1

Software Compilation and Optimization Techniques for Heterogeneous Multicore Platforms

Weihua SHENG[1], Jeronimo CASTRILLON[2] and Rainer LEUPERS[3]

[1]Huawei Hong Kong Research Center.
[2]Technische Universität Dresden. cfaed, 01062 Dresden, Germany.
[3]RWTH Aachen University. Templergraben 55, 52056 Aachen, Germany.

This chapter addresses the challenges associated with compilation and optimization techniques for heterogeneous multicore computing systems in the embedded industry. Wireless terminals and modems are typical examples of such systems, which demand high performance and energy efficiency at the same time. To fully exploit the computing power of those systems, the existing compiler technology for single processor systems does not suit the need and scale for multicore architectures anymore. The authors have applied a systematic approach to tackle the problems of application modeling, source-to-source compilation, flexible compiler infrastructure construction and software distribution for multicore architectures from a practical perspective. Several real-world multicore platforms as well as system-level virtual platforms have been successfully used to demonstrate the achievable speed-ups and versatility of the compilation and optimization techniques developed in this work.

Multi-Processor System-on-Chip: Vol. 2 – Applications,
coordinated by Weihua SHENG. © ISTE Editions 2019.

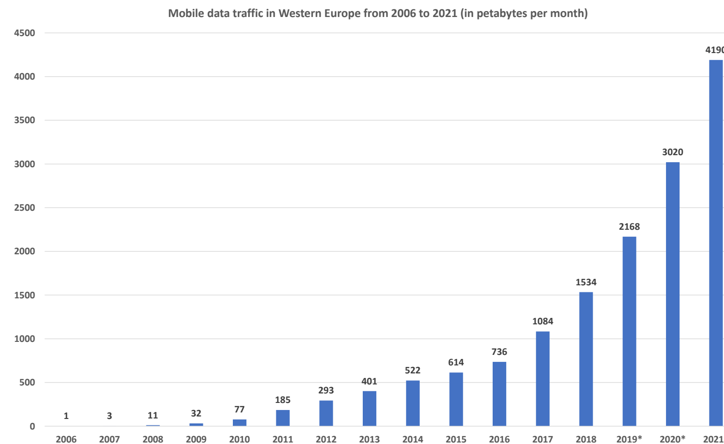

Figure 1.1 – Mobile data traffic in Western Europe from 2006 to 2021 (2019-2021 are projections), adapted from (A.T. Kearney 2010) and (Cisco Systems 2017)

1.1. Introduction

In the HPC (High-Performance Computing) industry, computer architecture evolution is predominantly driven by the increasing demand for superior performance. The trend of scaling up the frequency and number of transistors continued as long as possible, until the physical limits were reached and the power consumption became unmanageable, and CPU performance growth hit a wall around 2003. Between 2003 and 2005, multicore architectures were first introduced in mainstream computing products, such as desktop PCs and high-end servers, by semiconductor vendors for enhanced computing capabilities. The embedded industry followed suit, particularly adopting those multicores that consist of different types of processor (aka heterogeneous multicores).

Take the mobile phone as an example. Fig. 1.1 shows the ongoing mobile traffic growth in western Europe with projections until 2021. In 2016, western Europe's mobile data traffic amounted to 736,377 terabytes per month (Cisco Systems 2017). In 2021, the western European traffic from mobile devices (smartphones, laptops, tables, M2M connections, etc.) is projected to reach 4,189,615 terabytes per month. This projected compound annual growth rate (CAGR) amounts to 42 percent.

High-end smartphones nowadays feature high-definition video playback, fully fledged Internet capabilities, and multi-standard radio protocols. The list of integrated functions demanded by consumers continues to grow. The workload of a typical 4G smartphone had increased to 100 giga operations per second (GOPS) per Watt in 2010, including 40 GOPS for radio, 20 GOPS from media processing and 6.5 GOPS for graphics. The workload continues to increase at a steady rate, roughly by

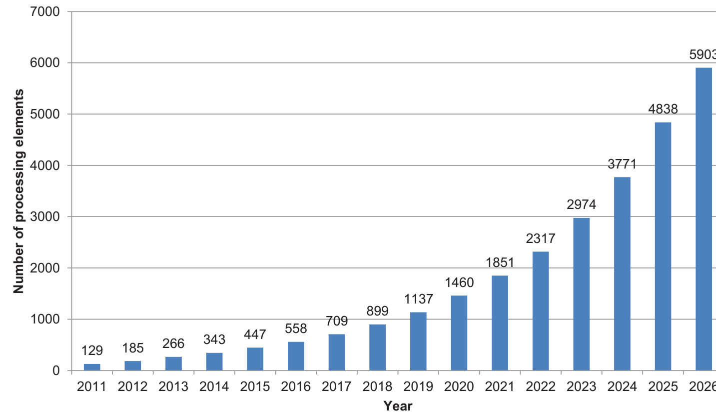

Figure 1.2 – SOC Consumer Portable Design Trends (adapted from (International Technology Roadmap for Semiconductors 2011)). The processing elements are referred to cores or parallel execution units in CPUs, GPUs, accelerators, etc.

an order of magnitude every five years (van Berkel 2009). Multiprocessing was introduced to mobile phone technology to address this requirement shortly after the shift in the HPC industry. The International Technology Roadmap for Semiconductors (ITRS) has recently predicted an exponential growth in the number of processing elements that consumer portable products will contain, estimated to reach almost 6000 cores in 2026 (as illustrated in Fig. 1.2).

Compared to other domains, embedded systems have much more stringent requirements in terms of power/energy efficiency. Today's mobile phones are expected to run on a single battery for hours to days without requiring recharging. As mentioned earlier, the power budget for the digital workload of a 4G smartphone (100 GOPS) is around 1W. This is significantly more power efficient than most modern PCs. The breakdown analysis of the workload vs. power consumption has shown that even homogeneous multiprocessing is realistically insufficient to meet this requirement (van Berkel 2009). The only feasible approach is to adopt heterogeneous multicore architectures that consist of specialized programmable processors and/or hardware accelerators; these offer a superior solution for the trade-off between performance and power.

Unfortunately, the flourishing of multicore architectures led by semiconductor companies has left several unresolved challenges from the perspective of software development.

To illustrate the current practice of MPSoC programming, Fig. 1.3 shows a comparison of programming flows for uni-processor systems and for MPSoCs. In the uni-processor flow, software programmers follow the sequential programming model (C

being the most popular language) and rely on the compilers to generate the target-specific code correctly and optimally, as shown in Fig. 1.3 (a). Software programmers focus on structuring algorithms correctly and are shielded from low-level architectural details by using a vendor compiler. This has been a successful practice in recent decades prior to the introduction of MPSoCs, thanks to the pivotal role of compilers. However, the traditional compiler technology does not scale for MPSoCs. Fig. 1.3 (b) demonstrates the current problematic programming flow for MPSoCs, which results in low software productivity. Applications must first be partitioned into parallel tasks, followed by spatial and temporal mapping of those partitioned tasks onto the MPSoC processing elements. As explained earlier, the programmable processors in heterogeneous MPSoCs nowadays often come with their own compilers and have their own software stacks (API, OS). Therefore, after partitioning and mapping, the correct code must be generated for each of the individual processors respectively, to be further compiled. Compared to the uni-processor, these are new, non-trivial tasks that are the programmers' responsibility. Unlike the pivotal role of traditional compilers in the uni-processor flow, there is little compiler support for the MPSoC programming. This practice is currently labor-intensive, error-prone and costly.

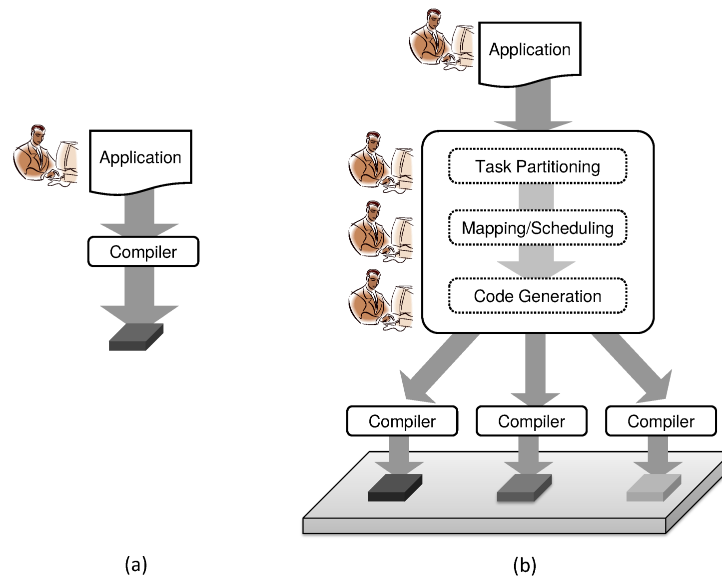

Figure 1.3 – Programming Flow (a) Uni-processor: Compilers perform an end-to-end translation from parsing source code, target independent optimization, target dependent optimization to final binary generation. (b) MPSoC: No compilation framework for MPSoCs is in place. The process (task partitioning, mapping/scheduling and code generation) is manual.

This chapter addresses the challenges associated with developing compilers and their optimizations for heterogeneous multicores for software development, focusing on embedded systems and using dataflow models [1]. This work has been developed as part of the MPSoC Application Programming Studio (MAPS) research project at the Institute for Communication Technologies and Embedded Systems (ICE) of RWTH Aachen University, which provided the technology foundation of the software tool provider Silexica Inc.

The remainder of the chapter is organized as follows. The dataflow modeling (especially process networks) is discussed in section 1.2. A clean, lightweight C language extension, CPN (i.e., C for process networks), is defined to capture streaming programming models. A retargetable source-to-source compiler has been developed to provide the key capabilities that facilitate the construction of compiler infrastructures for real-world, complex multicore architectures. Their details are explained in section 1.3. Some optimization techniques for software distribution are explained in section 1.4. Our experimental results are presented in 1.5. The chapter ends with a short summary in section 1.6.

