A Compiler Infrastructure for Embedded Heterogeneous MPSoCs

Weihua Sheng, Stefan Schürmans, Maximilian Odendahl,
Mark Bertsch, Vitaliy Volevach, Rainer Leupers and Gerd Ascheid
Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany
sheng@ice.rwth-aachen.de

ABSTRACT

Programming heterogeneous MPSoCs (Multi-Processor Systems on Chip) is a grand challenge for embedded SoC providers and users today. In this paper, we argue for the need and significance of positioning the language and tool design from the perspective of *practicality* to address this challenge. We motivate, describe and justify such a practical design of a compilation framework for heterogeneous MPSoCs targeting the domain of streaming applications, named MAPS (MPSoC Application Programming Studio). MAPS defines a clean, light-weight C language extension to capture streaming programming models. A retargetable source-to-source compiler is developed to provide key capabilities to construct practical compilation frameworks for real-world, complex MPSoC platforms. Our results have shown that MAPS is a promising compiler infrastructure that enables programming of heterogeneous MPSoCs and increases productivity of MPSoC software developers.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers; D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages; D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Design, Languages

Keywords

MPSoC programming, compiler infrastructure

1. INTRODUCTION

Heterogeneous MPSoCs are widely used in modern embedded systems such as wireless terminals and modems. The hardware integration level of commercial MPSoC platforms increases so rapidly that a single chip consisting of multiple programmable processors of different types becomes reality. Platforms such as OMAP [26], KeyStone [27] from TI (Texas Instruments) and P2012 [3] from STMicroelectronics are state-of-the-art examples of such MPSoCs. They provide higher performance and lower power consumption than previous uni-processor systems. However, programming such MPSoC platforms remains a grand challenge for embedded SoC providers and users today [20].

PMAM'2013 February 23, 2013, Shenzhen [Guangdong, China]

1.1 Embedded MPSoC Programming

This work focuses on the challenge of MPSoC programming particularly in the *embedded systems industry*. Programmability issues for multi-processor systems are not new and have been studied intensively in the area of HPC (High Performance Computing) systems. Despite the fact that the problem setting is very similar, manifold differences exist between industries:

- Embedded systems are highly specialized in terms of functionality, integration and energy efficiency. A much higher level of *disaggregation* in embedded electronics industry exists where a large number of competent IP vendors provide solutions in specialized areas, e.g. wireless modems and video decoders. Due to the dramatic increase in engineering costs and complexity of today's electronic designs, the embedded industry favors a horizontal business model where system companies build heterogeneous MPSoC based products consisting of many processors from different, individual IP vendors. The IP vendors usually have proprietary programming tool-chains (e.g. C compilers) for their products. This differs from the vertical model in HPC industry where major providers maintain control of the product development cycle ranging from hardware to software tool-chains.
- The life cycle for embedded products is typically shorter than for HPC systems. The pressure of *reducing time-to-market and time-in-market* forces companies to frequently evolve MPSoC platforms by incremental changes for new product generations. This requires that the compiler framework must be quickly *adaptable* and *retargetable* within a short time frame.

As a result of both factors, during the process of technical developments towards multicores in the embedded industry, the advancement in the hardware community is unfortunately not in sync with its counterpart in the software support and always occurs ahead. The MPSoC programmability support is treated as an *afterthought*.

1.2 Current Practice

To illustrate the problem of MPSoC programming, Fig. 1 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

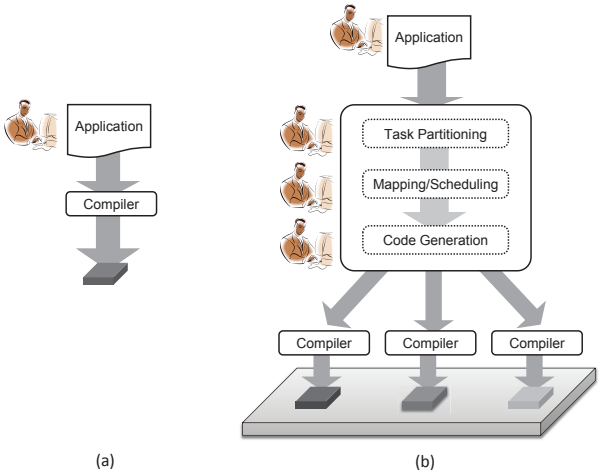


Figure 1: Programming Flow (a) *Uni-processor*: Compilers perform an end-to-end translation from parsing source code, target independent optimization, target dependent optimization to final binary generation. (b) *MPSoC*: No compilation framework for MPSoCs is in place. The process is largely manual.

