed by a runtime system, which further simplifies the challenging task of multi-core programming. Initially, OpenMP focused on regular loop level parallelism for homogeneous multi-core platforms. However, it was later extended to support both irregular parallelism by means its *tasking model*, and heterogeneous platforms by means of its *accelerator model*. The accelerator model is particular important for the embedded domain, as it enables the designer to exploit all types of cores in heterogeneous MPSoC, including DSPs [71, 74]. Furthermore, recent research efforts have confirmed the applicability of OpenMP in the embedded domain, as it has been demonstrated that it is feasible to use it in real time systems [77].
- **OpenCL:** This is a parallel programming model for heterogeneous systems, which is also an industry standard [70]. OpenCL follows a host-centric approach in which a host device (e.g., CPU) offloads data and computation typically to accelerator devices (e.g., GPUs or DSPs). In this programming model, computations are described as kernels, which are the basic units of execution (e.g., one iteration of a parallel loop). Kernels are written in a language called *OpenCL C*, which is simultaneously a subset and a superset of the C99 standard. In addition, OpenCL offers an API that allows the host to manage data transfers and kernel execution on the target devices. In the embedded domain OpenCL has also gained acceptance, and it is already available for a wide variety of heterogeneous embedded platforms [41, 74].
- **MPI:** This is a parallel programming model for distributed systems based on a library. It relies on the message passing principle, where both point-to-point and collective form communications are supported. MPI can be used in combination with other parallel programming models for shared memory systems, such as OpenMP. While MPI allows to exploit parallelism across nodes in a distributed system, OpenMP allows to exploit parallelism within each node. This approach is usually referred as *hybrid programming* [64]. MPI is currently the *de facto* standard for distributed systems in HPC, and it has been also applied in the embedded domain [5, 74].



(a) KPN (b) SDF

(c) DDF

Fig. 4: Example of Concurrent MoCs (P: Process, A: Actor)

2.1.2 Dataflow Programming Models

Dataflow or streaming *Models of Computation* (MoCs) appear to be one promising choice for describing signal processing applications. In dataflow programming models, an application is represented as a graph. The nodes of this graph (also called processes or actors) perform computation whereas the edges (also called channels) are used to transfer data among nodes. These MoCs originated from theoretical computer science for formally describing a computing system and were initially used to compute bounds on complexity. MoCs were thereafter used in the early 1980s to model VLSI circuits and only in the 1990s started to be utilized for modeling parallel applications. Dataflow programming models based on concurrent MoCs like *Synchronous Dataflow* (SDF) [46] and some extensions (like *Boolean Dataflow* (BDF) [47]) have been deeply studied in [68]. More general dataflow programming models based on *Dynamic Dataflow* (DDF) and *Kahn Process Networks* (KPN) [36] MoC have also been proposed [58, 44] (see also Chapters [26, 12]).

- **KPN Programming Model:** In this programming model, an application is represented as a graph $G = (V,E)$ like the one in Figure 4a. In such a graph, a node $p \in V$ is called *process* and represent computation. The edges represent unbounded FIFO channels for processes communication by means of data items or *tokens*. Processes can only be in one of two states: ready or blocked. The blocked state can only be reached by reading from *only one* empty input channel — *blocking read* semantics. A KPN is said to be determinate: the history of tokens produced on the communication channels is independent of the scheduling.
- **DDF Programming Model:** In this programming model, an application is also represented as a graph $G = (V,E,R)$ with R a family of sets, one set for every node in V. Edges have the same semantics as in the KPN model. Nodes are called *actors* and do not feature the blocking read semantics of KPN. Instead, every actor $a \in V$ has a set of *firing rules* $R_a \in R, R_a = \{R_{a,1}, R_{a,2}, \ldots\}$. A firing rule for an actor $a \in V$ with p inputs is a p-tuple $R_{a,i} = (c_1, \ldots, c_p)$ of conditions. A

condition describes a sequence of tokens that has to be available at the given input FIFO. Parks introduced a notation for such conditions in [61]. The condition $[X_1, X_2, \ldots, X_n]$ requires n tokens with values $X_1, X_2, \ldots, X_n$ to be available at the top of the input FIFO. The conditions $[*]$, $[*,*]$, $[*_{(1)}, \ldots, *_{(m)}]$ require at least 1, 2 and m tokens respectively with arbitrary values to be available at the input. The symbol $\perp$ represents any input sequence, including an empty FIFO. For an actor a to be in the ready state at least one of its firing rules need to be satisfied. An example of a DDF graph is shown in Figure 4c. In this example, the actor a_2 has 3 different firing rules. This actor is ready if there are at least two tokens in input `i1` and at least 1 token in input `i2`, or if the next token on input `i2` or `i1` has value 0. Notice that more than one firing rule can be activated, in this case the dataflow graph is said to be non-determinate.

- **SDF Programming Model:** An SDF can be seen as a simplification of DDF model[1], in which an actor with p inputs has only one firing rule of the form $R_{a,1} = (n_1, \ldots, n_p)$ with $n \in \mathbb{N}$. Additionally, the amount of tokens produced by one execution of an actor on every output is also fixed. An SDF can be defined as a graph $G = (V, E, W)$ where $W = \{w_1, \ldots, w_{|E|}\} \subset \mathbb{N}^3$ associates 3 integer constants $w_e = (p_e, c_e, d_e)$ to every channel $e = (a_1, a_2) \in E$. p_e represents the number of tokens produced by every execution of actor a_1, c_e represents the number of tokens consumed in every execution of actor a_2 and d_e represents the number of tokens (called delays) initially present on edge e. An example of an SDF is shown in Figure 4b with delays represented as dots on the edges. For the SDF in the example, $W = \{(3,1,0),(6,2,0),(2,3,0),(1,2,2)\}$.

Different dataflow models differ in their expressiveness, some being more general, some being more restrictive. By restricting the expressiveness, models possess stronger formal properties (e.g., determinism) which make them more amenable to analysis. For example, since the token consumption and production of an SDF actor are known beforehand, it is possible for a compiler to compute a plausible static schedule for an SDF. For a KPN instead, due to control dependent access to channels, it is impossible to compute a pure static schedule.

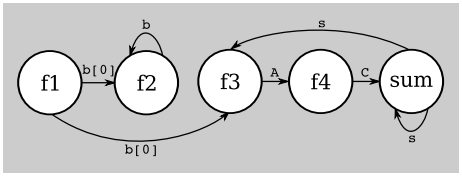
Apart from explicitly exposing parallelism, dataflow programming models became attractive mainly for two reasons. On the one hand, they are well-suited for graphical programming, similar to the block diagrams used to describe signal processing algorithms. On the other hand, some of the underlying MoC's properties facilitate the analysis performed by the tools. For example, channels explicitly expose data dependencies among computing processes/actors, and they have a distributed control flow which is easily mapped to different PEs.

To understand how dataflow models can potentially reduce the compilation effort, an example of an application written in a sequential and in two parallel forms is shown in Figure 5. Let us assume that the KPN parallel specification in Figure 5b represents the desired output of a parallelizing compiler. In order to derive this KPN from the sequential specification in Figure 5a, complex analyses have to be performed. For example, the compiler needs to identify that there is no dependency

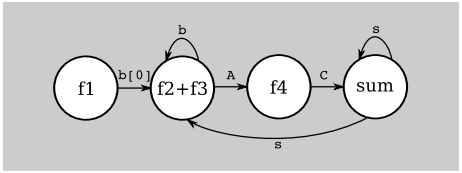
[1] Being more closely related to the so-called *Computation Graphs* [38]

```
1:  int b[N];
2:  int A[4][N];
3:  int c[4];
4:
5:  void foo()
6:  {
7:      int i,j,s;
8:
9:      b[0] = f1();
10:     for (i = 0; i < N; i++)
11:       f2(&b[0], &b[i], &A[i%4][0]);
12:     s = b[0];
13:     b[0] = 0;
14:     for (i = 0; i < 4*16; i++) {
15:         j = i % 4;
16:       f3(s, &A[j][0]);
17:       c[j] = f4(&A[j][0]);
18:       if (j == 3)
19:         s += sum(&c[0]);
20:     }
21: }
```

(a) C Implementation



(b) A “Good” KPN Representation



(c) A “Bad” KPN Representation

Fig. 5: KPN Example

on array `A` among lines `11` and `16` (i.e., between `f2` and `f3`), which is a typical example of dataflow analysis (see Section 2.3.4). Only for a restricted subset of C programs, namely *Static Affine Nested Loop Programs* (SANLP), similar transformations to that shown in Figure 5 have been implemented in [78]. Therefore, starting from a specification already parallel greatly simplifies the work of the compiler.

However, even with a parallel specification at hand, an MPSoC compiler has to be able to look inside the nodes in order to attain higher performance. With the applications becoming more and more complex, a compiler cannot completely rely on the programmer’s knowledge when decomposing the application into blocks. A block diagram can hide lots of parallelism in the interior of the blocks and thus, computing nodes cannot always be considered as *black boxes* but rather as *gray/white boxes* [52]. As an example of this, consider the KPN shown in Figure 5c. Assume that this parallel specification was written by a programmer to represent the same application logic in Figure 5a. This KPN might seem appropriate to a programmer, because the communication is reduced (5 instead of 6 edges). However, if functions `f2` and `f3` are time consuming, running them in parallel could be advantageous. However, in this representation the parallelism remains hidden inside block `f2+f3`.

Summary

Currently, MPSoC compilers should support sequential programming models as input, both because of the great amount of existing sequential legacy code and because of the generations of programmers that were taught to program sequentially. At the same time, the MPSoC compilers need to be aware of the properties of the target parallel programming models, particularly the forms of parallelism that they allow to express, as it will be discussed in Section 2.3.3.

2.2 Platform Description for MPSoC Compilers

After performing optimizations in the middle end, a single-core compiler backend generates code for the target platform based on a model of it. Such a platform model is also required by an MPSoC compiler, but in contrast to a single-core compiler flow, the architecture model may also be used during multiple phases of the compiler and not just by the backend, as Figure 2 shows. For example, if the programming model exposes some hardware details to the user, the front end needs to be able to cope with that and eventually perform consistency checks. Besides, some MPSoC optimizations in the middle end may need some information about the target platform as discussed in Section 2.3. Traditionally an architecture model describes:

- **Available operations**: In form of an abstract description of the *Instruction Set Architecture* (ISA). This information is mainly used by the code selection phase.
- **Available resources**: A list of hardware resources such as registers and functional units (in case of a superscalar or a VLIW). This information is used, for example, by the register allocator and the scheduler.
- **Communication links**: Describe how data can be moved among functional units and register files (e.g., cross paths in a cluster VLIW processor).
- **Timing behavior**: In form of *latency* and *reservation tables*. For each available operation, the latency table tells the compiler how long it takes to generate a result, whereas the reservation table tells the compiler which resources are blocked and for how long. This information is mainly used to compute the schedule.