1.2. Dataflow Modeling

1.2.1. *General Concepts*

Dataflow models subsume a family of models of computation where parallel entities are only allowed to communicate over dedicated channels (Ptolemaeus 2014). Intuitively, in a dataflow model an application is represented as a graph where nodes represent computation and edges represent first-in first-out (FIFO) communication buffers. Dataflow models differ from each other in the way computation can be triggered upon conditions in incoming edges. For instance, in synchronous dataflow (SDF) (Lee and Messerschmitt 1987), a prominent dataflow model, nodes have a fix condition upon which they can be executed (or *fired*). Additionally, one a node is executed it always produces a fixed amount of data items, called tokens. In the example in Fig. 1.4a, node *a2* requires 4 data items in its input channel and, when executed, it consumes the tokens and produce a single data item in its output. Such a node can, for example, model an averaging task that takes four integer and returns the average as floating number. A model as in the example, enables automatic reasoning about how such an application may make progress. For example, given that two tokens are available in *e3*, node *a1* can be executed twice, each time producing 2 tokens on *e1*. If extended with estimates of the duration each node execution, a schedule can be constructed to understand the earliest time at

1. Sequential code partitioning is also an important and essential part of this work. Due to the chapter length constraint, we omit the details here. Please refer to (Castrillon *et al*. 2011 ; Leupers and Castrillon 2010) and (Castrillon Mazo 2013).

which *a2* can be executed. The ability to construct efficient schedules at compile-time made SDFs quite appealing for modeling real time applications.

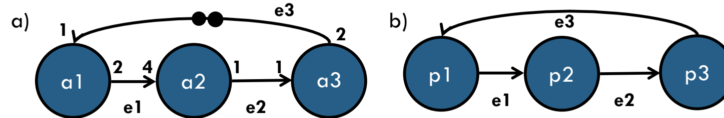

Figure 1.4 – a) Example SDF graph. Actors have fixed consumption and production rates. Initial tokens are represented as dots on the channels. b) Example of a process network. The abstract specification expresses that nodes can communicate if there is a channel that connects them, but does not specify how this communication takes place.

SDFs are simple in that they can only model static behavior. Over time, models have been extended to describe applications with more dynamic behavior, e.g., Cyclo-static SDF (CSDF) (Bilsen *et al*. 1996) and Variable-rate Phased Dataflow (Wiggers 2009) (see also (Stuijk *et al*. 2011) for an attempt to compare different models). The popularity of these models for embedded software (Sriram and Bhattacharyya 2009 ; Bhattacharyya *et al*. 2010) motivated research on frameworks for the analysis and synthesis of dataflow models (DOL (Thiele *et al*. 2007 ; Giannopoulou *et al*. 2016), CAL (Eker and Janneck 2003 ; Brunet 2015), MAPS (Leupers and Castrillon 2010 ; Leupers *et al*. 2017), SystemCoDesigner (Haubelt *et al*. 2007 ; Keinert *et al*. 2009), PREESM (Pelcat *et al*. 2014) or Daedalus (Bamakhrama *et al*. 2012 ; Nikolov 2009) to name a few).

1.2.2. *Process Networks*

Process networks are a model of computation in which applications are also represented as a graph with FIFO communication channels, similar to dataflow models. In contrast to dataflow models, there are no triggering conditions specified in the input channels of a node in a process network, i.e., no *firing semantics*. Instead, the graph representation only specify that nodes (aka processes) can communicate if a channel connects them. In Fig. 1.4b, for instance, processes *p1* and *p2* could execute in parallel for a long time before any actual data exchange take place. A process network can model not only SDF, CSDF and other simple dataflow graphs, but also more complex communication behavior. In particular, in a process network, it is possible to model channel accesses that depend on concrete values of incoming data.

Process networks are more expressive than many dataflow models, in the sense that they can model a larger set of applications. Since every process can be thought of as a independent thread, questions about termination or scheduling are undecidable. An interesting property of Kahn Process Networks (KPN) is that their execution is *determinate* (Gilles Kahn 1974), i.e., provided with the same input, the history

```
1  typedef struct { int i; double d; } my_struct_t;
2  typedef union { float f; short s[4]; } my_union_t;
3
4  __PNchannel char B[3][3];
5  __PNchannel my_struct_t C;
6  __PNchannel my_union_t D;
7  __PNchannel int A = {1, 2, 3}; /* Initial channel tokens */
```

Listing 1 – CPN Example Code: Channel Declaration

of tokens in all the channels is independent of the schedule. This is enabled by the fact that processes are not allowed to peek into channels before reading, but would block immediately when attempting to read from an empty channel. This is typically called *blocking-reads semantics*. Intuitively, whether peeking into a channel is successful depends on the time of arrival of tokens which does depend on the schedule. Determinism is an important property that enables automatic optimization of process networks without modifying the application behavior.

1.2.3. *C for Process Networks*

MAPS defines a clean, light-weight C language extension called CPN (C for Process Networks) to capture streaming models. A minimum set of new keywords is added to the C language to describe processes and channels. A language extension approach allows the user to specify the semantics of process networks at a high level, for example, containing enough structural information about the channel accesses. This first enables retargetability towards typical embedded MPSoCs where processing elements have different APIs and specific low-level primitives (that often cannot be abstracted by a common API). Second, programs are portable and the semantic analysis offers abundant opportunities for code transformations and optimizations (e.g. process fusion and fission). Although C is not an ideal vehicle in which to carry the concurrency specification, this design compromise is made considering the large C legacy code base and popularity of C in the embedded industry.

Channels

Channel declaration in CPN is similar to declaring a global variable in C with an additional keyword `__PNchannel`. Examples can be found in Listing 1. Elementary C types, such as `int`, `char` and `float`, and enumerations are valid channel types. Structures, unions and arrays of valid channel types are valid, too.

A channel is, by default, empty at the beginning of the execution. If initial tokens are needed on the channels, e.g., to avoid deadlocks, CPN also supports having initial tokens in channels by specifying initializers in the channel declaration (channel `A` at line 7 in Listing 1).

Processes

Similar to C++ templates, the concept of process templates is used in CPN for code reuse. A process template describes the functionality of a process and the channels this process needs to access (either read or write). Processes are always created as instances of process templates. The code's readability and conciseness is improved, for example, when multiple processes in a network share the same functionality.

An example of a KPN process template is shown in Listing 2. It describes the functionality of Run-Length Decoding (RLD). Run-Length Encoding (RLE) is a simple data compression technique used in fax machines, for example. Data are encoded into a stream of duos, i.e., the number of appearances and the data element itself. For example, the original data `AAAABBCCCCCDDD` is compressed into `4A2B5C3D`. RLD is the inverse of RLE. A KPN process template (`__PNkpn`) with the identifier `RLD` is shown in Listing 2. It reads integers from its input channel `EncIn` and outputs integers to the channel `DecOut` indicated by keywords `__PNin` and `__PNout` respectively. The body of a KPN process template can contain arbitrary code, which is allowed to access input and output channels at any point of its control flow.

The access to a channel is always explicitly made via `__PNin` or `__PNout` in the body for KPN-like processes, as the code generator cannot know if two consecutive accesses to the same channel intend to access the same channel item (token) or the next channel item in the general case. Those statements enable access to the next data tokens (read) or free entries (write) in the channel, respectively. The code inside the body of the `__PNin` or `__PNout` statement can access those items like a local variable. At lines 6-7 of Listing 2, first, a token from the input channel `EncIn` is read and assigned to a local variable `count`. Next, the data are decoded by writing the encoded data into the output channel `DecOut` using in the loop of lines 9-11. Both the `__PNin` and the `__PNout` statements have blocking semantics, i.e. they will suspend process execution until the channel contains enough data tokens or free entries.

The explicit channel access annotation looks somewhat unnecessary in the first place. A brief explanation is provided here to illustrate why such an explicit annotation is required, using Listing 2 as an example. If the example were without `__PNin` and `__PNout` statements, the accesses to channel EncIn at line 7 and 11 would become ambiguous. It is not possible for the code generator to reason that, in the case of line 7, a new token from the channel EncIn is read while in the case of line 11 a new token is read from the channel EncIn only at the first loop iteration and will be reused during the rest of the loop (lines 9-11). Any form of dependence analysis would not be sufficient to solve the problem because it requires the understanding of the application behavior. Therefore, the functionality of RLD could not be correctly specified. The code generator requires a clear, unambiguous indication as to whether or not to fetch tokens from a channel (or write tokens to a channel).

```
1  /* Run Length Decoding, e.g. 4A2B5C3D -> AAAABBCCCCCDDD */
2  __PNkpn RLD __PNin(int EncIn) __PNout(int DecOut)
3  {
4      int count, i;
5      while (1) {
6          __PNin(EncIn) /* read a token (# of appearances) from
               EncIn, e.g. 4 */
7              count = EncIn;
8          __PNin(EncIn) /* read a token (data itself) from EncIn,
               e.g. A */
9              for (i = 0; i < count; ++i) /* write data to DecOut,
                   e.g 4 times of A */
10                 __PNout(DecOut)
11                     DecOut = EncIn;
12     }
13 }
```

Listing 2 – CPN Example Code: KPN Process Template (Run Length Decoding)

```
1 __PNsdf Add __PNin(int a, int b) __PNout(int sum) {
2   /* initialization code could be placed here */
3   __PNloop { /* infinite loop */
4     /* a and b are read from the channel implicitly */
5     sum = a + b; /* channel variables are accessible in C code */
6     /* sum is written to the channel implicitly */
7   }
8 }
```

Listing 3 – CPN Example Code: SDF Process Template (Adder)

As the SDF frequently appears in streaming applications, a shortcut is provided in CPN to simplify the program representation. The example in Listing 3 defines an SDF process template (`__PNsdf`) for the functionality of an adder, reading integers from input channels `a` and `b` and writing integers to channel `sum`. The `__PNloop` statement resembles the infinite loop of an SDF process. Its body contains all the code to be executed between reading from all input channels and writing to all output channels. SDF-like processes implicitly access all their input and output channels in the `__PNloop` statement (line 3 in Listing 3). The number of tokens to access in every iteration is given in the `__PNin` and `__PNout` clauses of the SDF template header. If this number is not specified, the default value 1 is used.