being the most popular language) and rely on the *compilers* to generate target-specific code correctly and optimally, as shown in Fig. 1 (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 for the past few decades before the introduction of MPSoCs due to the pivotal role of compilers.

However, the traditional compiler technology does not scale for MPSoCs. Fig. 1 (b) demonstrates the current problematic programming flow for MPSoCs, which results in a low software productivity. Applications firstly need to be partitioned into parallel tasks, followed by the step of spatial and temporal mapping of those partitioned tasks onto MPSoC processing elements. As explained earlier, the programmable processors in heterogeneous MPSoCs nowadays often come with their own compilers and have own software stacks (API, OS). Therefore, after partitioning and mapping, correct code must be generated for those individual processors respectively to be further compiled. Compared to the uni-processor, those are new tasks on the programmers' shoulders now which are *non-trivial*. Unlike the pivotal role of traditional compilers in the uni-processor flow, little *compiler support* exists for MPSoC programming. This practice is currently labor-intensive, error-prone and costly.

1.3 Our Position

In this paper, the practical design of a compilation framework for heterogeneous MPSoCs targeting the domain of streaming applications, named MAPS, is presented. Streaming programming models have gained acceptance recently in embedded software design, as they closely resemble computation of signal processing, wireless and multimedia applications.

We argue for the need and significance of positioning the language and tool design from the perspective of *practicality* to address the MPSoC programming problem in this work. The design goal is being able to apply the proposed compilation framework in practice for real-life, complex MPSoC platforms. The MAPS compilation framework achieves this through two parts: (a) *a C language extension design* that models concurrent processes of streaming applications following the KPN (Kahn Process Networks) model in a clean, practical and efficient manner; and (b) *a retargetable source-to-source compiler design* that serves as an extensible infrastructure to automate the MPSoC compilation process.

The MAPS compiler framework and the C language extension together provide a combination of key capabilities that are critical for practical MPSoC programming:

1. *Portability and retargetability*: The C language extension design minimizes the cost for porting programs among different target architectures. The compiler framework supports retargetability for real-life, complex heterogeneous MPSoCs where processors have proprietary APIs and target specific primitives.
2. *Low entrance barrier*: The amount of effort which is needed to learn and migrate legacy C source code for MPSoCs is kept low.
3. *Clean and concise representation*: Specification of concurrency in source code is clean and concise.
4. *Flexible, extensible compiler*: As mentioned earlier, the complexity of compilation frameworks for MPSoCs requires a more flexible and extensible software architecture design compared to uni-processor compilers. A flexible, extensible compiler allows e.g. integration of different optimization modules and customization of the compiler framework for various use-case scenarios.

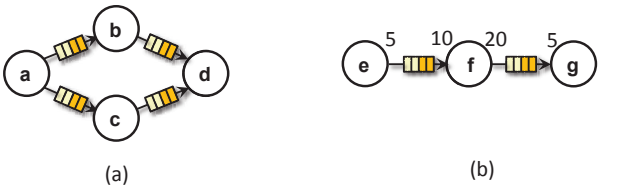
The rest of this paper is organized as follows: Section 2 describes the C language extension, CPN (C for Process Networks). Section 3 presents the design of MAPS compiler framework. Some limitations which exist in the current status of this work are discussed in Section 4. In Section 5, our experimental results are presented including applications and experiences of using MAPS as a compiler infrastructure. Section 6 gives a survey of related works and compares MAPS to them. Section 7 concludes the paper with a summary and outlook into future work.

2. LANGUAGE DESIGN

As this work focuses on streaming applications, a short description of the streaming program model is given firstly in this section. There are several different approaches to realize those models in programming practice: we chose using the language extension over other choices. The rationale behind our design decision is discussed, followed by an overview of CPN.