In the case of an MPSoC, a platform description has to provide similar information but at a different level. Instead of a representation of an ISA, the available operations describe which kinds of processors and hardware accelerators are in the platform. Instead of a list of functional units, the model provides a list of PEs and a description of the memory subsystem. The communication links represent no longer interconnections among functional units and register files, but possibly a complex Network-On-Chip (NoC) that interconnects the PEs among them and with the memory elements. Finally, the timing behavior has to be provided for individual operations (instructions).

Usually, the platform description is a graph representation provided in a given format (usually XML files, see Section 3 for practical examples). Recently, the *Multicore Association* has introduced a standard to specify multi-core platforms called *Software-Hardware Interface for Multi-Many-Core* (SHIM) [56]. This standard allows the abstraction of hardware properties that are key to enable multi-core tools. The SHIM implementation is based on XML files that describe the core types and the platform itself.

One of the main uses of the platform description is to enable the performance estimation of applications. Getting the timing behavior of given code blocks running on a particular MPSoC platform, is a major research topic and a requisite for an MPSoC compiler. Several performance estimation techniques, are applied in order to get specific execution times: *Worst/Best/Average Case Execution Time* (W/B/ACET) [79]. These techniques can be roughly categorized as follows [16]:

- **Analytical:** Analytical or static performance estimation tries to find theoretical bounds to the WCET, without actually executing the code. Using compiler techniques, all possible control paths are analyzed and bounds are computed by using an abstract model of the architecture. This task is particularly difficult in the presence of caches and other non-deterministic architectural features. For such architectures, the WCET might be too pessimistic and thus induces bad decisions (e.g., wrong schedules). There are already some commercial tools available for such purposes, aiT [4] is a good example.
- **Emulation-based:** The simulation time of cycle accurate models can be prohibitively high. Typical simulation speeds range from 1 to 100 KIPS (Kilo Instructions per Second). Therefore, some techniques emulate the timing behavior of the target platform in the host machine without modeling every detail of the processor by means of instrumentation. Source level timing estimation has proven to be useful for simple architectures [39, 33], the accuracy for VLIW or DSP processors is however not satisfactory. The authors in [25] use so-called *virtual back ends* to perform timing estimation by emulating the effects of the compiler back end and thus improving the accuracy of source level timing estimation considerably. With these techniques, simulation speeds of up to 1 GIPS are achievable.
- **Simulation-based:** In this case the execution times are measured on a simulator. Usually cycle accurate virtual platforms are used for this purpose [72]. Virtual platforms allow full system simulation, including complex chip interconnects and memory subsystems. Simulation-based models suffer from the *context-subset* problem, i.e., the measurements depend on the selection of the inputs.
- **Table-based:** This is a performance estimation technique based on source code instrumentation and a table with the costs of elementary processor operations. The cost of executing every elementary operation is based on the cost provided by the architecture model and the execution counts provided by the profiling information resulting from the execution of the instrumented code. This approach allows to identify application hot spots and provides an early idea of the application runtime. However, it is not very accurate, in particular for non-scalar architectures such as VLIW.

Summary

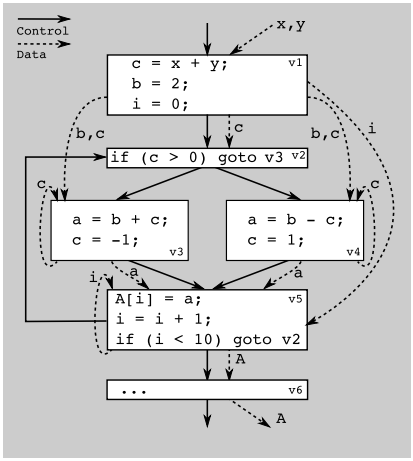
Platform models for MPSoC compilers describe similar features to those of traditional compilers but at a higher level. Processing elements and NoCs take the place of functional units, register files and their interconnections. On an MPSoC compiler, the platform model is no longer restricted to be used on the back end but a subset of it may be used by the front end and the middle end. Out of the information needed to describe the platform, the timing behavior is the most challenging. This timing information is needed for performing successfully software parallelization and distribution, as it will be described in the next sections.

```
1: ...
2: c = x + y;
3: b = 2;
4: for (i = 0; i < 10; i++) {
5:     if (c > 0) {
6:         a = b + c;
7:         c = -1;
8:     } else {
9:         a = b - c;
10:        c = 1;
11:    }
12:    A[i] = a;
13: }
14: ...
```

(a) Sample C Code

```
1: ...
2: c = x + y;
3: c = c > 0 ? c: -c;
4: A[0] = 2 + c;
5: for (i = 1; i < 10; i++)
6:     A[i] = 3;
7: ...
```

(b) Optimized Code



(c) CDFG for 6a

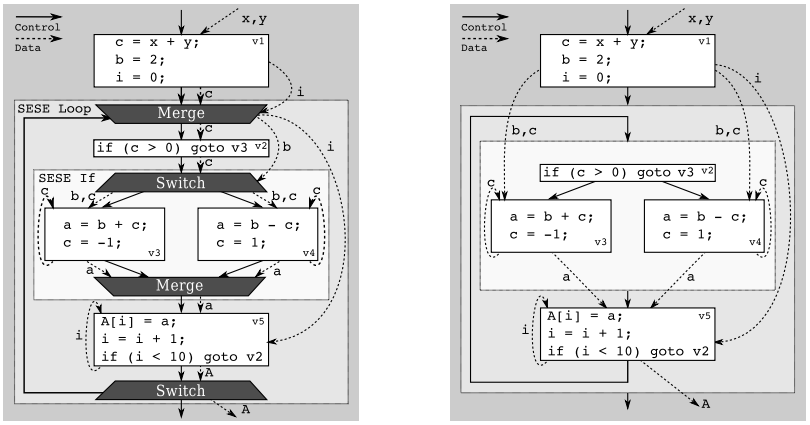
Fig. 6: Example of a CDFG

2.3 Software Parallelization

The software parallelization phase of an MPSoC compiler aims at identifying profitable parallelization opportunities hidden in legacy sequential code. The following sections will give more insights on the main challenges for software parallelization, namely the selection of an intermediate representation, the granularity issue, prominent parallel patterns and the problem of dataflow analysis.

2.3.1 Intermediate Representation (IR)

In a classical compiler, the front end translates application code into an *Intermediate Representation* (IR). Complex constructs of the original high level programming languages are lowered into the IR while keeping machine independence. The IR serves as basis for performing analysis (e.g., control and data flow), upon which many compiler optimizations can be performed. Although there is no *de facto* standard for IRs, most compiler IRs use graph data structures to represent the application. The fundamental analysis units used in traditional compilers are the so-called *Basic Blocks* (BB), where a BB is defined as *a maximal sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end* [10]. A procedure or function is represented as a *Control Flow Graph* whose nodes are BBs and edges represent the control transitions in the program. Data flow is analyzed inside a BB and as a result a *Data Flow Graph* is produced, where nodes represent statements (or instructions) and edges represent data dependencies (or precedence constraints).



(a) DFG (b) HTG

Fig. 7: Hierarchical IRs Examples for the Code in Fig. 6a

With intra-procedural analysis, data dependencies that span across BB borders can be identified. As a result both control and data flow information can be summarized in a *Control Data Flow Graph* (CDFG). A sample CDFG for the code in Figure 6a is shown in Figure 6c. BBs are identified with the literals `v1`,`v2`,...,`v6`. For this code it is easy to identify the data dependencies by simple inspection. Notice however, that the self-cycles because of variable `c` in `v3` and `v4` will never be executed, i.e., the definition of `c` in line `7` will never *reach* line `6`. Moreover, notice that the code in Figure 6a is equivalent to that in Figure 6b. Even for such a small program, a compiler needs to be equipped with powerful analysis to derive such an optimization.

For simple processors, the analysis at the BB *granularity* has been considered the state-of-the-art during the last decades. The instructions inside a BB will always be executed one after another in an in-order processor, and for that reason BBs are very well-suited for exploiting ILP. Already for more complex processors, like VLIW, BBs fall short to leverage the available ILP. *Predicated execution* and *software pipelining* [24] are just some examples of optimizations that cross the BB borders seeking for more parallelism. This quest for parallelism is even more challenging in the case of MPSoC compilers, as they must go beyond ILP. The question of granularity and its implication on parallelism becomes a major issue. The *ideal* granularity depends on the characteristics of the form of parallelism and of the target platform. Therefore, extensions to the CDFG have been proposed to address the granularity issue. One example of this is the *Statement Control Data Flow Graph* (SCDFG) [19] in which nodes are single statements instead of BBs to allow more flexibility. More insights on the granularity issue are provided in Section 2.3.2.

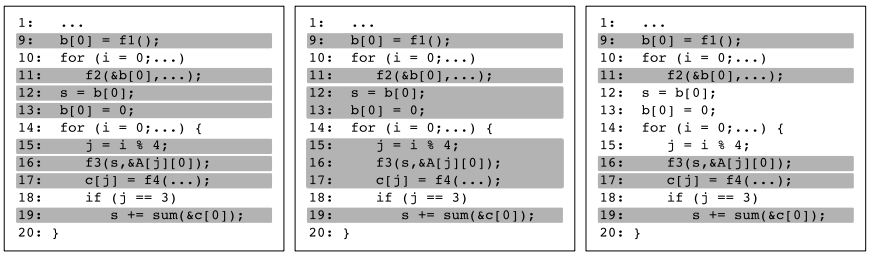
Another major issue for MPSoC compilers is the size of the solution space, which could be prohibitively large even for small applications. This issue has been addressed by introducing the notion of hierarchy in the IR, by also retaining high level information about program structure in the intermediate representation, such as

loops and conditional blocks. This is a powerful property that enables a divide-and-conquer parallelization approach in which code regions can be analyzed in isolation based on their type. The *Dependence Flow Graph* (DFG) [35] and the *Hierarchical Task Graph* (HTG) [63] are examples of representations that incorporate the notion of hierarchy, which have been already used in existing MPSoC compilers [7, 8, 22]. Figure 7a shows an example of a DFG for the code presented in Figure 6a. The DFG incorporates the notion of hierarchy by means of the so-called *Single-Entry Single-Exit* (SESE) regions. A SESE region is a sub-graph of the DFG, which has a unique incoming control edge leading to the region execution, and a unique outgoing control edge that exits the region. Regions can be nested or sequentially ordered and they can be statements, basic blocks, loops or conditional blocks (e.g. if or switch-case constructs). SESE regions related to loops and conditional blocks are enclosed by artificial nodes, namely `switch` and `merge`, as Figure 7a illustrates. A key feature of the artificial nodes is that they allow to re-route data dependencies inside regions where they are relevant. For example, in Figure 7a the data dependencies edges on `b` and `c` are re-routed inside the region `SESE If`, while the data dependency edge on `i` is bypassed, as it is not relevant for that particular region. This feature is useful not only for software parallelization analysis, but also for parallel code generation [9]. An example of a HTG for the code in Figure 6a is presented in Figure 7b. The aim of the HTG is to hide cyclic dependencies by leveraging the explicit hierarchy in a program. In general, the HTG has two main types of nodes: *simple* and *compound*. Single nodes are used to encapsulate a single statement or basic block, while compound nodes introduce hierarchy, as they contain other single or compound nodes. Compound nodes are the counter part of SESE regions in a DFG, as they represent high level program constructs (e.g., loops or conditional blocks). However, the drawback of the HTG is that it has no artificial nodes that allow to re-route data dependencies in and out of the compound nodes, which makes data dependence analysis more challenging.