Processes can be instantiated from previously defined process templates using `__PNprocess`. Listing 4 creates processes from the process templates defined above and connects channels to them.

Parallelism Types

CPN is designed to describe streaming applications and, more importantly, specify the parallelism in those programs explicitly. In this section, the parallelism types that

```
1 __PNchannel int decoder1_input = {4, 'A', 2, 'B', 5, 'C', 3, 'D'};
2 __PNchannel int decoder2_input = {3, 'E', 5, 'F', 4, 'G', 2, 'H'};
3
4 __PNchannel int decoder1_output, decoder2_output;
5 __PNchannel int add_output;
6
7 __PNprocess decoder1 = RLD __PNin(decoder1_input)
      __PNout(decoder1_output);
8 __PNprocess decoder2 = RLD __PNin(decoder2_input)
      __PNout(decoder2_output);
9 __PNprocess add = Add __PNin(decoder1_output, decoder2_output)
      __PNout(add_output);
```

Listing 4 – CPN Example Code: Process Instantiation

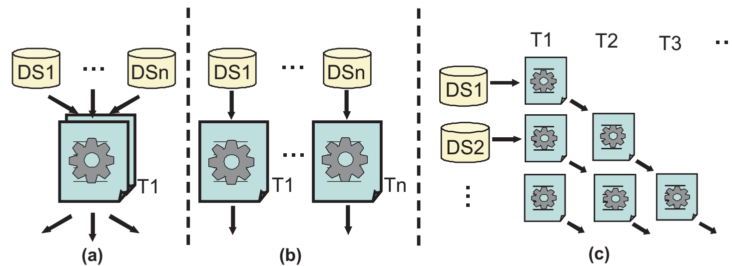

Figure 1.5 – Types of Parallelism: (a) DLP, (b) TLP, (c) PLP. (DS stands for Data Set and T stands for Task)

are most commonly seen in embedded streaming applications are introduced first and we demonstrate that CPN is able to express all of those parallelism types succinctly.

Many different types of parallelism exist in real-world applications. In the embedded domain that is the focus of this chapter, the most prominent types of coarse or macroscopic parallelism are data-level, task-level, and pipeline-level. These are illustrated in Figure 1.5 with descriptions below.

– Data-Level Parallelism (DLP): DLP represents a typical scenario where the same computational task is carried out on several disjoint data sets as illustrated in Figure 1.5 (a). DLP can be considered as a generalization of Single Instruction, Multiple Data (SIMD). DLP exists abundantly, e.g., in multimedia applications, where a decoding task performs the same operations on different portions of an image or video (e.g., Macroblocks in H.264). An example code to describe the DLP is shown in Listing 5. As with other programming models like OpenMP, where data parallelism is defined implicitly, we have worked on extensions that allow defining data-parallel processes and methodologies to change the amount of parallelism dynamically (Khasanov *et al*. 2018).

```
1 __PNchannel int DS1, DS2, DSN;
2 __PNchannel int Result1, Result2, ResultN;
3 __PNprocess T_1 = SameTaskT1 __PNin(DS1) __PNout(Result1);
4 __PNprocess T_2 = SameTaskT1 __PNin(DS2) __PNout(Result2);
5 __PNprocess T_N = SameTaskT1 __PNin(DSN) __PNout(ResultN);
```

Listing 5 – CPN Example Code: DLP

```
1 __PNchannel int DS1, DS2, DSN;
2 __PNchannel int Result1, Result2, ResultN;
3 __PNprocess T_1 = Task1 __PNin(DS1) __PNout(Result1);
4 __PNprocess T_2 = Task2 __PNin(DS2) __PNout(Result2);
5 __PNprocess T_N = TaskN __PNin(DSN) __PNout(ResultN);
```

Listing 6 – CPN Example Code: TLP

– Task- (or functional) Level Parallelism (TLP): It is common that in a computer program, different tasks can compute in parallel on different data sets. TLP is used to specify this kind of behavior, as shown in Figure 1.5 (b). Tasks may have dependencies on one another, but once a task has its data ready, it can execute in parallel with the already-running tasks in the system. An example code to describe the TLP is shown in Listing 6.

– Pipeline-Level Parallelism (PLP): In PLP, computation is broken into a sequence of tasks that are executed repetitively for different data sets, as shown in Figure 1.5 (c). These tasks are also called pipeline stages. It follows the principle of pipelining to achieve a higher throughput. At a given time, different tasks of the original functionality are executed concurrently on different data sets. This type of parallelism is most common in the signal processing domain, where the throughput is a critical system performance indicator. An example code to describe the PLP is shown in Listing 7.

```
1 __PNchannel pixels DS, intermediate1, intermediate2, intermediate3,
      output;
2 __PNprocess source = ReadImage __PNout(DS);
3 __PNprocess T_1 = PipelineTask1 __PNin(DS) __PNout(intermediate1);
4 __PNprocess T_2 = PipelineTask2 __PNin(intermediate1)
      __PNout(intermediate2);
5 __PNprocess T_3 = PipelineTask3 __PNin(intermediate2)
      __PNout(intermediate3);
6 __PNprocess T_N = PipelineTaskN __PNin(intermediate3)
      __PNout(output);
7 __PNprocess sink = WriteImage __PNin(output);
```

Listing 7 – CPN Example Code: PLP

1.3. Source-to-source Based Compiler Infrastructure

Once applications are written in CPN, a compiler framework is needed to take them as input, compile and generate the correct and optimized binary code for target MPSoC platforms. The complexity of MPSoCs nowadays requires a more flexible design in reality. In this section, the compiler design rationale is first discussed, followed by the details of our implementation.

1.3.1. *Design rationale*

Many previous approaches for multiprocessor systems have followed the principle of compiler design for uni-processors: the compiler works on input programs, builds an internal intermediate representation, performs optimizations and generates the target binary code. This black box monolithic design philosophy is:

1) the compiler generates the correct and optimized target code without (much) help from programmers;

2) the compiler has as much information about architectural details as possible in order to perform target-specific code generation and optimization.

The essence of this design philosophy is to incorporate the complexity into a single tool (compiler) that has been developed by a small group of highly skilled experts. Therefore, programmers exhibit enhanced productivity in designing the software that assumes a straightforward programming model (sequential) and a simple, common memory model.

We studied and revisited this monolithic design approach used in the past few decades (shown in Figure 1.6) and our observations are:

1) The complexity of hardware architectures grows so rapidly that a monolithic compiler is unable to keep pace with it. For most uni-processors, such as RISCs, compilers still manage to include architectural details. However, the gap between the architecture complexity and the amount of information that a single tool can incorporate has grown wider: more complex architecture templates appear, such as (clustered-) VLIW DSPs, ASIPs (Application-Specific Instruction set Processors) and eventually multicores, which have multiple scalar processors on a single chip. The level of user intervention required in developing and using compilers increases rapidly (shown in Figure 1.6), which offers further proof that the monolithic approach is reaching its feasibility limits.

2) Compilers usually take a significant amount of time to mature along with extensive investment. As already explained, heterogeneous MPSoC platforms will increasingly utilize individual IPs from different vendors. Therefore, a monolithic approach to multicores that generates the target specific binary code (for different processors) directly is not economically sound, since it does not leverage the existing C compilers for uni-processors.

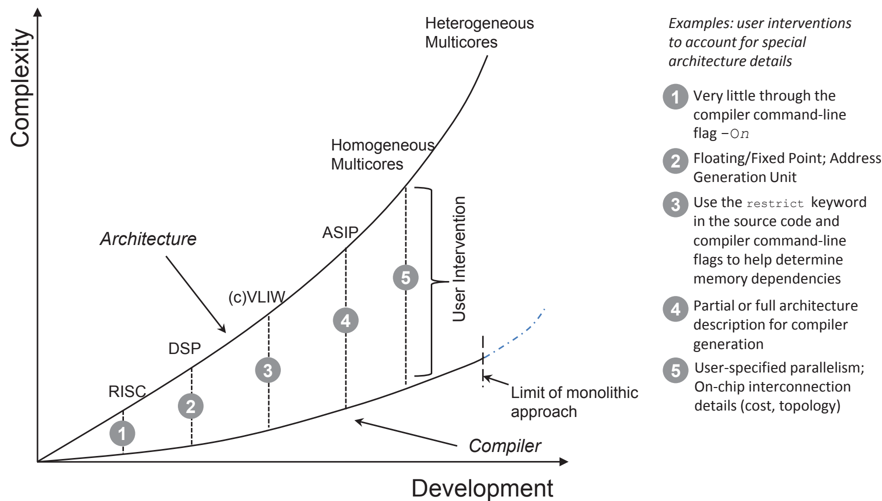

Figure 1.6 – Evolution of compiler development vs. architecture development

3) Compiler usage scenarios are now more diverse, particularly in the embedded domain. Multiple optimization goals are commonly seen, such as the performance, the energy efficiency and the code size.

Moving along the trajectory shown in Figure 1.6, we argue that for heterogeneous multicore systems, a new flexible, extensible compiler design is both desirable and required, as opposed to persisting with the monolithic approach. The user interventions to account for the new architecture trends will increase, including a user-defined mapping specification and objective functions for code generation and optimization.

1.3.2. *Implementation strategy*

The software architecture of the compiler framework developed to compile CPN programs to heterogeneous MPSoCs is introduced below. The complete compilation flow for a specific MPSoC is a tool framework consisting of many components. The core component cpn-cc, a source-to-source (CPN-to-C) compiler, is first elaborated, followed by a description of how the complete compilation framework can be constructed in a flexible, extensible manner guided by the user.