2.1 Streaming Models

Streaming (or dataflow) models have gained acceptance recently in embedded software design. KPN [11] is a prominent example widely used in practice to model signal processing applications. A KPN is represented as a directed graph which is built of nodes representing autonomous processes of computation. The edges between the nodes represent unbounded unidirectional FIFO (first-in first-out) channels that transmit data items. Reading from an empty FIFO channel results in the process being blocked until data to read becomes available. A four-process KPN example is shown in Fig. 2(a). The processes in a KPN can access channels anywhere in their arbitrary control flow so that it is flexible enough to model data dependent behavior. A true subset of KPN, SDF (Synchronous Dataflow) [18], requires that the control flow of an SDF process is an endless loop



(a) (b)

Figure 2: Process Network Examples: (a) A four-process KPN example (b) A three-process SDF example: the numbers indicate how many tokens are consumed or produced per iteration.

with pre-defined fixed channel accesses. Fig. 2(b) is an example of SDF with three processes. For instance, process f always consumes 10 tokens from process e, and produces 20 tokens for process g per iteration. SDF is very common in specifying signal processing algorithms in that many algorithms such as sampling and filter operations follow this model strictly. As KPN and SDF resemble the way humans think of parallel processing, the models are perceived as *intuitive* ways to specify streaming programs.

2.2 Design Rationale

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 to specify semantics of process networks at a high level e.g. containing enough structural information about the channel accesses. This firstly 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). Secondly, programs are portable and semantic analysis opens abundant opportunities for code transformations and optimizations (e.g. process fusion and fission). Although C is not an ideal vehicle to carry concurrency specification, this design compromise is made considering the large C legacy code base and popularity of C in the embedded industry.

2.3 CPN Overview

A brief overview of CPN is given in this section. CPN is an extension of the standard C language to support the description of streaming models. It is designed to keep the syntax as close to C as possible while making PNs structured and readable. We chose to introduce a few new keywords (prefixed with `__PN`) for channels, processes and channel accesses of streaming models.

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 to allow for code reuse. A process template describes the functionality of a process and the

```
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

```
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)

channels this process needs to access (either read or write). Processes are always created as *instances* of process templates. Readability and conciseness of the code is improved, e.g. 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 e.g. in fax machines. Data are encoded into a stream of duos, i.e. the number of appearances and the data element itself. For instance, the original data `AAAABBCCCCCDDD` is compressed into `4A2B5C3D`. RLD is the inverse of RLE. A KPN process template (`__PNkpn`) with identifier `RLD` is shown in Listing 2. It reads integers from its input channel `EncIn` and outputs integers to 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.

Access to a channel is always realized via `__PNin` or `__PNout` in the body for KPN-like processes. 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, firstly, a token from the input channel `EncIn` is read and assigned to a local variable `count`. Then, 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.

As 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

```
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)

```
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

template (__PNsdf) for an adder functionality, 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.

Language Features

As CPN is designed to model streaming programs, a number of specific features are built into the language extension for streaming computing to make the code further clean and concise. For instance, multiple-item access on channels and multicasting (i.e. multiple reader support on one channel) are supported. Here one feature is introduced in detail: *sliding window access*.

Sliding windows are often seen in stream programs [29]. It refers to the fact that some operations which work on multiple channel items operate in an overlapping manner on the data. A common example is the FIR (Finite Impulse Response) filter. Several channel items are accessed at the same time and most of those channel items might also be accessed on successive iterations.

We implemented a special keyword to model sliding the access window on a channel. Sliding the access window on a channel to read from means that a number of elements at the beginning of the channel array variable are marked as being

```
1  __PNkpn SobelHor __PNin(int in[WIDTH])
       __PNout(int out[WIDTH])
2  {
3    int y, x;
4    ...
5    __PNin(in:3) {
6      for (y = 1; y < HEIGHT - 1; ++y) {
7        __PNout(out) {
8          out[0] = 0;
9          for (x = 1; x < WIDTH - 1; ++x)
10           out[x] = (  -1 * in[0][x-1] + -2 *
                 in[0][x] + -1 * in[0][x+1]
11                     +  1 * in[2][x-1] +  2 *
                         in[2][x] +  1 *
                         in[2][x+1]
12                   ) / 8;
13         out[WIDTH - 1] = 0;
14       }
15       if (y < HEIGHT - 2)
16         __PNmove(in:1);
17     }
18   }
19   ...
20 }
```

Listing 5: CPN Example Code: Sliding window

consumed, the remaining items are moved to the front of the channel array variable and the rest of the channel array variable is filled with new items from the channel. If not enough new channel items are available, the operation will block. Similar operations can apply to a channel for writing with sliding window access.