2.3.2 Granularity and Partitioning

Granularity is one major issue for software parallelization and has a direct impact on the form and degree of parallelism that can be achieved [6]. We define partitioning as the process of analyzing an application and fragmenting it into blocks with a given granularity suitable for parallelism extraction. In this sense, the process of constructing CFGs out of an application as discussed before can be seen as a partitioning. The following are the most intuitive granularities for MPSoC compilers:

- **Statement:** A statement is the smallest standalone entity of a programming language. An application can be broken to the level of statements and the relations among each of them. The statements could be simple expressions, such as arithmetic operations or function calls. This granularity provides the highest degree of freedom to the analysis but could prevent ILP from being exploited at the single-core level. Moreover, the parallelization overhead for such small granularity could be prohibitively large.



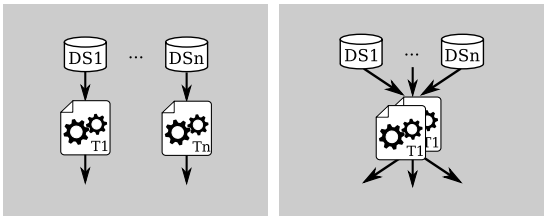
(a) Statement (b) Basic Block (c) Function

Fig. 8: Granularity examples

- **Basic Block:** As already discussed, traditional compilers work on the level of BBs, as they are well suited for ILP. However, in practice BBs could be either too big or too small for coarse-grained parallelism extraction. A BB composed of a sequence of function calls inside a loop would be seen as a single node, and potential parallelism will be therefore hidden. On the other extreme, a couple of small basic blocks divided by simple control constructs could be better handled by a single-core compiler with support for predicated execution.
- **Function:** A function is defined as a subroutine with its own stack. At this level, only function calls are analyzed and the rest of the code is considered as irrelevant. As with BBs, this granularity can be too coarse or too fine-grained depending on the application. It is possible to force a coding style, where parallelism is explicitly written in a way that the behavior is factorized into functions. However, an MPSoC compiler should not make any assumption on the coding style.

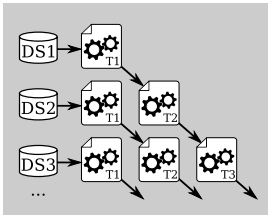
As an example, partitions at different granularity levels for the program introduced in Figure 5a are shown in Figure 8. The partition at statement level is shown in Figure 8a. In this example the statements at lines `12`, `13` and `15` are too light weight. The partition of function `foo` at BB level is shown in Figure 8b. The BB on line `9` is too light weight in comparison to the other BBs, whereas the BB in lines `16-17` may be too coarse. Finally, the partition at function level is shown in Figure 8c. This partition happens to match the KPN derivation introduced in Figure 5b. Whether this granularity is appropriate or not, depends on the amount of data flowing between the functions and the timing behavior of each one of the functions.

As illustrated with the examples, it is not clear what will be the ideal granularity for an MPSoC compiler to work on. Existing research efforts have been directed towards the identification of a suitable granularity for particular parallelism patterns and platforms [7, 20, 22]. The approach is usually based on partitioning an application into *code blocks* of arbitrary granularity by means of heuristics or clustering algorithms, which use the previously described granularities as the starting point (i.e., a code block is built by clustering multiple statements). In the remaining of this chapter we refer to code blocks as statements, BBs, SESE regions, functions or the result of clustering algorithms.

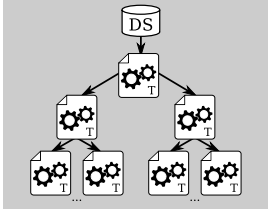


(a) TLP

(b) DLP



(c) PLP



(d) RLP

Fig. 9: Parallelism Patterns

2.3.3 Parallelism Patterns

While a traditional compiler tries to exploit fined-grained ILP, the goal of an MPSoC compiler is to extract coarser parallelism. The most prominent forms of coarse-grained parallelism are illustrated in Figure 9 and described in the following.

- ***Task Level Parallelism* (TLP):** In TLP different tasks can compute in parallel on different data sets as shown in Figure 9a. This form of parallelism is inherent to programming models based on concurrent MoCs (see Section 2.1). Tasks may have dependencies to each other, but once a task has its data ready, it can execute in parallel with the already running tasks in the system. Typically, TLP can be exploited by the parallel execution of independent function calls or loops.
- ***Data Level Parallelism* (DLP):** In DLP the same computational task is carried out on several disjoint *Data Sets*, as illustrated in Figure 9b. This is one of the most scalable forms of parallelism. DLP is typically present in multimedia applications, where a decoding task performs the same operations on different portions of an image or video. Several programming models provide support for DLP, e.g. OpenMP by means of its `for` construct.
- ***Pipeline Level Parallelism* (PLP):** In PLP a computation within a loop is broken into a sequence of tasks called *stages*, as Figure 9c shows. These tasks follow a producer-consumer relationship in which there is a flow of data from the first to the last stage. PLP is a well-suited form of parallelism for streaming applications in the embedded domain, in which there are serially dependent tasks that continuously operate on a flow of data (e.g., audio/video encoding-decoding).

- ***Recursion Level Parallelism* (RLP):** In RLP tasks are created from self-calls in functions that exhibit multiple recursion (i.e., recursive functions that contain two or more self-calls). Applications with multiple recursion typically implement divide-and-conquer algorithms, which recursively break problems into smaller sub-problems that are more simple to solve. A scalable form of nested parallelism can be exploited if the sub-problems are independent (i.e., the recursive call-sites are mutually independent). In RLP each task can further spawn parallel work as nested tasks in subsequent recursive calls, as illustrated in Figure 9d.

Exploiting these kinds of parallelism is a must for an MPSoC compiler, which has to be therefore equipped with powerful flow and dependence analysis capabilities.

2.3.4 Flow and Dependence Analysis

Flow analysis includes both control and data flow. The result of these analyses can be summarized in a CDFG, a DFG or a HTG, as discussed at the beginning of this section. Data flow analysis serves to gather information at different program points, e.g., about available defined variables (*reaching definitions*) or about variables that will be used later in the control flow (*liveness analysis*). As an example, consider the CDFG in Figure 6c in which a reaching definitions analysis is carried out. The analysis tells, for example, that the value of variable `c` in line `5` can come from three different definitions in lines `2`, `7` and `10`.

Data flow analysis deals mostly with scalar variables, like in the previous example, but falls short when analyzing the flow of data when explicit memory accesses are included in the program. In practice, memory accesses are very common through the use of pointers, structures or arrays. Additionally, in the case of loops, data flow analysis only says if a definition reaches a point but does not specify exactly in which iteration the definition is made. The analyses that answer these questions are known as *array analysis*, *loop dependence analysis* or simply *dependence analysis*.

Given two statements $S1$ and $S2$, dependence analysis determines if $S2$ depends on $S1$, i.e., if $S2$ cannot execute before $S1$. If there is no dependency, $S1$ and $S2$ can execute in any order or in parallel. Dependencies are classified into control and data:

- Control Dependency: A statement $S2$ is control dependent on $S1$ ($S1\ \delta^c\ S2$) if whether or not $S2$ is executed depends on $S1$'s execution. In the following example, $S1\ \delta^c\ S2$:

```
S1: if (a > 0) goto L1;
S2: a = b + c;
S3: L1: ...
```

- Data Dependencies:
  - *Read After Write* (RAW, also *true/flow dependency*): There is a RAW dependency between statements $S1$ and $S2$ ($S1\ \delta^f\ S2$) if $S1$ modifies a resource that $S2$ reads thereafter. In the following example, $S1\ \delta^f\ S2$:

```
S1: a = b + c;          S2: d = a + 1;
```

```
S1: for (i = N; i > X; i++) {
S2:     for (j = M; j > Y; j--) {
S3:         A[3*i + 2*j + 2][i + j + 1] = ...;
S4:         ... = A[4*i + j + 1][2*i + j + 1];
S5:     }
S6: }
```

(a) NP Complete

```
S1: while (i) {
S2:     f1(&A[i]);
S3:     ... = A[f2(i)];
S4: }
```

(b) Inter-procedural analysis

```
S1: scanf("%d", &i);
S2: scanf("%d", &j);
S3: A[i] = ...;
S4: ... = A[j];
```

(c) Undecidable

Fig. 10: Examples of Dependence Analysis

– *Write After Write* (WAW, also *output dependency*): There is a WAW dependency between statements $S1$ and $S2$ ($S1\ \delta^o\ S2$) if $S2$ modifies a resource that was previously modified by $S1$. In the following example, $S1\ \delta^o\ S2$:

```
S1: a = b + c;          S2: a = d + 1;
```

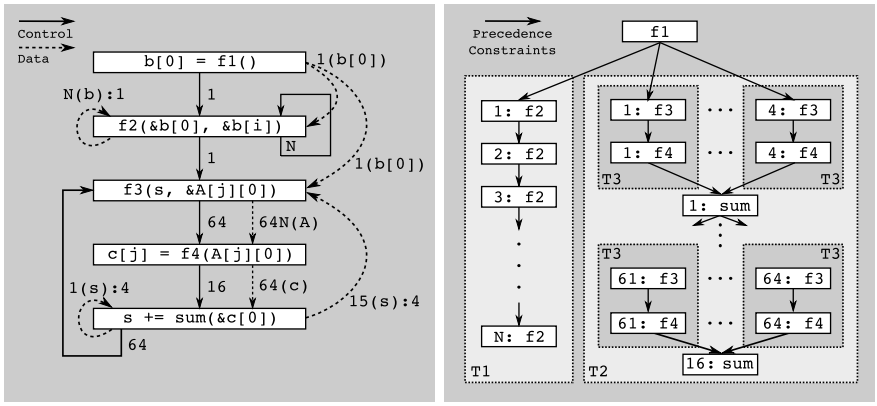
– *Write After Read* (WAR, also *anti-dependency*): There is a WAR dependency between statements $S1$ and $S2$ ($S1\ \delta^a\ S2$) if $S2$ modifies a resource that was previously read by $S1$. In the following example, $S1\ \delta^a\ S2$:

```
S1: d = a + 1;          S2: a = b + c;
```