Instead of a monolithic approach, a source-to-source (CPN-to-C) compiler, cpn-cc, was developed as the core component in the framework. The cpn-cc is implemented based on Clang (Lattner 2008), the C frontend of the LLVM compiler infrastructure. Figure 1.7 (a) shows a high-level, internal structure of cpn-cc which consists

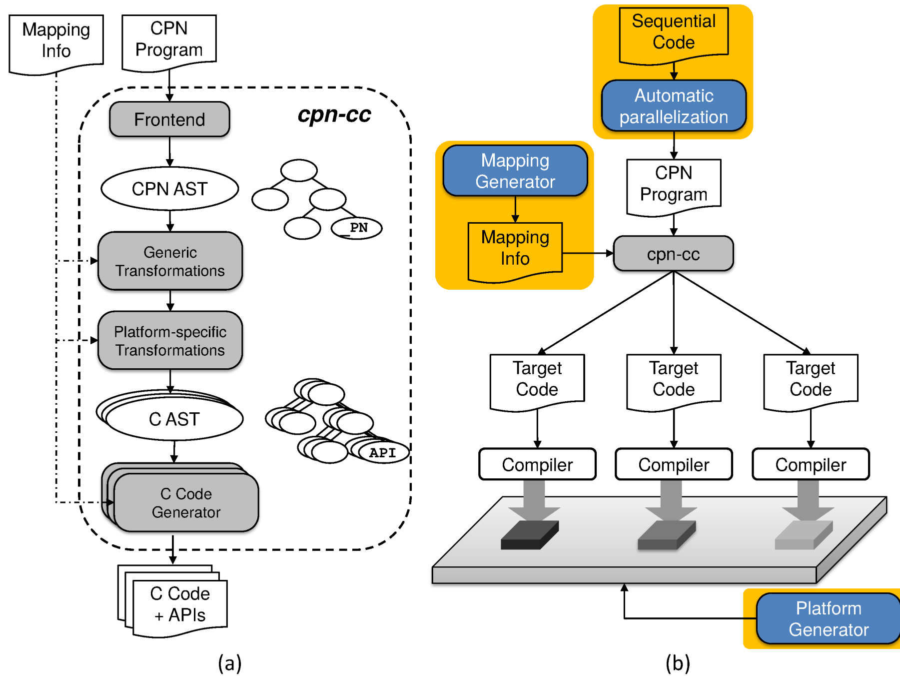

Figure 1.7 – (a) Structure of the source-to-source compiler cpn-cc (b) An example of a complete compiler framework for a three-processor heterogeneous MPSoC: the parts with a shaded background are optional to the core compiler framework.

of the frontend, generic and platform-specific transformations, and a C code generator. Those components and some special considerations during the implementation are described below:

– Frontend: The frontend starts with C preprocessors to provide the functionalities, such as the inclusion of header files. After preprocessing, the source code is processed by the tokenizer, to which new CPN keywords were added, and the parser, which was extended with new grammar rules for CPN syntax elements, such as process templates and channel accesses. Then, the CPN-aware semantic analysis builds the abstract syntax tree (AST) as an intermediate representation in the memory for further processing. It contains all CPN language elements and constructs occurring in the source code.

– AST Transformations: After preparation by the frontend, the AST transformations perform source-to-source translation. They are categorized into generic and platform-specific transformations, similar to the sequence of code optimizations in a classic uni-processor compiler (Aho *et al.* 2006). Generic transformations are platform-independent. One example is the transformation from SDF templates to

KPNs, whereby SDF process templates are rewritten into KPNs with an endless loop with explicit channel accesses in the behavior code. Platform-specific transformations replace the AST nodes of CPN constructs by C nodes representing platform-specific API calls, e.g. FIFO primitives. Therefore, the result of the AST transformations will be a plain C code AST without any of the extensions introduced by CPN (but with target-specific API calls). Several ASTs are built in this phase, one for each processor on the target MPSoC platform.

– C Code Generator: As the last step of source-to-source translation, the AST Clang printer generates the C source code from the ASTs after transformations. In some cases, other additional auxiliary files, such as configurations and makefiles, also need to be generated to be further compiled with the individual processor native C compilers.

Compared to other source-to-source translation tools that are not based on a formally built AST (e.g., textual replacements), our approach provides a cleaner and more powerful infrastructure for source code transformation based on the AST. The full semantic information allows code optimizations in a larger program context. For example, data type checking and variable manipulation can be carried out in the compilation while other textual replacement approaches cannot do this reliably. The readability of the generated source code is also retained and the structure is close to the original code. Compared to other approaches not based on an AST, more effort is needed to build a fully working compiler, however. The effort is well justified, in that cpn-cc is designed to be retargetable, and most cpn-cc components can be re-used for different target MPSoC platforms.

A complete compilation framework for heterogeneous MPSoCs can be built around the cpn-cc in a flexible and extensible manner to suit different MPSoC target platforms in different scenarios. An example of such a framework for a specific MPSoC is shown in Figure 1.7 (b):

– In addition to CPN programs that are considered as the functional specification, an important input provided by the user is the mapping info, which specifies the spatial mapping of processes to processing elements and temporal order for execution. The mapping info, together with the CPN program, determines the performance of the application on the target hardware platform. Unlike the monolithic approach, which tries to automatically compute the mapping and scheduling, we have designed the mapping info as an input to the core compiler in order to keep a clean and lean software architecture. The mapping info can be provided by the user manually or can be generated by an external tool.

– Transformations are implemented in a modular way, and can thus be customized for different target platforms in a plug-and-play fashion. A large portion of common transformations can be reused among many MPSoC targets, which significantly eases the retargeting process of the framework. Target specific transformations can be developed to better exploit the specific hardware details.

– The compilation framework can be used in collaboration with other state-of-the-art tools, thanks to the extensible design. For example, an intelligent mapping info generator, referred to as Mapping Generator in Figure 1.7 (b), can be easily integrated with the core compiler framework. In some embedded system design scenarios, the software development needs to proceed simultaneously with the architecture development, e.g., using a Platform Generator (see Figure 1.7). Our work also facilitates the retargeting of the compiler framework to work with the Electronic System Level (ESL) design tools for early system level design.

The main concept of this flexible and extensible design is to enable the re-use of components in constructing compiler frameworks for different MPSoC platforms. Therefore, retargetability can be achieved with minimal effort. Furthermore, the retargeting only needs to be done once for every new platform. As designing embedded systems today inevitably involves many other third-party and vendor tools, this design also makes it easy to collaborate with those tools, thus preserving existing software investments.

1.4. Software Distribution

The MAPS framework was built as an interactive tool for the programmer to modify and optimize the application within the IDE. To further aid the programmer, MAPS also includes a set of configurable algorithms to automate different steps of the optimization process. The most notable such a process is that of software distribution, which accounts to *mapping* computation (processes) to the heterogeneous cores of the platform and communication (channels) to communication resources (e.g., communication APIs, memories and interconnect). Several works address automatic software distribution in a similar way to MAPS, for performance, energy, thermal distribution among other optimization goals, e.g., (Thiele *et al*. 2007 ; Quan and Pimentel 2014 ; Hascoët *et al*. 2017 ; Marwedel *et al*. 2011 ; Das *et al*. 2015). Given the requirements imposed by the process networks programming model, MAPS rely on analysis of traces to determine static and hybrid mappings, as discussed in the following.

1.4.1. *KPN Analysis*

As discussed in section 1.2, static dataflow programming models like SDF expose the communication patterns in the access rates to channels. For this reason, most frameworks for the analysis of such graphs consider actors as black boxes. KPN applications does not do so, so that communication patterns need to be analyzed by looking in the actual implementation of the behavior of processes. Consider the example in Fig. 1.8. Given the hypothetical control flow of process $P1$, a tool can infer how many times the process writes to channel $C2$ before reading from channel $C1$ (determined by the variable `x` and the initial value of `i`). In MAPS, a *gray* model of

processes is obtained by analyzing the processes in isolation and collecting traces of channel access events as exemplified on the right of Fig. 1.8.

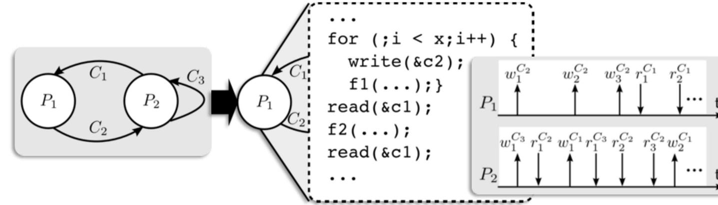

Figure 1.8 – Illustration of process traces. Example process network with sample code of the logic of process $P1$, alongside a hypothetical trace of read and write events that may result from the processes inner control flow.

Trace-based analysis is enabled by the deterministic nature of KPNs. That is, provided with the same input stimuli, every process will always produce the same history of tokens in the channels, independent of the schedule. MAPS leveraged this property to abstract processes as trains of events with annotated execution costs corresponding to the different core types of the platform (Castrillon *et al*. 2010). Technically, this was achieved via source code instrumentation and by using source code cost estimation tools (Gao *et al*. 2009 ; Eusse *et al*. 2014) whenever the target hardware was unavailable. Based on traces, algorithms could optimize the placement of events in the virtual execution time line on the target heterogeneous systems without changing the application semantics (cf. next section). Naturally, different input stimuli would create different traces, each of which would have to be analyzed independently. Some recent efforts have tried to jointly optimize for multiple execution traces of the same application by clustering process behaviors (Goens and Castrillon 2015). Results are so far inconclusive.

1.4.2. *Static KPN Mapping*

A static mapping is one that is computed at compile time and, typically, one in which compile-time decisions are not allowed to be overwritten at runtime. For the problem setup considered in MAPS, this boils down to (i) mapping computation, i.e., deciding which particular core executes which process, and (ii) mapping communication, i.e., deciding with which API over which hardware resources a channel is to be implemented. The latter one allocates memory for communication channels and thus includes the process of buffer sizing, i.e., deciding what bounds are to be set on the logical FIFO buffers to reduce the chances of encountering deadlocks and improving throughput.