Sliding the access window is done with the following command __PNmove(channel:number) where channel denotes the channel identifier and number the (constant) number of items the access window is to be slid. Listing 5 shows a code excerpt of the two-dimensional Sobel edge detection application. The algorithm performs filtering horizontally and vertically to compute an approximation of the gradient. Listing 5 is the horizontal filter part. At line 5 using __PNin(in:3), it asks to take 3 lines of the input image in for computing the gradient. Assume here that lines 7, 8 and 9 of the input image are read in. At line 16 using __PNmove(in:1), the access window is moved by 1, meaning that the lines 8, 9, and 10 of the image become the next 3 lines for calculation.

3. COMPILER DESIGN

Once applications are written in CPN, a compiler framework is needed to take them as input, compile and generate correct and optimized binary code for target MPSoC platforms. Ideally, it would act in the same role as the black-box compiler for uni-processors (see Figure 1(a)). The complexity of MPSoCs nowadays requires a more flexible design in reality, though. Our compiler design rationale is firstly discussed in this section, followed by the details of our implementation.

3.1 Design Rationale

Many previous approaches for multiprocessor systems follow the principle of compiler design for uni-processors: the compiler works on input programs, builds an internal intermediate representation, performs optimizations and generates target binary code. This black-box *monolithic* design philosophy is that (a) the compiler performs generation of correct and optimized target code without (much) help from programmers; (b) the compiler has as much information about architectural details as possible in order to per-

form target-specific code generation and optimization. The essence of this design philosophy is to incorporate the complexity into a single tool (compiler) that is developed by a small group of highly skilled experts. Therefore, programmers have a much improved productivity in designing software assuming 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 3) and our observations are:

1. The complexity of hardware architectures grows in such rapid manner that a monolithic compiler is unable to keep pace. For most of uni-processors like RISCs, compilers still managed 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, like (clustered-) VLIW DSPs, ASIPs (Application-Specific Instruction set Processors) and eventually multicores which have multiple of those scalar processors on a single chip. The level of user intervention required in developing and using compilers increases rapidly (shown in Figure 3), which is another proof that the monolithic approach is reaching its limit of feasibility.
2. Compilers usually take a significant time to mature with an enormous amount of investment. As explained in Section 1, heterogeneous MPSoC platforms will utilize more and more individual IPs from different vendors. Therefore, a monolithic approach for multicores that generate target specific binary code (for different processors) directly is not economical, as it does not leverage the existing C compilers for uni-processors.
3. The usage scenarios of compilers are much more diverse now, especially in the embedded domain. Multiple optimization goals are commonly seen such as performance, energy efficiency and code size.

Moving along the trajectory shown in Figure 3, we argue that for heterogeneous multicore systems, a new flexible, extensible compiler design is desired and required, instead of continuing the monolithic approach. The user interventions to account for the new architecture trends will further grow, including a user-defined mapping specification and objective functions for code generation and optimization.

3.2 Our Implementation

The software architecture of the compiler framework developed to compile CPN programs to heterogeneous MPSoCs is introduced in this section. 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 firstly elaborated followed by a description on 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 [16], the C frontend of LLVM compiler infrastructure. Figure 4 (a) shows a high-level, internal structure of *cpn-cc* which consists 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:

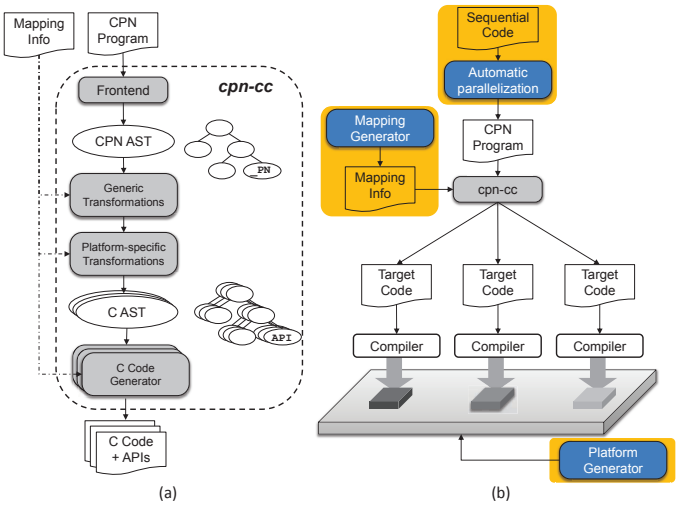


Figure 4: (a) Structure of the source-to-source compiler *cpn-cc* (b) An example of a complete compiler framework for a three-processor heterogeneous MPSoC: the parts with a shaded background are optional to the core compiler framework.

- **Frontend**: The frontend starts with C preprocessors to provide functionality like 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 like process templates and channel accesses. Then, the CPN-aware semantic analysis builds the AST (Abstract Syntax Tree) 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 the job of source-to-source translation. They are categorized into *Generic* and *Platform-specific* transformations, very similar to the sequence of code optimizations in a classic uni-processor compiler [1]. Generic transformations are platform independent. One example is the transformation from SDF templates to KPNs, where 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 printer of Clang generates the C source code from the ASTs after transformations. In some cases, other additional auxiliary files like configurations and makefiles also need to be generated in order to be further compiled with the individual processor native C compilers.

Compared to other source-to-source translation tools which are not based on a formally built AST (e.g. textual replace-

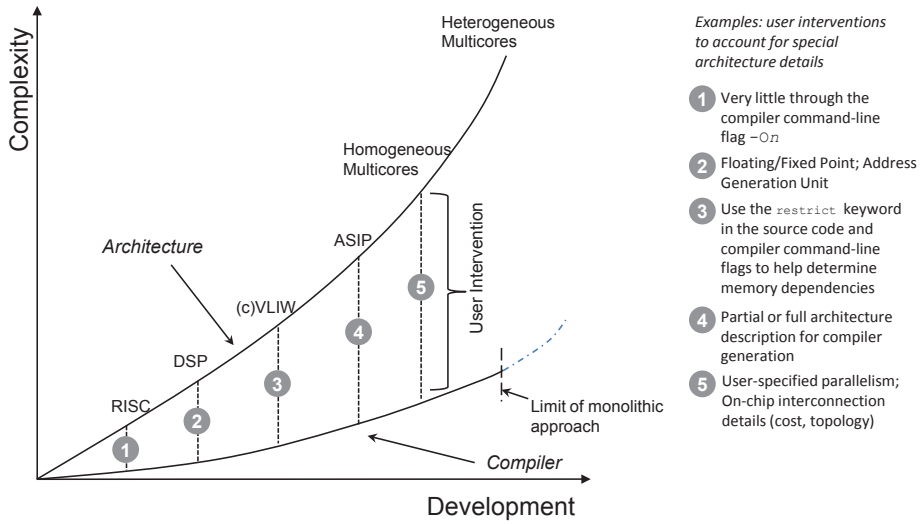


Figure 3: Evolution of compiler development vs. architecture development

ments), 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. The readability of the generated source code is also kept 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, though. The effort is well justified in that cpn-cc is designed to be retargetable and a majority of 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 4 (b):

- In addition to CPN programs that are considered as 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. 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 thus can 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 eases a lot the retargeting process of the framework. Target specific transformations can be developed in order to better exploit the specific hardware details.
- The compilation framework is able to collaborate with other state-of-the-art tools thanks to the *extensible* design. For example, an intelligent mapping info generator, referred as Mapping Generator in Figure 4 (b), can be easily integrated with the core compiler framework. In some scenarios of embedded system design, software development needs to proceed simultaneously with the architecture development e.g. using a Platform Generator (see Figure 4). Our work also enables retargeting the compiler framework to work with ESL (Electronic System Level) design tools for early system level design. Another possible usage scenario also shown in Figure 4 is to include an external tool which converts a sequential C code automatically into CPN code to further improve the productivity of programmers.