Obviously, two statements can exhibit different kinds of dependencies simultaneously. Computing these dependencies is one of the most complex tasks inside a compiler, both for single-core and for multi-core systems. For a language like C, the problem of finding all dependencies *statically* is NP complete and in some cases undecidable. The main reason for this being the use of pointers [31] and indexes to data structures that can only be resolved at runtime. Figure 10 shows three sample programs to illustrate the complexity of dependence analysis. In Figure 10a, in order to determine if there is a RAW dependency between $S3$ and $S4$ ($S3\delta^f S4$) across iterations, one has to solve a constrained integer linear system of equations, which is NP complete. For the example, the system of equations is:

$$\begin{aligned} 3x_1 + 2x_2 + 2 &= 4y_1 + y_2 + 1 \\ x_1 + x_2 + 1 &= 2y_1 + y_2 + 1 \end{aligned}$$

subject to $X < x_1, y_1 < N$ and $Y < x_2, y_2 < M$. Notice for example that there is a RAW dependency between iterations $(1,1)$ and $(2,-2)$ on `A[7][3]`. In order to analyze the sample code in Figure 10b, a compiler has to perform inter-procedural analysis to identify if `f1` modifies the contents of `A[i]` and to sort out the potential return values of `f2(i)`. This problem could be potentially undecidable. Finally, the



(a) Summarized CDFG (b) Unrolled Dependencies

Fig. 11: Dependence Analysis on Example in Figure 5a

code in Figure 10c is an extreme case of the previous one, in which it is impossible to know the values of the indexes at compile time. The complexity of dependence analysis motivated the introduction of memory disambiguation at the programming language level, such as the `restrict` keyword in C99 standard [80].

For an MPSoC compiler, the situation is not different. The same kind of analysis has to be performed at the granularity produced by the partitioning step. Array analysis could still be handled by a vectorizing compiler for one of the processors in the platform. The MPSoC compiler has to perform the analysis at a coarser granularity level in which function calls will not be an exception. This is for example the case for the code in Figure 5a. In order to derive KPN representations, like those presented in Figure 5b-c, the compiler needs to be aware of the side effects of all functions. For example, it has to make sure that function `f2` does not modify the array `A`, otherwise there would be a dependency (an additional channel in the KPN) between processes `f2` and `f3` in Figure 5b. The dependence analysis should also provide additional information, for example, that the `sum` function is only executed every four iterations of the loop. This means that every four instances of `f3` followed by `f4` can be executed in parallel. This is illustrated in Figure 11. A summarized version of the CDFG is shown in Figure 11a. In this graph, data edges are annotated with the variable that generates the dependency and, in the case of *loop-carried* dependencies, with the *distance* of the dependency [54]. The distance of a dependency tells after how many iterations a defined value will be actually used. With the dependency information, it is possible to represent the precedence constraints along the execution of the whole program as shown in Figure 11b. In the figure, `n: f` represents the *n*-th execution of function `f`. With this partitioning, it is possible to identify two different forms of parallelism: `T1` and `T2` represent TLP, whereas `T3` represents DLP. This is a good example where flow and dependence analysis help determining a partitioning that exposes coarse grained parallelism.

Due to the complexity of static analyses, multiple research groups started to rely on *Dynamic Data Flow Analysis* DDFA [7, 20, 76]. Unlike static analyses, where dependencies are determined at compile time, DDFA uses traces obtained from profiling runs. This analysis is of course not fully safe and the results need approval from the developer. In general, DDFA is used to obtain a coarse measure of the data flowing among different portions of the application in order to derive plausible partitions and in this way identify DLP, TLP, PLP and/or RLP. Being a profile-based technique, the quality of DDFA depends on a careful selection of the input stimuli. In interactive programming environments, DDFA can provide hints to the programmer about where to perform code modifications to expose more parallelism.

Summary

Traditional compilers work at the basic block granularity which is well suited for ILP. MPSoC compilers in turn need to be equipped with powerful flow analysis techniques, that allow to partition the application into a suitable granularity. This granularity may not match any mainstream granularity and may depend on the parallel pattern. The partitioning step must break the application into code blocks from which coarse level parallelism such as DLP, TLP, PLP or RLP can be extracted.

2.4 Software Distribution

The software distribution phase in an MPSoC compiler aims at deciding *where* and *when* to execute tasks of a parallel application on the target platform. In this chapter we discuss two forms of software distribution: (*i*) *accelerator offloading* in host-centric programming models and (*ii*) *mapping and scheduling* of dataflow MoCs.

2.4.1 Accelerator Offloading

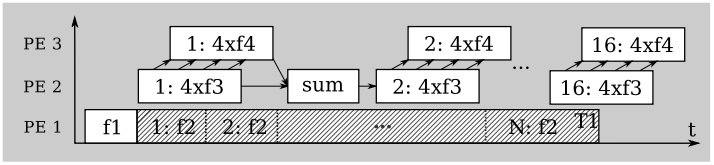
The use of specialized accelerators, such as DSPs and GPUs, has gained popularity due to their high peak performance/watt ratio in contrast to homogeneous multi-cores. However, the heterogeneity introduced by the accelerators makes the programmability of these platforms a complex task. Therefore, multiple host-centric parallel programming models have been proposed to address the challenge of accelerator computing (see Section 2.1.1). These models can be classified as *low-level*, such as OpenCL, or high-level directive-based, such as the OpenMP accelerator model. Despite these efforts to provide a convenient programming model, developers still have to manually specify the code regions to be offloaded and the data to be transferred, while at the same time taking into account that profitable accelerator computing is enabled by abundant DLP and low offloading overhead.

The accelerator offloading analysis in MPSoC compilers is enabled by hierarchical IRs in which applications are decomposed into structured code regions or blocks. An example of these IRs is the DFG introduced in Section 2.3.1, which has been successfully used for accelerator offloading analysis in [9]. The use of hierarchical IRs together with the architectural model of the target platform, enables a divide-and-conquer approach in which every region (typically loops with DLP) can be analyzed in isolation to reason about its potential performance improvement when it is offloaded to a particular accelerator. On the one hand, the region-based analysis allows to compare the performance of a particular region running on a host core with the performance running on an accelerator device. On the other hand, this approach allows to estimate the offloading overhead by looking at the incoming and outgoing data dependencies of the region. Therefore, region-based analysis enables MPSoC compilers to decide whether or not to offload a given region to a particular accelerator, as it provides information about the key aspects for profitable accelerator computing, namely region execution performance and offloading overhead. Finally, the compiler has to be also aware of the desired target programming model to synthesize the appropriate code to offload code regions (see Section 2.5).

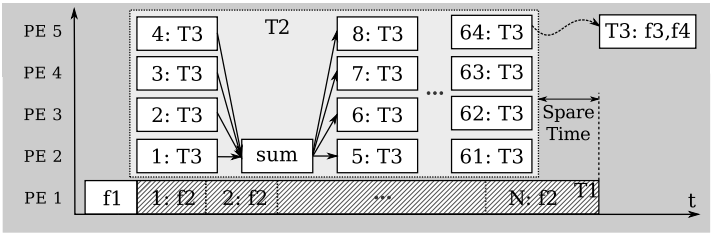
2.4.2 Mapping and Scheduling of Dataflow MoCs

Mapping and scheduling in a traditional compiler is done in the backend provided a description of the architecture. Mapping refers to the process of assigning operations to instructions and functional units (code selection) and variables to registers (register allocation). Scheduling refers to the process of organizing the instructions in a timed sequence. The schedule can be computed statically (for RISC, DSPs and VLIWs) or dynamically at runtime (for Superscalars), whereas the mapping of operations to instructions is always computed statically. The main purpose of mapping and scheduling in single-core compilers had been always to improve performance. Code size is also an important objective for embedded processors (specially VLIW). Only recently, power consumption became an issue. However, the reduction in power consumption with backend techniques does not have a big impact on the overall system power consumption.

In an MPSoC compiler similar operations have to be performed. Mapping, in this context, refers to the process of assigning code blocks to PEs and logical communication links to physical ones. In contrast to the single-core case, mapping can be also dynamic. A code block could be mapped at runtime to different PEs, depending on availability of resources. Scheduling for multi-cores has a similar meaning as for single-core, but instead of scheduling instructions, the compiler has to schedule code blocks. The presence of different application classes, e.g. real time, add complexity to the optimizations in the compiler. Particularly, there is much more room for improving power consumption in an MPSoC; after all, power consumption is one of the MPSoC drivers in the first place.



(a) Partition with PLP, direct implementation of Figure 5b



(b) Full Parallelism Exposed in Figure 11b

Fig. 12: Mapping and Scheduling Examples for Code in Figure 5a

The result of scheduling and mapping is typically represented in form of a *Gantt Chart*, similar to the ones presented in Figure 12. The PEs are represented in the vertical axis and the time in the horizontal axis. Code blocks are located in the plane, according to the mapping and the scheduling information. In Figure 12a functions `f1` and `f2` are mapped to PE 1, the functions `f3` and `sum` are mapped to PE 2 and function `f4` to processor PE 3.

Given that code blocks have a higher time variability than instructions, scheduling can be rarely performed statically. Pure static scheduling requires full knowledge of the timing behavior and is only possible for very predictable architectures and regular computations, like in the case of *systolic arrays* [42]. If it is not possible to obtain a pure static schedule, some kind of synchronization is needed. Different scheduling approaches require different synchronization schemes with different associated performance overhead. In the example, the timing information of task `T3` is not known precisely. Therefore the exact starting time of function `sum` cannot be determined and a synchronization primitive has to be inserted to ensure correctness of the result. In this example, a simple *barrier* is enough in order to ensure that the execution of `T3` in PE 3, PE 4 and PE 5 has finished before executing function `sum`.