Given the lack of information in KPN specification, most mapping approaches resort to metaheuristics, most prominently with Genetic Algorithms (Pimentel *et al*.

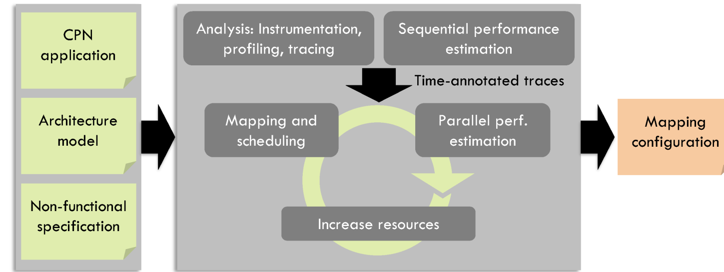

Figure 1.9 – Overview of the mapping flow.

2006 ; Nikolov 2009 ; Thiele *et al*. 2007). MAPS aimed at leveraging the information in the traces to devise faster heuristics to compute the mapping (Castrillon *et al*. 2013). The generic mapping flow within MAPS is shown in Fig. 1.9. After analysis and sequential performance estimation, an iterative process is started that computes static mappings on an increasing set of allowed resources. By adding resources (cores and memory space), the tool can expose different trade-offs to the programmer and also attempt to find the minimal resource configuration that meets application constraints. Internally, each mapping configuration is simulated in a parallel performance estimation that uses the traces collected during analysis to emulate how the target system would execute the application.

Different heuristics are included in MAPS to compute the mapping of the application given a set of resources. Among them, Group-Based Mapping (GBM) is worth highlighting (Castrillon *et al*. 2012). This heuristic works on a graph representation of the traces, where dependencies between events are explicitly modeled with edges. Dependencies include traditional read-after-write dependencies which model the blocking read semantics of KPN applications. In the example trace in Fig. 1.8, the event $r_1^{C_2}$ of $P2$ can only succeed after process $P1$ has executed the corresponding write event ($w_1^{C_2}$). Blocking write semantics due to finite buffer sizes are also modeled via similar dependencies. In the example, assuming a buffer size of 2 for channel $C2$, would mean that the third write to the channel ($w_3^{C_2}$) can only execute after the consumer process $P2$ has read twice. More complex dependencies, e.g., supporting events that write multiple tokens at once and more complex communication protocols are also supported (Odendahl *et al*. 2013). GBM uses this graph representation to compute mappings of traces to cores in the platform in a similar way to other heuristics for directed acyclic graphs. The key idea of GBM is to iteratively compute mappings of groups of application elements (processes or channels) to groups of platform resources (cores or memories). The priority in which groups are created and mapped is determined by the equivalent of the critical path, considering all possible mappings. By mapping to groups, the heterogeneity of the platform is directly addressed. Concrete resources are decided in a final step. For further details, the reader is referred

to (Castrillon *et al*. 2012). For an analysis and comparison of heuristics and genetic algorithms, the reader is referred to (Goens *et al*. 2016).

1.4.3. *Hybrid KPN Mapping*

With embedded systems becoming more adaptive and running multiple workloads at once, fixed mapping approaches became outdated. Hybrid approaches refer to attempt to make mappings more flexible but still retain properties computed at compile time (predictable performance and energy consumption). Typically this consists in producing multiple mapping configurations with different resource utilization at compile time and letting the runtime decide which variant to deploy (Schor *et al*. 2012 ; Quan and Pimentel 2013). In (Quan and Pimentel 2014) for example, the authors propose a method in which pareto-optimal points are computed at compile-time for different configurations, with an adaptable runtime presented in (Quan and Pimentel 2016). These configurations are switched at runtime according to the given set of active applications. A similar strategy is used in (Weichslgartner *et al*. 2016 ; Schwarzer *et al*. 2018).

We have worked on identifying application and architecture symmetries that allow defining mapping equivalent classes (Goens, Siccha and Castrillon 2017). Symmetries enable transformations by the compiler or the runtime that preserve design tradeoffs, so that properties predicted at compile time are preserved at runtime. By making the runtime system aware of such transformations we demonstrated better time predictability on off-the-shelf multi-core systems (Goens, Khasanov, Hähnel, Smejkal, Härtig and Castrillon 2017). Our runtime featured up to 510× less performance variability and 83× less energy variability when compared to the default Linux scheduler on an Odroid XU4. We are currently working on joint application optimization, proactively looking into future workload change (Khasanov and Castrillon 2020).

1.5. Results

A tooling infrastructure must be validated in real use cases, which are presented below as experimental results. We have evaluated this design in:

– illustrative uses of MAPS for real-word MPSoC platforms and other scenarios; and

– how retargetability of MAPS for different MPSoC platforms is achieved.

1.5.1. *Applications and experiences*

MAPS, as a MPSoC compiler infrastructure, has been applied and retargeted to a number of MPSoC platforms to demonstrate its feasibility and value in automating the

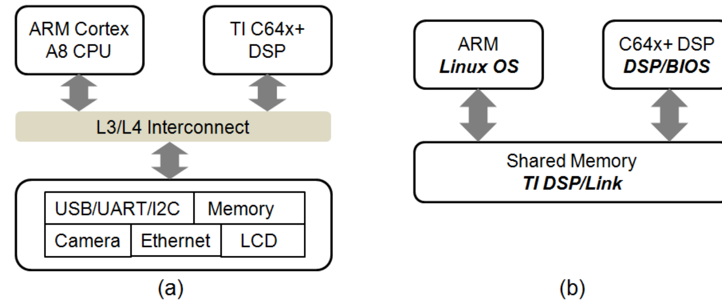

Figure 1.10 – (a) OMAP3530 Block Diagram (b) Overview of TI OMAP Software

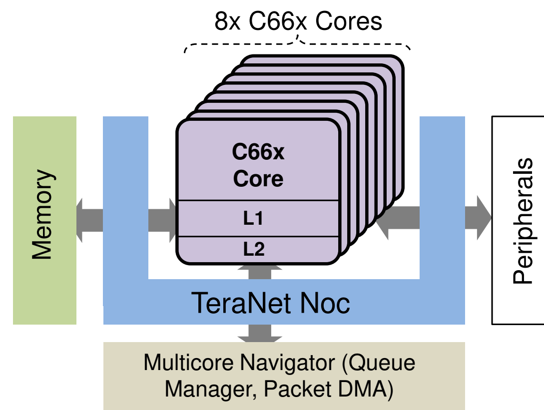

Figure 1.11 – TI C6678 block diagram

compilation process. Those platforms include commercial platforms that are available in silicon such as TI OMAP3530 and TI C6678. Virtual prototypes of MPSoC platforms (e.g. Synopsys MCO (Schirrmeister and Sheridan 2011)) are also supported, and other multicore platforms (mostly x86 or ARM based) where a Pthreads environment is available. First, we briefly describe those platforms below.

– TI OMAP3530 was announced as a multimedia processor by TI in 2009, and is shown in Fig. 1.10 (a). It consists of two processors, an ARM Cortex A8 CPU running at 550 MHz and a TI C64x DSP clocked at 400 MHz. The hardware evaluation board used in this work possesses 128 Mbyte DDR SDRAM at 166MHz shared between the ARM and the DSP. Software-wise, Linux is used as OS on the ARM side. The Pthread library and the POSIX IPC (Inter-Process Communication) are available for realizing the thread management and communications. On the DSP side, a proprietary lightweight multitasking operating system, called DSP/BIOS (Dart 2001), is provided by

TI. The communication between ARM and DSP is handled by TI's DSP/Link layer. Fig. 1.10(b) gives an overview of the software stack for the OMAP platform. General TI OMAP programming strategies are discussed in (Kloss 2003). The rule of thumb is to map signal processing tasks to DSP and control flow processing to ARM. However, no systematic methodology or tooling support is available.

– TI C6678 (Texas Instruments 2011) is based on TI's KeyStone I multicore architecture. It integrates eight C66x DSPs shown in Fig. 1.11. Besides the local L1/L2 memory for each DSP and global shared memory, the C6678 platform is featured with the Multicore Navigator that consists of more than eight thousand queues for direct communications between processor cores. This feature is particularly desirable for streaming programs. A software development toolchain (compiler, linker, etc.) is available for the DSP from TI. However, no platform-wide compilation framework is available.

– Synopsys MCO (MultiCore Optimization) technology is a SystemC-based virtual platform solution for hardware/software co-development in the early system design phase. Multicore platforms can be modeled using a high-level abstraction of a processing element, named Virtual Processing Unit (VPU) (Kempf *et al*. 2005). The VPUs on the platform are able to execute SystemC modules that model application tasks. Those tasks contain C code that can be extended with explicit timing annotations and with communication directives to access the communication infrastructure of the modeled hardware platform. An automatic compilation framework is desired for multicore platforms in MCO to enable seamless hardware/software co-development. MAPS has been retargeted to support MCO so that system architects can quickly evaluate software design choices on a high-level virtual platform.

– Pthreads is a widely known parallel programming API and is widely available on multicore platforms (e.g., x86 or ARM-based). MAPS has been retargeted to generate streaming programs using the Pthreads API to realize concurrent processes with FIFO communications. This expands the platforms that MAPS supports to virtually all platforms that are capable of running Pthreads. Alternatively, this option is also valuable when programmers perform functional verification on the host.