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

4. CURRENT LIMITATIONS

There are limitations in the current status of our work. The main target platforms which MAPS addresses are those consisting of programmable processors. Those processors are C programmable by vendor C compilers. There is an increasing usage of hardware accelerators e.g. using ASIC, FPGA or weakly programmable processors in embedded MPSoC platforms. Our current framework cannot directly work for those yet. Additionally, our current implementation has much room for advanced automatic parallelization techniques for further improvements. Currently, the retargeting process of MAPS is not fully automated. We plan to address those limitations in our future work.

5. RESULTS

The previous two sections describe our design and implementation of the MAPS compiler infrastructure. In this section, we evaluate this design in: (a) illustrative uses of MAPS for real-word MPSoC platforms and other scenarios; (b) how retargetability of MAPS for different MPSoC platforms is achieved.

5.1 Applications and Experiences

MAPS, as a MPSoC compiler infrastructure, has been used for a number of MPSoC platforms to demonstrate its *feasibility* and value in automating the 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 [25]) 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.

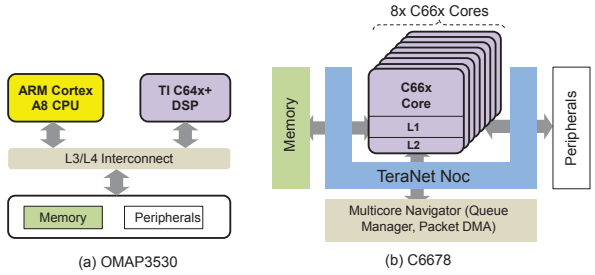


Figure 5: MPSoC platform block diagrams

- **TI OMAP3530** [26] is a heterogeneous MPSoC platform which features an ARM Cortex A8 processor and a TI C64x+ DSP (Digital Signal Processor), shown in Fig. 5(a). Software-wise, Linux is used as OS on the ARM, while a lightweight multitasking operating system (TI DSP/BIOS) is used on the DSP. TI also provides a DSP/Link layer which handles the inter-processor communications. OMAP is a typical MPSoC platform where compiler tool chains for processors are separate and different. The gcc compiler needs to be used for the ARM processor while the DSP requires a proprietary C compiler from TI.
- **TI C6678** [27] is based on TI's KeyStone multicore architecture which integrates eight C66x DSPs, shown in Fig. 5(b). 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 tool-chain (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) [14]. 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 largely available on multicore platforms (e.g. x86 or ARM based). MAPS has been retargeted to generate stream programs using the Pthreads API to realize concurrent processes with FIFO communications. This expands the platforms that MAPS supports to virtually all platforms which 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. Though there exist some differences in different MAPS instances for these platforms, the basic principle and flow applies to all. As an example, the case of using MAPS for the TI OMAP3530 is introduced below, shown in Fig. 6. 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. 6, a simplified code excerpt of the transformation results for FIFO channel accesses is shown. The generated code is both editable and readable, thus enabling opportunities for performance fine-tuning by programmers. Note, it is critical that all transformations take place at the AST level which provides complete semantic contents of the source code. For instance, the channel *DecOut* is a FIFO communication which occurs between different processors under this mapping, while the channel *EncIn* communication is local. This information needs to 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 source-to-source compilation, target specific C files are generated for ARM and DSP and the existing vendor tool-chains are re-used. The compilation flow is thus fully automated.

Though this paper is not intended to study parallelization of specific benchmarks using MAPS, some performance results are presented here from using MAPS on TI multi-DSP platform C6678 for completeness (see [23] for retargeting MAPS to C6678 in detail). Two signal processing benchmarks were selected: a digital audio filter and an airborne radar application. The digital audio application implements fast convolution filtering using FFT (Fast Fourier Transform) 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. The second radar application is to detect moving objects on the ground from air by sending periodic radar pulses. Fig. 7 shows graphical representations of both benchmarks. Both benchmarks existed initially as sequential C code. The conversion from sequential C code to parallel CPN versions took us about half day to complete per benchmark, thanks to CPN being close to C.