Scheduling Approaches

Which scheduling approach to utilize depends on the characteristics of the application and the properties of the underlying MoC used to describe it. Apart from pure static schedules, one can distinguish among the following scheduling approaches:

- **Self-timed Scheduling:** Typical for applications modeled with dataflow MoCs. A self-timed schedule is close to a static one. Once a static schedule is computed, the code blocks are ordered on the corresponding PEs, and synchronization primitives are inserted that ensure the presence of data for the computation. This kind of scheduling is used for SDF applications. For a more detailed discussion the reader is referred to [68].
- **Quasi-static Scheduling:** Used in the case where control paths introduce a predictable time variation. In this approach, unbalanced control paths are balanced and a self-timed schedule is computed. Quasi-static scheduling for dynamically parameterized SDF graphs is explored in [14] (see also Chapter [75]).
- **Dynamic Scheduling:** Used when the timing behavior of the application is difficult to predict and/or when the number of applications is not known in advance (like in the case of general purpose computing). The scheduling overhead is usually higher, but so is also the average utilization of the processors in the platform. There are many dynamic scheduling policies. *Fair queue scheduling* is common in general purpose operating systems (OSs), whereas different flavors of priority based scheduling are typically used in embedded systems with real time constraints, e.g., *Rate Monotonic* (RM) and *Earliest Deadline First* (EDF).
- **Hybrid Scheduling:** Term used to refer to scheduling approaches in which several static or self-timed schedules are computed for a given application at compile time, and are switched dynamically at run-time depending on the *scenario* [27]. This approach is applied to streaming multimedia applications, and allows to adapt at runtime making it possible to save energy [52].

Virtually every MPSoC platform provides support for implementing mapping and scheduling. The support can be provided in software or in hardware and might restrict the available policies that can be implemented. This has to be taken into account by the compiler, which needs to generate/synthesize appropriate code (see Section 2.5).

Computing a Schedule

Independent of which scheduling approach and how this is supported, the MPSoC compiler has to compute a schedule (or several of them). Finding an optimal one in terms of performance is known to be NP-complete even for simple *Directed Acyclic Graphs* (DAGs). Single-core compilers therefore employ heuristics, most of them being derived from the classical *List Scheduling* algorithm [32]. Computing a schedule for multi-core platforms is by no means simpler. The requirements and characteristics of the schedule depend on the underlying MoC with which the application was modeled. In this chapter we distinguish between application modeled with centralized and distributed control flow.

Centralized control flow:

Single-core compilers deal with centralized control flow, i.e., instructions are placed in memory and a central entity dictates which instructions to execute next, e.g., the *program counter* generator. The scheduler in a traditional single-core compiler leaves the control decisions out of the analysis and focus on scheduling instructions inside a BB. Since the control flow inside a BB is linear, there are no circular data dependencies and the data dependence graph is therefore acyclic. The resulting DAG is typically scheduled with a variant of the list scheduling algorithm.

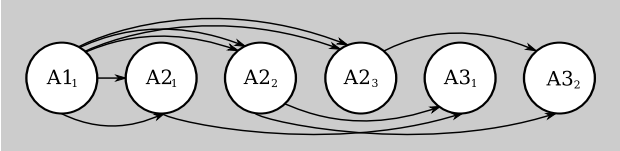
In order to achieve a higher level of parallelism, single-core compilers apply different techniques that go beyond BBs. Typical examples of this techniques include *loop unrolling* and *software pipelining* [45]. An extreme example of loop unrolling was introduced in the previous section, where the dependence graph in Figure 11a was completely unrolled in Figure 11b. Note that the graph in Figure 11b is acyclic and could be scheduled with the list scheduling algorithm. The results of list scheduling with 5 resources would look similar to the scheduling traces in Figure 12b.

In principle, the same scheduling approach can be used for multi-core. However, since every core in a MPSoC has its own control flow, a mechanism has to be implemented to transfer control. In the example in Figure 12b, some core has the control code for the loop in line `14` of Figure 5 and activates the four parallel tasks `T3`. There are several ways of handling this distribution of control. Parallel programming models like Pthreads and OpenMP offer source level primitives to implement *forks* and *joins*. Some academic research platforms offer dedicated instructions to send so-called *control tokens* among processors [81].

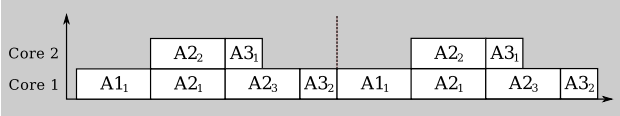
Distributed control flow:

Parallel programming models based on concurrent MoCs like the ones discussed in Section 2.1.2 feature distributed control flow. For applications represented in this way, the issue of synchronization is greatly simplified and can be added to the logic of the channel implementation. Simple applications represented as acyclic task precedence graphs with predictable timing behavior can be scheduled with a list scheduling algorithm or with one of many other available algorithms for DAGs. For a survey on DAG scheduling algorithms the reader is referred to [43]. Applications, where precedence constraints are not explicit in the programming model and where communication can be control dependent, e.g., KPNs are usually scheduled dynamically. Finally, for applications represented as SDF, a self-timed schedule can be easily computed.

- **KPN scheduling:** KPNs are usually scheduled dynamically. There are two major ways of scheduling a KPN: *data* and *demand* driven. In data driven scheduling, every process in the KPN with available data at its input is in the ready state. A dynamic scheduler then decides which process gets executed on which processor at runtime. A demand driven scheduler first schedules processes with no output



(a) Derived DAG with $\mathbf{r} = 1\ 3\ 2^T$



(b) Possible Schedule on Two Cores

Fig. 13: Example of SDF scheduling, for SDF in Figure 4b

channels. These processes execute until a read blocks in one of the input channels. The scheduler triggers then only the processes from which data has been requested (*demanded*). This process continues recursively. For further details the reader is referred to [61].

- **SDF scheduling:** As mentioned before, SDFs are usually scheduled using a self-timed schedule, which requires a static schedule to be computed in the first place. There are two major types of schedules: *blocked* and *non-blocked schedules*. In the former, a schedule for one cycle is computed and is repeated without overlapping, whereas in the latter, the execution of different iterations of the graph are allowed to overlap. For computing a blocked schedule, a *complete cycle* in the SDF has to be determined. A complete cycle is a sequence of actor firings that brings the SDF to its initial state. Finding a complete cycles requires that (*i*) enough initial tokens are provided in the edges and (*ii*) there is a non trivial solution for the system of equations $\Gamma \cdot \mathbf{r} = 0$, where $[\Gamma_{ij}] = p_{ij} - c_{ij}$, and p_{ij} c_{ij} are the number of tokens that actor i produces to and consumes from channel j respectively. In the literature, $\mathbf{r}$ is called *repetition vector* and Γ *topology matrix*. As an example, consider the SDF in Figure 4b. This SDF has a topology matrix:

$$\Gamma = \begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix}$$

and a repetition vector is $\mathbf{r} = [1\ 3\ 2]^T$. By *unfolding* the SDF according to its repetition vector and removing the feedback edges (those with delay tokens) one obtains the DAG shown in Figure 13a with a possible schedule on two cores sketched in Figure 13b. Using this procedure, the problem of scheduling an SDF is turned into DAG scheduling, and once again, one of the many heuristics for DAGs can be used. See Chapter [29] for further details.

For general application models and with the aim to obtain better results than with human-designed heuristics, several optimization methods are used. *Integer Linear Programming* is used in [59] and a combination of Integer Linear Programming and *Constraint Programming* (CP) is employed in [13]. *Genetic Algorithms* have also been used for this purpose, see Chapter [12]. Apart from scheduling and mapping code blocks and communication, a compiler also needs to map data. Data locality is already an issue for single-core systems with complex memory architectures: caches and *Scratch Pad Memories* (SPM). In multi-core systems, maximizing data locality and minimizing *false sharing* is an even bigger challenge [37].

Summary

Software distribution in the form of accelerator offloading and mapping and scheduling is one of the major challenges of MPSoC compilers. Different application constraints lead to new optimization objectives. Besides, different programming models with their underlying MoC allow different scheduling approaches. Most of these techniques work under the premise of accurate performance estimation (Section 2.2) which is by itself a hard problem. In addition, due to the high heterogeneity of signal processing multi-core systems, mapping of data represents a bigger challenge than in single-core systems.

2.5 Code Generation

The code generation phase of an MPSoC compiler is ad-hoc due to the heterogeneity of MPSoCs. To name a few examples: the cores are heterogeneous where the programming models may differ, the communications networks (and thus the APIs) are heterogeneous, and the OS-service libraries implementations can vary from one to another. After the software parallelization and the distribution phases, the code generation of an MPSoC compiler acts like a *meta-compiler* on top of multiple off-the-shelf compilers of the target MPSoC, to coordinate the compilation process. In this process, the code generator first performs a source-to-source transformation of the input application (which is either a sequential code or an abstract dataflow MoC), into a concrete parallel implementation, which is then further compiled with the tool-chain (including assemblers, compilers and linkers) of the target MPSoC. This tool-chain in turn can enable its own optimization features to further improve the code quality.

During the source-to-source transformation multiple steps take place, such as implementation of the parallel patterns according to the programming model, assignment of code blocks to cores, generation of the code for communication and scheduling, linking with the low-level libraries, among others. The complexity of the code generation process depends on the parallel programming model. For ex-

ample, the code transformations for OpenMP are minimal, since it only implies inserting simple compiler directives. In contrast, other parallel programming models, such as Pthreads or OpenCL require heavy program transformations. For example, in OpenCL the kernels have to be extracted and the host code managing kernel execution and data transfers has to be added. Similarly, for abstract dataflow MoCs, the code generator has to make use of target specific OS APIs and libraries to create concrete implementations of actors/processes and FIFO channels.

In an MPSoC, PEs will communicate with each other using the NoC, which requires communication/synchronization primitives (e.g., semaphores, message passing) correctly set in place of the code blocks that the MPSoC compiler distributes to the PEs. Again, due to the heterogeneous nature of the underlying architecture, the same communication link may look very different in the implementation, e.g., when the sending/receiving points are in different PEs. Embedded applications often need to be implemented in a portable fashion for the sake of software re-use. *Abstraction* of the communication functions to a higher level into the programming model is widely practiced, though it is still very ad-hoc and platform-specific. Recently, the Multicore Association has published the first draft of Multicore Communications API (MCAPI), which is a message-passing API to capture the basic elements of communication and synchronization that are required for closely distributed embedded systems [55]. This might have been a good first step in this area.

As discussed in Section 2.4.2, the scheduling decision is a key factor in the MPSoC compiler, especially in dataflow MoCs for embedded computing where real-time constraints have to be met. No matter which scheduling policy is determined for the final design, the functionality has to be implemented, in hardware, or software, or in a hybrid manner. A common approach is to use an off-the-shelf OS, often an RTOS, to enable the scheduling. There are many commercial solutions available such as QNX and WindRiver. The scheduler implementation in hardware is not uncommon for embedded devices, as software solutions may lead to larger overhead, which is not acceptable for RT-constrained embedded systems. Industry and academia have delivered promising results in this area, though more successful stories are still needed to justify this approach. A hybrid solution is a mixture, where some acceleration for the scheduler is implemented in hardware while flexibility is provided by software programmability, therefore customizing a trade-off between efficiency and flexibility. If the scheduling is not helped by e.g., an OS or a hardware scheduler, the code generation phase needs to generate or synthesize the scheduler e.g., [21] and [44].