We have successfully applied the MAPS compiler infrastructure for these MPSoC platforms. Although there exist some differences in different MAPS instances for these platforms, the basic principle and flow apply to all. As an example, the case of using MAPS for the TI OMAP3530 is introduced below, as shown in Fig. 1.12. The example from Listing 4 is used as an input program for the cpn-cc compiler. A mapping info file is required for cpn-cc to perform source-to-source compilation, e.g., the spatial mapping of processes to processors of OMAP3530. The cpn-cc compiler builds and transforms the AST to replace `__PN` nodes by OMAP3530 specific APIs. At the right side of Fig. 1.12, a simplified code excerpt of the transformation results for FIFO channel accesses is shown. The generated code is both editable and readable, thus creating opportunities for performance fine-tuning by programmers. It is

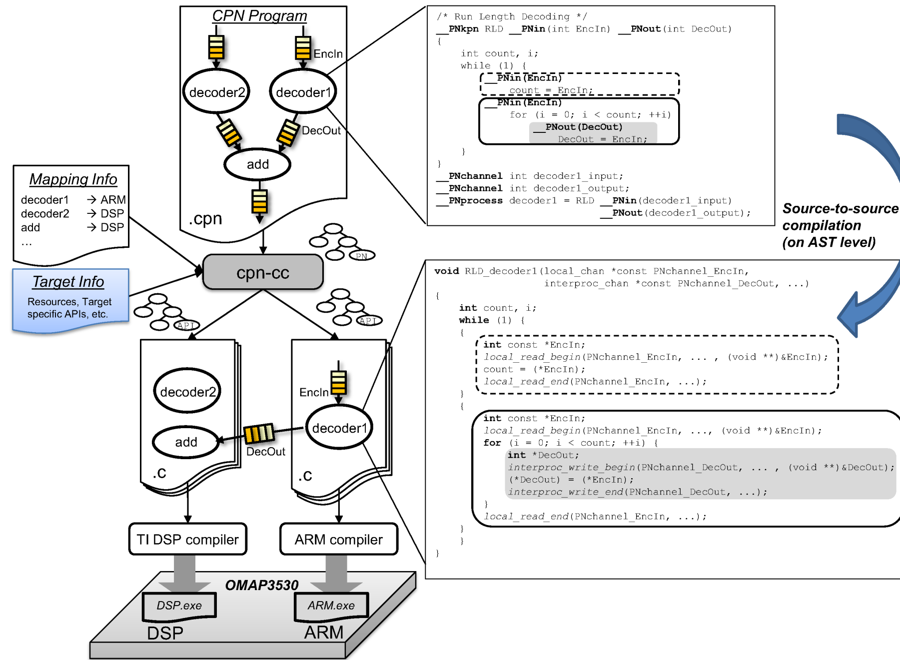

Figure 1.12 – A complete compilation framework for OMAP3530 using MAPS

important to note that it is critical that all transformations take place at the AST level that provides the complete semantic contents of the source code. For instance, the channel $DecOut$ is a FIFO communication that occurs between different processors under this mapping, while the channel $EncIn$ communication is local. This information must be determined by the compiler in order to select the correct target-specific API: in this case, *interproc_* functions for inter-processor FIFOs and *local_* for local FIFOs. Simple textual replacement techniques are insufficient in this case. After the source-to-source compilation, target specific C files are generated for the ARM and the DSP and the existing vendor toolchains are re-used. The compilation flow is thus fully automated.

Two case studies of applications are presented here by using MAPS on the TI multi-DSP platform C6678 for completeness. Two signal processing benchmarks were selected: a digital audio filter and an airborne radar application.

The digital audio application implements fast convolution filtering using the Fast Fourier Transform (FFT) and inverse FFT and it performs low-pass filtering on a stereo audio signal. Two parallelized versions of the application are written in CPN programming model to explore parallelism: (a) exploiting parallel processing on left and

right channel of the input signal simultaneously; (b) further parallelization by splitting 1024-point Fourier transforms into two 512-point blocks. Fig. 1.13 shows graphical representations of both versions of the digital filter. The benchmark existed initially as sequential C code. The conversion from sequential C code to parallel CPN versions took around half a day to complete, thanks to the CPN being close to C. As the CPN language separates the functional specification of processes and overall topology specification, the two versions differ only by a few lines, in that the further parallelization mainly modifies the topology.

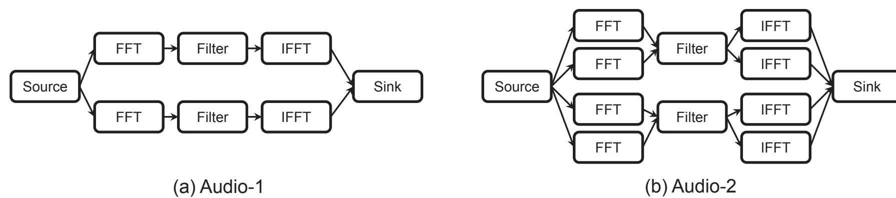

Figure 1.13 – Benchmark: digital audio filter

To evaluate benchmarks on the C6678 platform, we used a mapping heuristic, Group-Based Mapping (GBM), explained in section 1.4, to generate spatial mapping for benchmarks. The GBM takes constraints, e.g., available number of processors as inputs to compute mapping. Fig. 1.14 shows the performance results when the constraint on the number of available processors increases from 1 to 8 for the audio filter. Each data point corresponds to the runtime result of the generated mapping, which is measured on the hardware board. It is evident that the speed-ups achieved increase when the number of available processors rises. The higher number of parallel processes in Audio-2 gives better results than Audio-1, in that 8 cores on the C6678 platform can be better utilized by Audio-2.

The second is a radar application to detect moving objects on the ground from the air by sending periodic radar pulses. The benchmark was also initially available as sequential C code. The first conversion into a CPN program took half a day. Fig. 1.15 shows the graphical representation of the first version. The topology of the parallelized version basically follows the algorithmic division of the application. Similarly, we performed the mapping step to C6678 platform, and the results are shown in the Fig. 1.15(a). It is clear that the performance of the first process network version of the radar application did not scale well beyond 4 cores. The speedup stayed about the same from 4 cores to 8 cores, which indicated that the efficiency of the parallel platform usage actually decreased.

We have investigated this saturation, and found the reason to be that the original algorithmic division of the application led to a process network topology that has

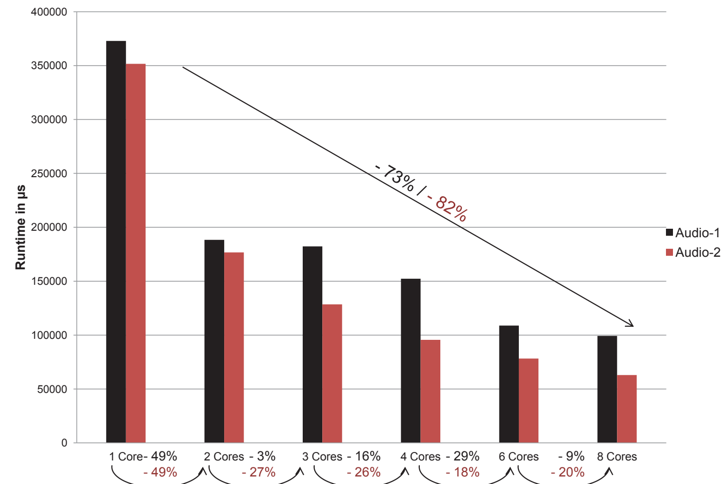

Figure 1.14 – Results of the audio filter (the percentage indicates the reduction of runtime between two mappings)

large deviance in the individual process runtime. This indicates that some process took a long time to finish, while others needed a relatively short time. In particular, the processes Calculate Filter and Apply Filter (shown in gray in Fig. 1.15) are the main number-crunching tasks in this application. This imbalance led to an inefficient usage of the available cores on the C6678 platform.

Similar problems often arise in the process of refining the algorithmic specification for optimized performance towards a particular parallel architecture. We attempted to solve this by merging the tasks that require a short time to finish and splitting the long-running processes into small ones. The processes with the same boxing style in Fig. 1.15 were merged into one process, e.g. Pulse Compression, Reordering2, Reordering1 and Calculate Steer Vectors. The processes, Calculate Filter and Apply Filter which has data-level parallelism, are split into four smaller processes. The changes were implemented using the CPN language with very little effort. The improved process network topology is shown in the Fig. 1.17(a). We then used the same mapping heuristic to generate the spatial mapping for the improved radar application. Fig. 1.16(b) showed that performance figures such as the speedup and efficiency are much better than those from the initial version. Fig. 1.17(b)-(d) plotted the detailed spatial mapping results for two, four and eight cores, respectively. The process merging and splitting allowed further exploitation of the computing resources of the C6678 platform.

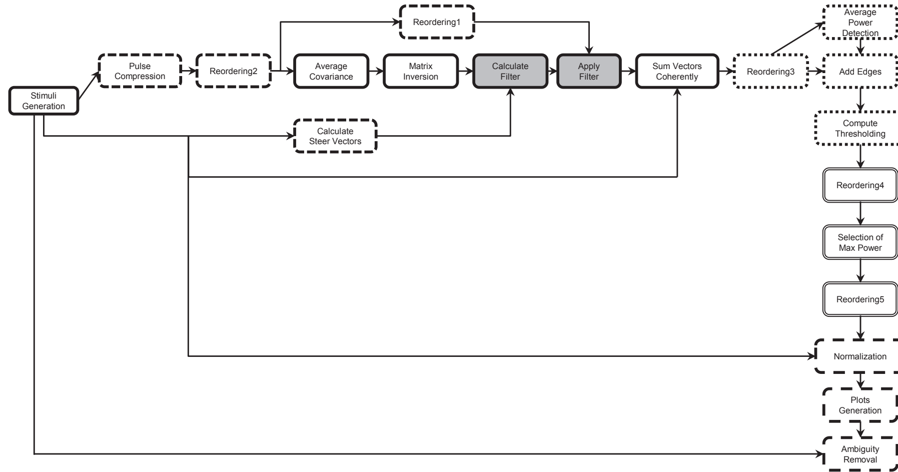

Figure 1.15 – Benchmark: radar application

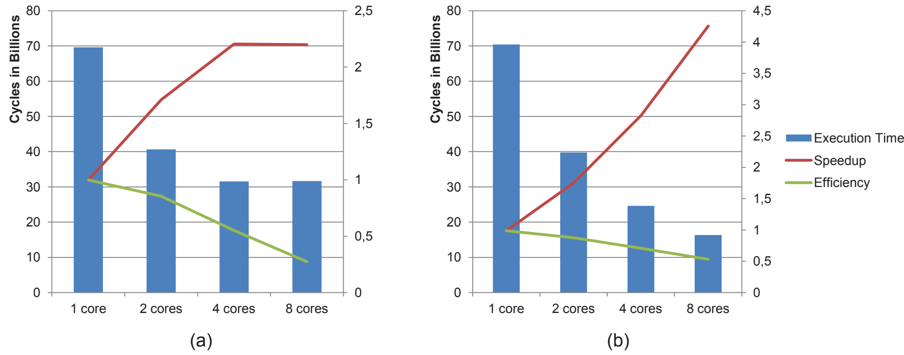

Figure 1.16 – Results of the radar applications: Performance (a) initial version; (b) improved version. $Speedup = \frac{Execution\ \ Time}{Sequential\ \ Time}$; $Efficiency = \frac{Speedup}{Number\ of\ cores\ used}$