In order to evaluate both benchmarks on the C6678 platform, we used a mapping heuristic, GBM (Group-Based

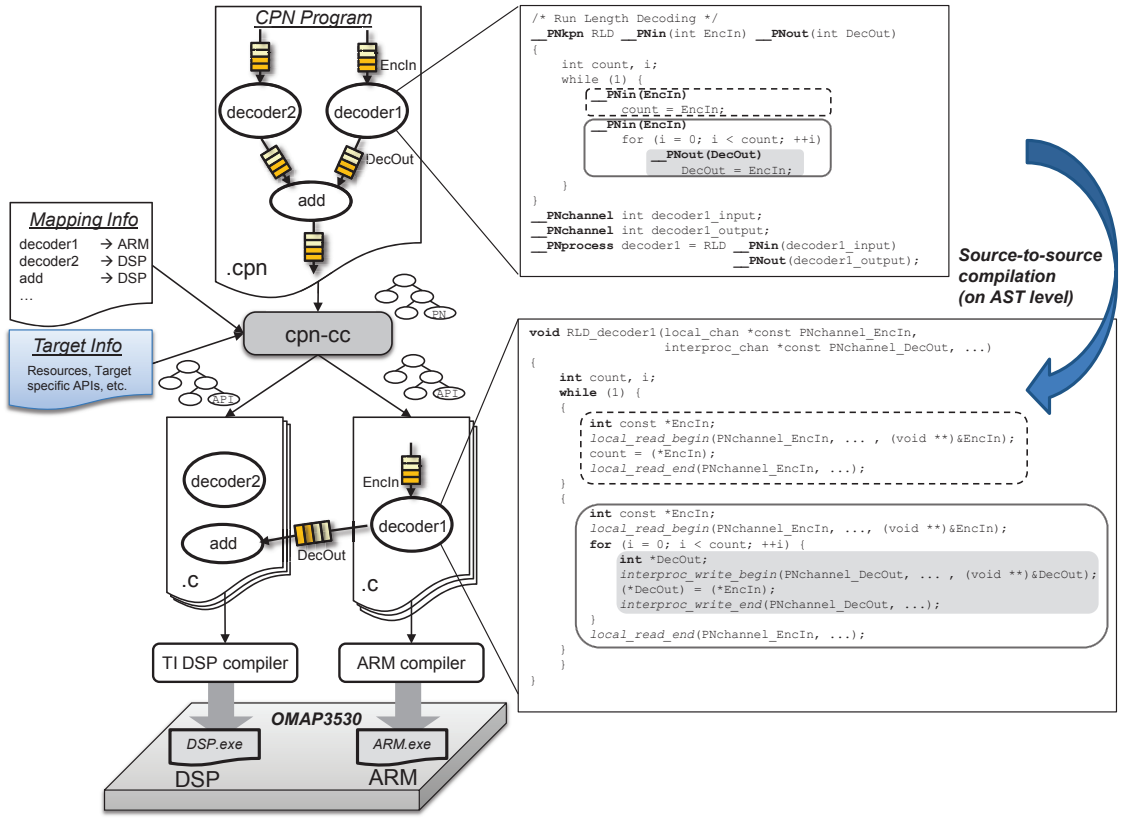


Figure 6: A complete compilation framework for OMAP3530 using MAPS

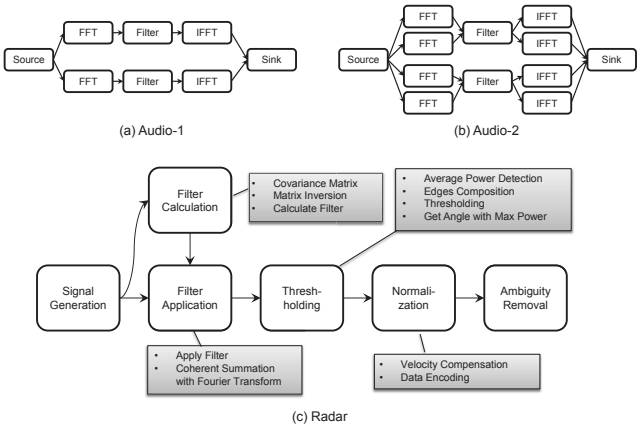


Figure 7: Benchmarks

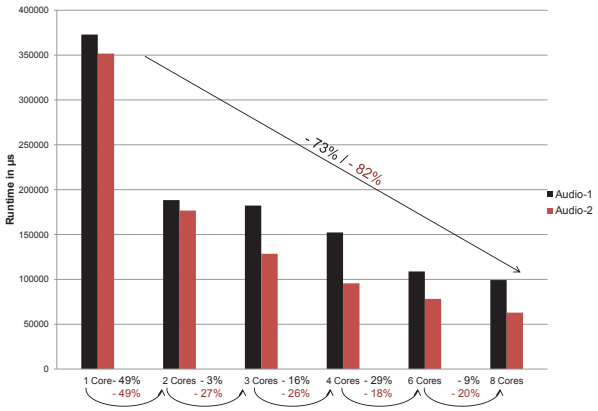


Figure 8: Results of the audio filter

Mapping) proposed in [5], to generate spatial mapping for both benchmarks. GBM takes constraints e.g. available number of processors as inputs to compute mapping. Fig. 8 and Fig. 9 show the performance results when the constraint on available number of processors increases from 1 to 8 for the audio filter and from 1 to 4 for the radar application respectively. Each data point corresponds to the runtime result of the generated mapping which is measured on the hardware board. It can be seen that achieved speed-ups increase when the number of available processors rises for both benchmarks. 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 productivity of software developers is also increased greatly as the compilation process is automated. In addition, programs written in CPN are also portable to other MPSoC platforms.

5.2 Retargetability

One of the main *parameters* of different MAPS instances is so-called *target info* shown on the left of Fig. 6. It contains 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 info. Our experience on the retargetability of the MAPS infrastructure is reported in this section.

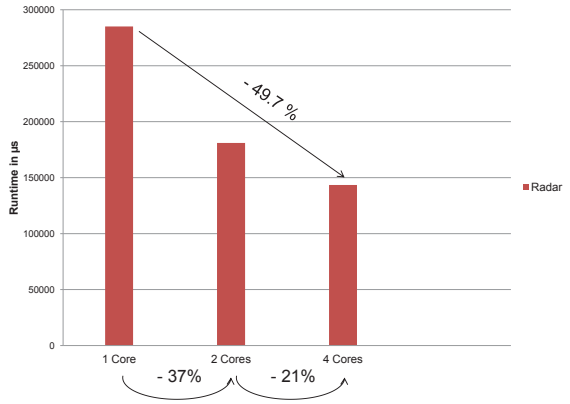


Figure 9: Results of the radar application

Table 1 summarizes the retargeting process that was done 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 gives the knowledge where concurrent processes are possible to execute simultaneously to the compiler.