Summary

Code generation is a complicated process, where many efforts are made to hide the compilation complexity via layered SW stacks and APIs. Heterogeneity will cause ad-hoc tool-chains to exist for a long time. The complexity of the code generation process depends of the parallel programming model.

3 Case Studies

As discussed in Section 2, the complexity of MPSoC compilers grows rapidly compared to single-core compilers. Nowadays, MPSoC compiler constructions for different industrial platforms and academic prototypes are still very much ad-hoc. This section surveys some prominent examples to show the readers how concrete implementations address the various challenges of MPSoC compilers.

3.1 Academic Research

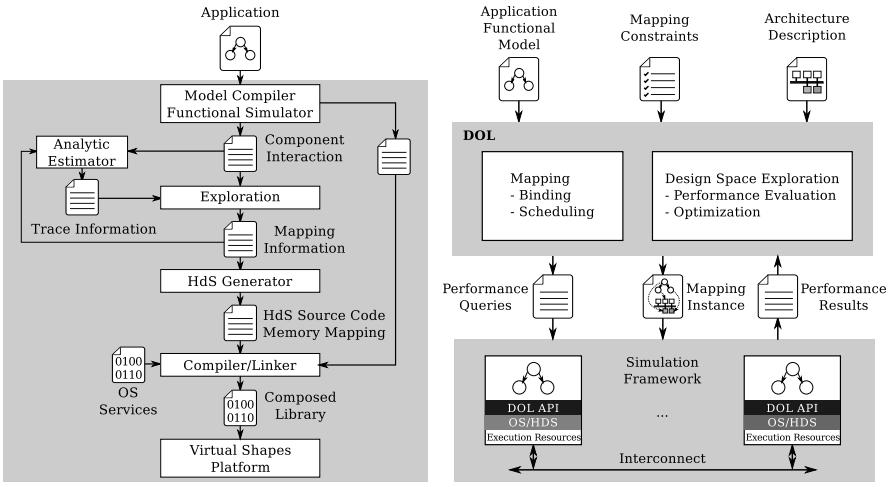
In academia, vast research efforts have been recently directed towards MPSoC compiler technologies. Since the topic is very heterogeneous in nature, it has caught the attention of different research communities, such as real-time computing, compiler optimization, parallelization and fast simulation. A considerable amount of efforts have been invested in areas, such as MoCs, automatic parallelization and virtual simulation platforms. Compared to their counterparts in industry, the academic researchers focus mostly on the *upstream* of the MPSoC compilation flow, e.g., using MoCs to model applications, automatic task-to-processor mapping, early system performance estimation and holistic construction of MPSoC compilers.

3.1.1 SHAPES

SHAPES [60] is a European Union FP6 Integrated Project whose objective is to develop a prototype of a *tiled* scalable hardware and software architecture for embedded applications featuring inherent parallelism. The major SHAPES building block, the RISC-DSP tile (RDT), is composed of an Atmel Magic VLIW floating-point DSP, an ARM9 RISC processor, on-chip memory, and a network interface for on- and off-chip communication. On the basis of RDTs and interconnect components, the architecture can be easily scaled to meet the computational requirements.

The SHAPES framework is shown in Figure 14a. The starting point is the Model-driven compiler/Functional simulator, which takes an application specification in the form of process networks as input. High-level mapping exploration involves the trace information from the Virtual Shapes Platform (VSP) and the performance results from the Analytical Estimator, based on multi-objective optimization considering throughput, delay, predictability and efficiency. With the mapping information, the Hardware dependent Software (HdS) phase then generates the necessary dedicated communication and synchronization primitives, together with OS services.

The central part of the SHAPES software environment is the Distributed Operation Layer (DOL) framework [67]. The DOL structure and interactions with other tools and elements are shown in the Figure 14b. DOL mainly provides the MPSoC software developers two main services: system level performance analysis and process-to-processor mapping exploration.



(a) Software Development Environment (b) DOL Framework

Fig. 14: SHAPES Design Flow

- *DOL Programming Model:* DOL uses process networks as its programming model — the structure of the application is specified in an XML format consisting of processes, software channels and connections, while the application functionality is specified in C/C++ and process communications are performed by the DOL APIs, e.g., `DOL_read()` and `DOL_write()`. DOL uses a special *iterator* element to allow the user to instantiate several processes of the same type. For the process functionality in C/C++, a set of coding rules needs to be followed. In each process there must be an `init` and a `fire` procedure. The `init` procedure allocates and initializes data, which is called once during the application initialization. The `fire` procedure is called repeatedly afterwards.
- *Architecture Description:* DOL aims at mapping, therefore its architecture description abstracts away several details of the underlying platform. The XML format contains three types of information: *structural elements* such as processors/memories, *performance data* such as bus throughputs, and *parameters* such as memory sizes.
- *Mapping Exploration:* DOL mapping includes 2 phases: performance evaluation and optimization. Performance evaluation collects the data from both analytical performance evaluation and the simulation. The designer defines the optimization objectives and DOL uses evolutionary algorithms to generate the mapping.

With the mapping descriptor the HdS layer generates hardware dependent implementation codes and makefiles. Thereafter, the application can be compiled and linked against communication libraries and OS services. The final binary can be executed on the VSP or on the SHAPES hardware prototype.

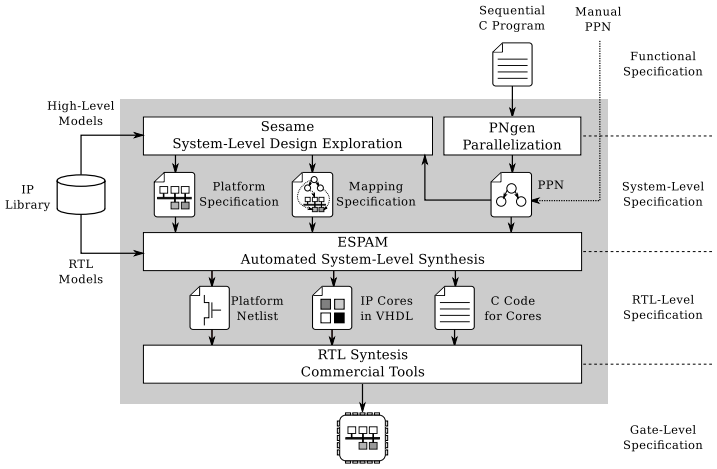


Fig. 15: Daedalus Framework

3.1.2 Daedalus

Daedalus framework [58] is a tool-flow developed at Leiden University for automated design, programming and implementation of MPSoCs starting at a high level of abstraction. The Daedalus design-flow is shown in Figure 15. It consists of three key tools, PNgen tool, Sesame (Simulation of Embedded System Architectures for Multilevel Exploration) and ESPAM (Embedded System-level Platform synthesis and Application Modeling), which work together to offer the designers a single environment for rapid system-level architectural exploration and automated programming and prototyping of multimedia MPSoC architectures. The PNgen tool automatically transforms the sequential application into a parallel specification in the form of *Polyhedral Process Networks* (PPNs), which are a subset of KPNs. The code that can be expressed in PPNs should be analyzable in the *polyhedral model* [48], which implies that the input sequential code is restricted to *Static Affine Nested Loop Programs* (SANLP). Then, the PPNs are used by Sesame modeling and simulation tool to perform a system-level *design space exploration* (DSE), where the performance of multiple mappings, HW/SW partitions and target platform architectures is quickly evaluated using high-level models from the IP library. Finally, the most promising mapping and platform specifications resulting from the DSE, together with the application specification (PPN) are the inputs to the ESPAM synthesis tool. The ESPAM tool uses these inputs along with the low-level RTL models from the IP library to automatically generate synthesizable VHDL code that implements the hardware architecture. It also generates, from the XML specification of the application, the C code for those processes that are mapped on to programmable cores, including the code for synchronization of the communication between the processors. Furthermore, commercial synthesis tools and the component compilers can be used to process the outputs for fast hardware/software prototyping.

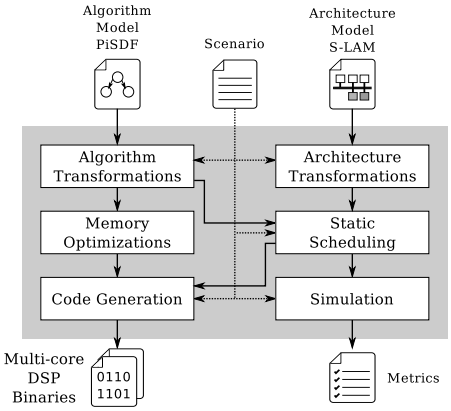


Fig. 16: PREESM Framework

3.1.3 PREESM

The *Parallel and Real-time Embedded Executives Scheduling Method* (PREESM) is a framework for rapid prototyping and code generation, whose primary target is multi-core DSP platforms [62]. PREESM is developed at the Institute of Electronics and Telecommunications-Rennes (IETR) in collaboration with Texas Instruments. The PREESM framework is shown in Figure 16. It takes as input an algorithm specification, an architectural model and a scenario that links the algorithm with the architecture. The *Parameterized and Interfaced Synchronous Dataflow* (PiSDF) is the MoC used here for the algorithm specification. PiSDF is an extension of SDF in which the production and consumption rates of the actors and the FIFO delays can be parameterized. The System-Level Architecture Model (S-LAM) describes the target platform as a graph in which the processing elements offer the processing capabilities for the actors and the communication elements offer the FIFO communication capabilities. The algorithm and architecture models are then transformed to enable scheduling and memory optimizations. On the one hand, the scheduling optimization aims at providing a static schedule that is deadlock-free. On the other hand, the memory optimization aims at reducing the memory requirements by allowing the re-utilization of memory for the FIFOs during code generation. Finally, the PREESM simulation facilities allow to asses the system performance by providing a gantt chart of the parallel execution of the algorithm, speedup estimates and memory requirements. Finally, the code generation stage emits the software for the selected multi-core DSP platform, which includes the necessary instructions for proper inter-core communication, cache management and synchronization. PREESM has been successfully evaluated in commercial multi-core DSP platforms, such as the ones from the Keystone family from Texas Instruments described in Section 3.2.1.