The two case studies represented typical design space exploration process of parallelizing applications for multicore platforms. The CPN language not only eases the initial conversion from sequential code into parallelized form but also facilitates optimizing the process networks incrementally. Software developers' productivity is also significantly enhanced as the compilation process is automated. The first case study (digital audio filter) took around one person-day to finish parallelization and mapping

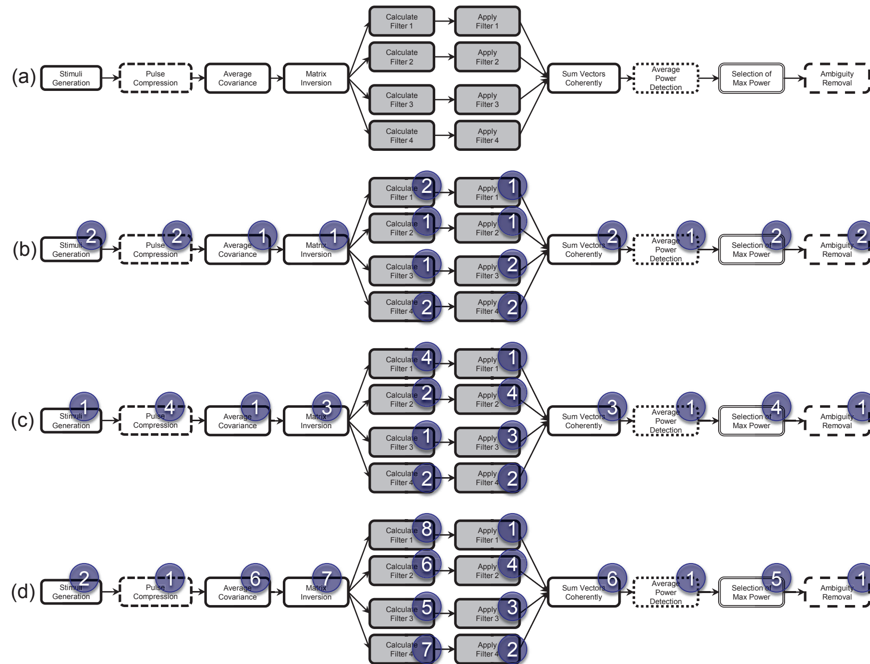

Figure 1.17 – Mapping results of the radar application: improved version. (a) process network topology (only edges that communicate major data traffic are plotted for the sake of clarity; (b) mapping to two cores; (c) mapping to four cores; (d) mapping to eight cores.

to the TI platform for each parallelized version. As a comparison, we have also attempted the same procedure for the digital audio filter application manually using the native tools. It cost a week or so to evaluate just one data point (two cores) in Fig. 1.14. The majority of the overhead spent was on the debugging effort. We have estimated that for the audio filter case the productivity improvement is around 30x – 150x, subject to the programmer's experience level. The more complex the applications are, the more productivity the programmer will gain. Last but not at least, programs written in CPN are also portable to other MPSoC platforms.

1.5.2. *Retargetability*

One of the main parameters of different MAPS instances is the so-called target info shown on the left of Fig. 1.12. It contains the target specific information such as available resources (processors, inter-communication schemes) and API calls, which needs to be known to the source-to-source compiler. For retargeting MAPS towards

different MPSoC platforms, it is essential to update this information. Our experience on the retargetability of the MAPS infrastructure is reported in this section.

Table 1.1 summarizes the retargeting process that was carried out towards the reported MPSoC platforms. The main types of information required for retargeting are:

– Platform details: this includes how many programmable processors the target platform has. This reveals where it is possible to simultaneously execute concurrent processes to the compiler.

– Multi-tasking Runtime: this indicates how the run-time environments of processors of the target MPSoC support concurrent processes. Often in cases such as that of an OS like Linux or proprietary ones running on a processor, the OS has multi-tasking APIs to manage concurrent tasks (or processes).

– Communication: processes of streaming applications run simultaneously while communicating with one another. Therefore, it is necessary for the compiler to know the possible ways in which communication between processes may be realized. This also includes the type (inter- or intra- processor) and specific API calls.

	TI OMAP3530	TI C6678	Synopsys MCO	Pthreads
Platform details	ARM + DSP	Nx DSPs	Nx VPUs	Multicores supporting Pthreads
Multi-tasking Runtime	OS thread (ARM and DSP)	OS thread	MCO task modules	OS thread
Communication	shared memory	shared memory or message-passing	shared memory or message-passing	shared-memory
Retargeting effort	$20d$	$20d$	$10d$	$5d$

Table 1.1 – Retargeting MAPS towards MPSoC platforms

The actual retargeting process consists of utilizing this target-specific information in various locations within the MAPS infrastructure. The effort that we invested in supporting these platforms is reported in Table 1.1. It ranges from 20 person-days to 5 person-days, depending on the complexity of target platforms. The time also includes the initial time required to learn the platform details. This effort is acceptable and justified by the increase in the automation level of the compilation process.

1.6. Summary

More than a decade ago, major computing processor manufacturers began to integrate multiple (simple) cores into a single chip, namely multicores, to maintain scaling according to Moore's law. While the transition from scalar (uni)processors to multicores is something of an evolutionary step in terms of hardware, it has instigated fundamental changes in software development. Things got even worse when requirements are so different from the HPC to embedded systems and more and more heterogeneity

has been seen in the hardware. As commonly agreed, the software development process for multicores has to be revolutionized. Though many research efforts have been made to provide solutions for developing the multicore software, lacking practical tooling support has seriously jeopardized maximizing the potential of multicores.

This work describes our systematic approach to tackling the multicore programming challenge from a practical perspective. A compiler infrastructure for heterogeneous embedded multicores has been developed. As proof and validation of this work, the tooling infrastructure has been applied successfully in many academic and industrial case studies of multicore design and development. A clean, light-weight C language extension called CPN (C for Process Networks) is developed to capture streaming models which are common in embedded applications. It is designed to keep the syntax as close to C as possible while making process networks structured and readable. A minimum set of new keywords is added to the C language to describe concurrent processes and channels which act as the communication between processes. A source-to-source compiler, cpn-cc, was developed as the core component for a multicore compiler framework. The implementation of cpn-cc is based on the Clang/LLVM by mainly using the AST transformations in the frontend. Unlike the compilers for scalar processors, the cpn-cc does not only need CPN programs as input but also a mapping info that specifies the spatial and temporal mapping of processes to processing elements available in the target multicore platform. The cpn-cc and surrounding software components in the framework are made to be extensible and customizable to suit different requirements in the multicore design practice. Several real-world multicore platforms, such as TI OMAP 3530 and TI C6678 as well as system level virtual platforms like Synopsys MCO, have been used as target platforms successfully. The results of mapping many benchmarks onto multicore hardware are presented. The applications have achieved good speed-ups and software development productivity has also been greatly improved. In addition, advanced use cases, such as automatic calibration of streaming applications for software mapping exploration and the legacy software migration for a tablet (Sheng *et al*. 2013) have demonstrated the versatility of the multicore compiler infrastructure developed in this work (Sheng *et al*. 2014). Finally, we discussed the state-of-the-art in software distribution, detailing the mapping flow of the MAPS framework and providing insight into current research directions.

1.7. Bibliography

Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D. (2006), *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Prentice Hall.

A.T. Kearney (2010), ‘A Viable Future Model for the Internet’.

Bamakhrama, M. A., Zhai, J. T., Nikolov, H., Stefanov, T. (2012), A Methodology for Automated Design of Hard-Real-Time Embedded Streaming Systems, *in* DATE 2012: Proceedings of the 15th Design, Automation, and Test in Europe conference.

Bhattacharyya, S. S., Deprettere, E. F., Leupers, R., Takala, J. (2010), *"Handbook of Signal Processing Systems"*, Springer.

Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J. (1996), Cyclo-static Dataflow, *Signal Processing, IEEE Transactions on*, 44(2), 397 –408.

Brunet, S. C. (2015), Analysis and optimization of dynamic dataflow programs, PhD thesis, Ecole Polytechnique Federale de Lausanne (EPFL).

Castrillon, J., Leupers, R., Ascheid, G. (2013), MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs, *IEEE Transactions on Industrial Informatics*, 9(1), 527–545.

Castrillon, J., Sheng, W., Leupers, R. (2011), Trends in embedded software synthesis, *in* Proceedings of the International Conference Embedded Computer Systems: Architecture, Modeling and Simulation (SAMOS), 2011, IEEE, pp. 347–354.

Castrillon, J., Tretter, A., Leupers, R., Ascheid, G. (2012), Communication-Aware Mapping of KPN Applications onto Heterogeneous MPSoCs, *in* DAC.