3.2 Industrial Case Studies

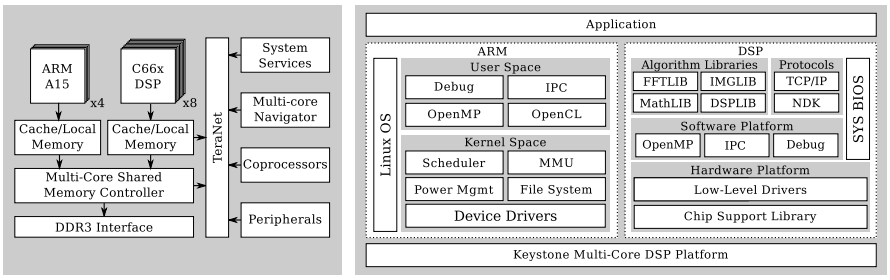
Several large semiconductor companies have already a few mature product lines aiming at different segments of the market due to the application-specific nature of the embedded devices. The stringent time-to-market window calls for the necessity to adopt platform-based MPSoC design methodology. That is, a new generation of an MPSoC architecture is based on a previous successful model with some evolutionary improvements. Compared to their counterparts in academia, the MPSoC software architects in industry focus more on the software tools re-use (considering the huge amount of certified code), providing abstractions and conveniences to the programmers for software development and efficient code-generation.

3.2.1 TI Keystone Multi-Core DSP Platform

The Keystone is a family of MPSoCs from Texas Instruments for high performance systems [73], which integrates RISC and DSP cores together with application specific co-processors and peripherals. The application domains of the Keystone platforms include high performance computing, wireless communications, networking, and audio/video processing. The Keystone architecture provides a high internal bandwidth by allowing non-blocking accesses to the processing cores, co-processors and peripherals. This is enabled by four main components: Multicore Navigator, TeraNet, Multicore Shared Memory Controller (MSMC) and HyperLink. The Multicore Navigator is a hardware controller for packet-based communication. Typical use cases are: message exchange or data transfer among cores, and data transfers between cores and co-processors or peripherals. The TeraNet is a low latency switch fabric that allows the movement of the Multicore Navigator packets among the main components within the Keystone platforms. The Multicore Shared Memory Controller allows to access the shared memory without using the TeraNet, which avoids interference with the packet movement. Finally, the HyperLink allows to interconnect multiple Keystone MPSoC.

Currently, there are two generations of the Keystone family. In the first generation, only DSPs were integrated as programmable cores. The architecture of the DSPs used in the Keystone platforms is called *C66x*. One interesting feature of the C66x cores is that they have both fixed-point and floating-point computation capabilities. In the second generation, the major enhancement is the integration of Cortex-A15 multi-core processors. In addition, the storage and bandwidth capacities of the main components were increased. Figure 17a shows the 66AK2H12 devices of the Keystone II family. This device offers a quad-core Cortex-A15 processor and eight C66x DSP cores, along with the main components of the Keystone family.

Figure 17b illustrates the software stack that TI provides for the Keystone platforms [74]. This software stack is divided into two coordinated sub-stacks, one for the ARM cores and another one for DSP cores. TI promotes the philosophy of *abstractions* among the software layers to hide just enough details for the developers at different roles/layers.



(a) 66AK2H12 Keystone II Device (b) Keystone Software Stack

Fig. 17: TI Keystone Multi-Core DSP Platform

- OS Level: At the OS level the choice on the ARM side is Linux and on the DSP side is the TI-RTOS kernel (formerly known as SYS BIOS, which was the successor of DSP/BIOS) [74]. The TI-RTOS is optimized for real-time multi-tasking and scheduling. Along with the OS, low-level device drivers are provided to enable the use of hardware components in the Keystone platforms by higher software layers.
- Software Platform Level: The support for multi-core programming is at software platform level, including the TI IPC package [74] for inter-core communication and the support for industry standards, such as OpenMP and OpenCL. At this level there are also packages that enable tools for debugging, instrumentation and multi-core performance.
- Algorithm Level: Algorithms/codecs are usually allocated onto the DSP due to its computation power. At this level TI provides optimized libraries for multiple domains from general purpose math and signal processing libraries (e.g., DSPLIB and MATHLIB) to application specific libraries (e.g., IMGLIB and FFTLIB) [74].
- Application Level: The application developer uses the software layers introduced earlier to build the final system. Third-party tools that provide valuable add-ons such as GUI or streaming frameworks can be ported here.

The *abstractions* among the layers are realized by the standardized interfaces. Therefore, different teams can work in different domains at the same time thus boosting the productivity. Moreover, this also enables the possibility of third-parties participating in the TI software stack to provide valuable/commercial solutions, e.g., multi-core development tools and application-level GUI frameworks.

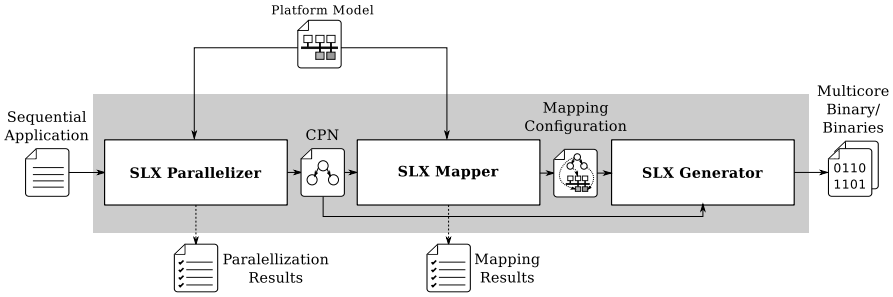


Fig. 18: SLX Tool Suite

3.2.2 Silexica: SLX Tool Suite

Silexica (SLX) [66] is a provider of software automation tools that addresses the increasingly complex task of multi-core programming in a variety of application domains, such as embedded vision, automotive and wireless telecommunications. Silexica is a spin-off of the Institute for Communication Technologies and Embedded Systems (ICE) at RWTH Aachen University. Its core technology is the *SLX Tool Suite* shown in Figure 18. This tool suite has its roots in the academic project called *MPSoC Application Programming Studio* (MAPS), which started over a decade ago at ICE. The SLX Tool Suite is an excellent example of the adoption by the industry of the MPSoC compiler technologies described in this chapter, since it addresses the challenges of application modeling, platform description, software parallelization, software distribution, and code generation.

The SLX Tool Suite is composed of three main tools: *SLX Parallelizer*, *SLX Mapper* and *SLX Generator*. For an effective target-specific analysis, this tool suite uses fast and accurate software performance estimation technologies and an architectural model of the target platform. First, the SLX Parallelizer helps to migrate legacy C/C++ applications into the multi-core domain by identifying profitable parallelization opportunities. This parallelizer focuses on parallel patterns, such as DLP, PLP and TLP (see Section 2.3.3). As an output it provides source level information, which helps developers to understand the parallelization opportunities and its potential. In addition, the parallelized application can be exported using industry standards, such as OpenMP, or as the SLX specification called *C for Process Networks* (CPN). CPN is a language extension that allows to specify applications as dataflow MoCs (e.g KPNs). The CPN specification can be either derived from the SLX Parallelizer analysis or manually by the developer. The SLX Mapper performs the task of software distribution by analyzing the computation and communication behavior of the CPN specification, to automatically distribute the processes on the platform cores and the FIFO channels on the platform interconnects. Finally, the SLX Generator is a source-to-source translation tool that takes both the CPN and the mapping specification generated by the SLX Mapper, to emit architecture-aware code, which is further compiled with the native tool-chain of the target platform.

4 Summary

In this chapter is presented an overview of the challenges for building MPSoC compilers and described some of the techniques, both established and emerging, that are being used to leverage the computing power of current and yet to come MPSoC platforms. The chapter concluded with selected academic and industrial examples that show how the concepts are applied to real systems.

It can be observed how new programming models are being proposed that change the requirements of the MPSoC compiler. It was discussed that, independent of the programming model, an MPSoC compiler has to find a suitable granularity to expose parallelism beyond the instruction level (ILP), demanding advanced analysis of the data and control flows. Software distribution is one of the most complex tasks of the MPSoC compiler and can only be achieved successfully with accurate performance estimation or simulation. Most of these analyses are target-specific, hence the MPSoC itself needs to be abstracted and fed to the compiler. With this information, the compiler can tune the different optimizations to the target MPSoC and finally generate executable code.

The whole flow shares similarities with that of a traditional single-core compiler, but is much more complex in the case of a multi-core embedded system. In this chapter it was presented some foundations and described approaches to deal with these problems. However, there is still a great amount of research to be done to make the leap from a high level specification to executable code as transparent as it is in the single-core case.

References

1. Eclipse. http://www.eclipse.org/. Visited on Jan. 2010
2. GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/. Visited on Jan. 2010
3. OpenMP Application Programming Interface. Version 4.5. http://www.openmp.org. Visited on Mar. 2017
4. AbsInt: aiT worst-case execution time analyzers. http://www.absint.com/ait/. Visited on Nov. 2009
5. Agbaria, A., Kang, D.I., Singh, K.: LMPI: MPI for heterogeneous embedded distributed systems. In: 12th International Conference on Parallel and Distributed Systems - (ICPADS'06), vol. 1, pp. 8 pp.– (2006)
6. Aguilar, M.A., Aggarwal, A., Shaheen, A., Leupers, R., Ascheid, G., Castrillon, J., Fitzpatrick, L.: Multi-grained Performance Estimation for MPSoC Compilers: Work-in-progress. In: Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion, CASES '17, pp. 14:1–14:2. ACM, New York, NY, USA (2017)
7. Aguilar, M.A., Eusse, J.F., Ray, P., Leupers, R., Ascheid, G., Sheng, W., Sharma, P.: Towards parallelism extraction for heterogeneous multicore Android devices. International Journal of Parallel Programming pp. 1–33 (2016)
8. Aguilar, M.A., Leupers, R., Ascheid, G., Kavvadias, N.: A toolflow for parallelization of embedded software in multicore DSP platforms. In: Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES '15, pp. 76–79. ACM, New York, NY, USA (2015)

9. Aguilar, M.A., Leupers, R., Ascheid, G., Murillo, L.G.: Automatic parallelization and accelerator offloading for embedded applications on heterogeneous MPSoCs. In: Proceedings of the 53rd Annual Design Automation Conference, DAC '16, pp. 49:1–49:6. ACM, New York, NY, USA (2016)
10. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
11. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley. Tech. rep., EECS Department, University of California, Berkeley (2006)
12. Bacivarov, I., Haid, W., Huang, K., Thiele, L.: Methods and tools for mapping process networks onto multi-processor systems-on-chip. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
13. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MPSoCs via decomposition and no-good generation. Principles and Practices of Constrained Programming - CP 2005 (DEIS-LIA-05-001), 107–121 (2005)
14. Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing **49**(10), 2408–2421 (2001)
15. Carro, L., Rutzig, M.B.: Multi-core systems on chip. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
16. Castrillon, J., Leupers, R.: Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap. Springer Publishing Company, Incorporated (2013)
17. Castrillon, J., Sheng, W., Jessenberger, R., Thiele, L., Schorr, L., Juurlink, B., Alvarez-Mesa, M., Pohl, A., Reyes, V., Leupers, R.: Multi/many-core programming: Where are we standing? In: 2015 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1708–1717 (2015)
18. Castrillon, J., Sheng, W., Leupers, R.: Trends in embedded software synthesis. In: SAMOS, pp. 347–354 (2011)
19. Ceng, J.: A methodology for efficient multiprocessor system on chip software development. Ph.D. thesis, RWTH Aachen Univeristy (2011)
20. Ceng, J., Castrillon, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T., Kunieda, H.: MAPS: an integrated framework for MPSoC application parallelization. In: DAC '08: Proceedings of the 45th annual conference on Design automation, pp. 754–759. ACM, New York, NY, USA (2008)
21. Cesario, W., Jerraya, A.: Multiprocessor Systems-on-Chips, chap. Chapter 9. Component-Based Design for Multiprocessor Systems-on-Chip, pp. 357–394. Morgan Kaufmann (2005)
22. Cordes, D.A.: Automatic parallelization for embedded multi-core systems using high-level cost models. Ph.D. thesis, TU Dortmund (2013)
23. Diakopoulos, N., Cass, S.: The top programming languages 2016. http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016. Visited on Feb. 2017
24. Fisher, J., P., F., Young, C.: Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. Morgan-Kaufmann (Elsevier) (2005)
25. Gao, L., Huang, J., Ceng, J., Leupers, R., Ascheid, G., Meyr, H.: TotalProf: a fast and accurate retargetable source code profiler. In: CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, pp. 305–314. ACM, New York, NY, USA (2009)
26. Geilen, M., Basten, T.: Kahn process networks and a reactive extension. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
27. Gheorghita, S., T. Basten, H.C.: An overview of application scenario usage in streaming-oriented embedded system design. www.es.ele.tue.nl/esreports/esr-2006-03.pdf. Visited on Mar. 2017

28. Gupta, R., Micheli, G.D.: Hardware-software co-synthesis for digital systems. In: IEEE Design & Test of Computers, pp. 29–41 (1993)
29. Ha, S., Oh, H.: Decidable signal processing dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
30. Hewitt, C., Bishop, P., Greif, I., Smith, B., Matson, T., Steiger, R.: Actor induction and meta-evaluation. In: POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 153–168. ACM, New York, NY, USA (1973)
31. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: PASTE '01, pp. 54–61. ACM Press (2001)
32. Hu, T.C.: Parallel sequencing and assembly line problems. Oper. Res. **9**(6), 841–848 (1961)
33. Hwang, Y., Abdi, S., Gajski, D.: Cycle-approximate retargetable performance estimation at the transaction level. In: DATE '08: Proceedings of the conference on Design, automation and test in Europe, pp. 3–8. ACM, New York, NY, USA (2008)
34. Hwu, W.M., Ryoo, S., Ueng, S.Z., Kelm, J.H., Gelado, I., Stone, S.S., Kidd, R.E., Baghsorkhi, S.S., Mahesri, A.A., Tsao, S.C., Navarro, N., Lumetta, S.S., Frank, M.I., Patel, S.J.: Implicitly parallel programming models for thousand-core microprocessors. In: DAC '07: Proc. of the 44th Design Automation Conference, pp. 754–759. ACM, New York, NY, USA (2007)
35. Johnson, R.C.: Efficient program analysis using dependence flow graphs. Ph.D. thesis, Cornell University (1994)
36. Kahn, G.: The semantics of a simple language for parallel programming. In: J.L. Rosenfeld (ed.) Information Processing '74: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York, NY (1974)
37. Kandemir, M., Dutt, N.: Multiprocessor Systems-on-Chips, chap. Chapter 9. Memory Systems and Compiler Support for MPSoC Architectures, pp. 251–281. Morgan Kaufmann (2005)
38. Karp, R.M., Miller, R.E.: Properties of a model for parallel computations: Determinacy, termination, queuing. SIAM Journal of Applied Math **14**(6) (1966)
39. Karuri, K., Al Faruque, M.A., Kraemer, S., Leupers, R., Ascheid, G., Meyr, H.: Fine-grained application source code profiling for ASIP design. In: DAC '05: Proceedings of the 42nd annual conference on Design automation, pp. 329–334. ACM, New York, NY, USA (2005)
40. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
41. Khronos Group: OpenCL embedded boards comparison 2015. https://www.khronos.org/news/events/opencl-embedded-boards-comparison-2015. Visited on Mar. 2017
42. Kung, H.T.: Why systolic architectures? Computer **15**(1), 37–46 (1982)
43. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. **31**(4), 406–471 (1999)
44. Kwon, S., Kim, Y., Jeun, W.C., Ha, S., Paek, Y.: A retargetable parallel-programming framework for MPSoC. ACM Trans. Des. Autom. Electron. Syst. **13**(3), 1–18 (2008)
45. Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. SIGPLAN Not. **23**(7), 318–328 (1988)
46. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
47. Lee, E.A.: Consistency in dataflow graphs. IEEE Trans. Parallel Distrib. Syst. **2**(2), 223–235 (1991)
48. Lengauer, C.: Loop parallelization in the polytope model. In: Proceedings of the 4th International Conference on Concurrency Theory, CONCUR '93, pp. 398–416. Springer-Verlag, London, UK, UK (1993)
49. Leupers, R.: Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, Norwell, MA, USA (1997)
50. Leupers, R.: Code selection for media processors with SIMD instructions. In: DATE '00, pp. 4–8. ACM (2000)

51. Li, L., Huang, B., Dai, J., Harrison, L.: Automatic multithreading and multiprocessing of C programs for IXP. In: PPoPP '05: Proc. of the 10th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 132–141. ACM, New York, NY, USA (2005)
52. Ma, Z., Marchal, P., Scarpazza, D.P., Yang, P., Wong, C., Gmez, J.I., Himpe, S., Ykman-Couvreur, C., Catthoor, F.: Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms. Springer (2007)
53. Martin, G.: ESL requirements for configurable processor-based embedded system design. http://www.us.design-reuse.com/articles/article12444.html. Visited on Mar. 2017
54. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
55. Multicore Association: MCAPI - Multicore Communications API. http://www.multicore-association.org/workgroup/mcapi.php. Visited on Mar. 2017
56. Multicore Association: Software-hardware interface for multi-many-core (SHIM) specification v1.00. http://www.multicore-association.org. Visited on Mar. 2017
57. National Instruments: LabView. http://www.ni.com/labview/. Visited on Mar. 2017
58. Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E.: Daedalus: Toward composable multimedia MP-SoC design. In: DAC '08: Proceedings of the 45th annual conference on Design automation, pp. 574–579. ACM, New York, NY, USA (2008)
59. Palsberg, J., Naik, M.: Multiprocessor Systems-on-Chips, chap. Chapter 12. ILP-based Resource-aware Compilation, pp. 337–354. Morgan Kaufmann (2005)
60. Paolucci, P.S., Jerraya, A.A., Leupers, R., Thiele, L., Vicini, P.: SHAPES:: a tiled scalable software hardware architecture platform for embedded systems. In: CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, pp. 167–172. ACM, New York, NY, USA (2006)
61. Parks, T.M.: Bounded scheduling of process networks. Ph.D. thesis, Berkeley, CA, USA (1995)
62. Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.F., Aridhi, S.: Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In: 2014 6th European Embedded Design in Education and Research Conference (EDERC), pp. 36–40 (2014). DOI 10.1109/EDERC.2014.6924354
63. Polychronopoulos, C.D.: The hierarchical task graph and its use in auto-scheduling. In: Proceedings of the 5th International Conference on Supercomputing, ICS '91, pp. 252–263. ACM, New York, NY, USA (1991)
64. Rabenseifner, R., Hager, G., Jost, G.: Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 427–436 (2009)
65. Sharma, G., Martin, J.: MATLAB (R): A language for parallel computing. International Journal of Parallel Programming **37**(1) (2009)
66. Silexica: SLX Tool Suite. http://www.silexica.com. Visited on Mar. 2017
67. Sporer, T., Franck, A., Bacivarov, I., Beckinger, M., Haid, W., Huang, K., Thiele, L., Paolucci, P., Bazzana, P., Vicini, P., Ceng, J., Kraemer, S., Leupers, R.: SHAPES - a scalable parallel HW/SW architecture applied to wave field synthesis. In: Proc. 32nd Intl Audio Engineering Society Conference, pp. 175–187. Audio Engineering Society, Hillerod, Denmark (2007)
68. Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker, Inc., New York, NY, USA (2000)
69. Standard for information technology - portable operating system interface (POSIX). Shell and utilities. IEEE Std 1003.1-2004, The Open Group Base Specifications Issue 6, section 2.9: IEEE and The Open Group
70. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (2010)
71. Stotzer, E.: Towards using OpenMP in embedded systems. OpenMPCon: Developers Conference (2015)
72. Synopsys: Virtual Platforms. https://www.synopsys.com/verification/virtual-prototyping.html. Visited on Mar. 2017

73. Texas Instruments: Keystone Multicore Devices. http://processors.wiki.ti.com/ index.php/Multicore. Visited on Mar. 2017
74. Texas Instruments: Software development kit for multicore DSP Keystone platform. http://www.ti.com/tool/bioslinuxmcsdk. Visited on Mar. 2017
75. Theelen, B.D., Deprettere, E.F., Bhattacharyya, S.S.: Dynamic dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
76. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.: Towards a holistic approach to auto-parallelization – integrating profile-driven parallelism detection and machine-learning based mapping. In: PLDI 0-9: Proceedings of the Programming Language Design and Implementation Conference. Dublin, Ireland (2009)
77. Vargas, R., Quinones, E., Marongiu, A.: OpenMP and timing predictability: A possible union? In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE '15, pp. 617–620. EDA Consortium, San Jose, CA, USA (2015)
78. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: A tool for improved derivation of process networks. EURASIP J. Embedded Syst. **2007**(1), 19–19 (2007)
79. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 1–53 (2008)
80. Working Group ISO/IEC JTC1/SC22/WG14: C99, Programming Language C ISO/IEC 9899:1999
81. Zalfany Urfianto, M., Isshiki, T., Ullah Khan, A., Li, D., Kunieda, H.: Decomposition of task-level concurrency on C programs applied to the design of multiprocessor SoC. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E91-A**(7), 1748–1756 (2008)