Castrillon, J., Velásquez, R., Stulova, A., Sheng, W., Ceng, J., Leupers, R., Ascheid, G., Meyr, H. (2010), Trace-based kpn composability analysis for mapping simultaneous applications to mpsoc platforms, *in* Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 753–758.
URL: *http://dl.acm.org/citation.cfm?id=1870926.1871107*

Castrillon Mazo, J. (2013), Programming heterogeneous MPSoCs: tool flows to close the software productivity gap, PhD thesis.
URL: *http://publications.rwth-aachen.de/record/211242/files/4534.pdf*

Cisco Systems (2017), 'Cisco Visual Networking Index: Global Mobile 2017'.

Dart, D. (2001), DSP/BIOS Kernel Technical Overview, *Texas Instruments Application Report*, (SPRA780).

Das, A., Singh, A. K., Kumar, A. (2015), Execution trace–driven energy-reliability optimization for multimedia mpsocs, *ACM Trans. Reconfigurable Technol. Syst.*, 8(3), 18:1–18:19.

Eker, J., Janneck, J. W. (2003), CAL Language Report Specification of the CAL Actor Language, Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley.
URL: *http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html*

Eusse, J. F., Williams, C., Leupers, R. (2014), Coex: A novel profiling-based algorithm/architecture co-exploration for asip design, *ACM Transactions on Reconfigurable Technology and Systems*, .

Gao, L., Huang, J., Ceng, J., Leupers, R., Ascheid, G., Meyr, H. (2009), 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, ACM, New York, NY, USA, pp. 305 – 314.

Giannopoulou, G., Poplavko, P., Socci, D., Huang, P., Stoimenov, N., Bourgos, P., Thiele, L., Bozga, M., Bensalem, S., Girbal, S., Faugere, M., Soulat, R., de Dinechin, B. D. (2016), Dol-bip-critical: A tool chain for rigorous design and implementation of mixed-criticality multi-core systems, Technical Report 363, ETH Zurich, Laboratory TIK.

Gilles Kahn (1974), The Semantics of a Simple Language for Parallel Programming, *in* IFIP Congress 74, North Holland, Amsterdam, pp. 471–475.

Goens, A., Castrillon, J. (2015), Analysis of process traces for mapping dynamic kpn applications to mpsocs, *in* M. Götz, G. Schirner, M. A. Wehrmeister, M. A. Al Faruque, A. Rettberg, (eds), System Level Design from HW/SW to Memory for Embedded Systems. IESS 2015. IFIP Advances in Information and Communication Technology, vol 523, Springer International Publishing, Foz do Iguaçu, Brazil, pp. 116–127.

Goens, A., Khasanov, R., Castrillon, J., Polstra, S., Pimentel, A. (2016), Why comparing system-level MPSoC mapping approaches is difficult: a case study, *in* Proceedings of the IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16), Ecole Centrale de Lyon, Lyon, France, pp. 281–288.

Goens, A., Khasanov, R., Hähnel, M., Smejkal, T., Härtig, H., Castrillon, J. (2017), Tetris: a multi-application run-time system for predictable execution of static mappings, *in* Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES'17), SCOPES '17, ACM, New York, NY, USA, pp. 11–20.
URL: *http://doi.acm.org/10.1145/3078659.3078663*

Goens, A., Siccha, S., Castrillon, J. (2017), Symmetry in software synthesis, *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2), 20:1–20:26.
URL: *http://doi.acm.org/10.1145/3095747*

Hascoët, J., Desnos, K., Nezan, J., de Dinechin, B. D. (2017), Hierarchical dataflow model for efficient programming of clustered manycore processors, *in* 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 137–142.

Haubelt, C., Falk, J., Keinert, J., Schlichter, T., Streubühr, M., Deyhle, A., Hadert, A., Teich, J. (2007), A SystemC-based Design Methodology for Digital Signal Processing Systems, *EURASIP J. Embedded Syst.*, 2007(1), 22 p.

International Technology Roadmap for Semiconductors (2011), System Drivers, *International Technology Roadmap for Semiconductors 2011 Edition*, .

Keinert, J., Streubühr, M., Schlichter, T., Falk, J., Gladigau, J., Haubelt, C., Teich, J., Meredith, M. (2009), SystemCoDesigner – an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications, *ACM Trans. Des. Autom. Electron. Syst.*, 14, 1:1–1:23.
URL: *http://doi.acm.org/10.1145/1455229.1455230*

Kempf, T., Doerper, M., Leupers, R., Ascheid, G., Meyr, H., Kogel, T., Vanthournout, B. (2005), A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms, *in* Proceedings of the conference on Design, Automation and Test in Europe, pp. 876–881.

Khasanov, R., Castrillon, J. (2020), Energy-efficient runtime resource management for adaptable multi-application mapping, *in* Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE), DATE '20, EDA Consortium.

Khasanov, R., Goens, A., Castrillon, J. (2018), Implicit data-parallelism in kahn process networks: Bridging the macqueen gap, *in* Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM'18), co-located with 13th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC), PARMA-DITAM '18, ACM, New York, NY, USA, pp. 20–25.
URL: *http://doi.acm.org/10.1145/3183767.3183790*

Kloss, N. (2003), Application Programming Strategies for TI's OMAP Solutions, *Embedded Edge*, .

Lattner, C. (2008), 'LLVM and Clang: Next generation compiler technology', The BSD Conference, Ottawa, Canada.

Lee, E. A., Messerschmitt, D. G. (1987), Synchronous Data Flow, *in* Proceeding of the IEEE, vol. 75.

Leupers, R., Aguilar, M. A., Eusse, J. F., Castrillon, J., Sheng, W. (2017), *MAPS: A Software Development Environment for Embedded Multicore Applications*, Springer Netherlands, pp. 1–33.

Leupers, R., Castrillon, J. (2010), MPSoC programming using the MAPS compiler, *in* Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific, pp. 897 –902.

Marwedel, P., Bacivarov, I., Lee, C., Teich, J., Thiele, L., Xu, Q., Kouveli, G., Ha, S., Huang, L. (2011), Mapping of Applications to MPSoCs, *in* Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on, pp. 109 –118.

Nikolov, H. (2009), System-Level Design Methodology for Streaming Multiprocessor Embedded Systems, PhD thesis, Universiteit Leiden.

Odendahl, M., Castrillon, J., Volevach, V., Leupers, R., Ascheid, G. (2013), Split-cost communication model for improved mpsoc application mapping, *in* Proceedings of the International Symposium on System on Chip (SoC), 2013, IEEE, pp. 1–8.

Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.-F., Aridhi, S. (2014), Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming, *in* 2014 6th european embedded design in education and research conference (EDERC), IEEE, pp. 36–40.

Pimentel, A. D., Erbas, C., Polstra, S. (2006), A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels, *IEEE Transactions on Computers*, 55(2), 99–112.

Ptolemaeus, C., (ed.) (2014), *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org.
URL: *http://ptolemy.org/books/Systems*

Quan, W., Pimentel, A. D. (2013), A scenario-based run-time task mapping algorithm for mpsocs, *in* Proceedings of the 50th Annual Design Automation Conference, DAC '13, ACM, New York, NY, USA, pp. 131:1–131:6.
URL: *http://doi.acm.org/10.1145/2463209.2488895*

Quan, W., Pimentel, A. D. (2014), Towards exploring vast mpsoc mapping design spaces using a bias-elitist evolutionary approach, *in* DSD 2014.

Quan, W., Pimentel, A. D. (2016), Scenario-based run-time adaptive mpsoc systems, *Journal of Systems Architecture*, 62, 12–23.

Schirrmeister, F., Sheridan, P. (2011), Optimizing Multicore System Performance, *Synopsys Insight, Issue 1*, .

Schor, L., Bacivarov, I., Rai, D., Yang, H., Kang, S.-H., Thiele, L. (2012), Scenario-based design flow for mapping streaming applications onto on-chip many-core systems, *in* Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '12, ACM, New York, NY, USA, pp. 71–80.
URL: *http://doi.acm.org/10.1145/2380403.2380422*

Schwarzer, T., Roloff, S., Richthammer, V., Khaldi, R., Wildermann, S., Glass, M., Teich, J. (2018), On the complexity of mapping feasibility in many-core architectures, *in* 2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp. 176–183.

Sheng, W., SchÃ¼rmans, S., Odendahl, M., Bertsch, M., Volevach, V., Leupers, R., Ascheid, G. (2014), A compiler infrastructure for embedded heterogeneous mpsocs, *in* Parallel Computing, vol. 40, Elsevier, pp. 51–68.

Sheng, W., Szymanski, P., Leupers, R., Ascheid, G. (2013), Software migration for parallel execution on a multicore tablet: A case study, *in* IEEE 7th International Symposium on Embedded Multicore SoCs (MCSoC-13).

Sriram, S., Bhattacharyya, S. S. (2009), *Embedded Multiprocessors: Scheduling and Synchronization*, 2 edition, Marcel Dekker, Inc., New York, NY, USA.

Stuijk, S., Geilen, M., Theelen, B., Basten, T. (2011), Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications, *in* Embedded Computer Systems (SAMOS), 2011 International Conference on, pp. 404–411.

Texas Instruments (2011), 'C6678 Multicore Fixed and Floating-Point Digital Signal Processor', [Online] Available `http://ti.com/product/tms320c6678#technicaldocuments/` (accessed 06/2012).

Thiele, L., Bacivarov, I., Haid, W., Huang, K. (2007), Mapping Applications to Tiled Multiprocessor Embedded Systems, *in* ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design, IEEE Computer Society, Washington, DC, USA, pp. 29–40.

van Berkel, C. H. K. (2009), Multi-core for mobile phones, *in* Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 1260–1265.

Weichslgartner, A., Wildermann, S., Götzfried, J., Freiling, F., Glaß, M., Teich, J. (2016), Design-time/run-time mapping of security-critical applications in heterogeneous mpsocs, *in* Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPES '16, ACM, New York, NY, USA, pp. 153–162.
URL: *http://doi.acm.org/10.1145/2906363.2906370*

Wiggers, M. H. (2009), Aperiodic Multiprocessor Scheduling for Real-Time Stream Processing Applications, PhD thesis, University of Twente. 978-90-365-2850-4.