- **Multi-tasking Runtime:** this indicates how the runtime environments of processors of the target MPSoC support concurrent processes. Often in case 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 execute simultaneously while communicating with each other. Therefore, it is necessary for the compiler to know possible ways to perform the communication between processes. This also includes the type (inter- or intra- processor) and specific API calls.

The actual retargeting process consists of utilizing this target specific information in various places of the MAPS infrastructure. The effort that we spent to support these platforms is reported in Table 1. It ranges from 20 man-days to 5 man-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 automation level of compilation process.

6. RELATED WORKS

There has been a wide range of works on streaming programming and compilation frameworks (see [12] for a recent survey). Those are categorized into four classes below.

Approaches based on new languages

New languages, instead of using or extending existing languages, have been proposed to specify concurrency in streaming programs. StreamIt [30] from MIT is an architecture independent language which is based on SDF with some extensions. The basic unit of computation is the *filter*, which implements an independent actor to translate data from input channels to output channels. In addition, hierarchical primitives such as pipeline, split-join and feedback loop are provided in StreamIt to compose filters into graphs. Other similar works include Tiny-SHIM [8], CAL [9] and ACOTES Project [22].

In general, new language approaches have salient advantages that a programming model can be built as close to streaming computing characteristics as possible and code optimization can benefit from many structural hints (e.g. hierarchical primitives in StreamIt). A drawback is that new languages usually require a long period of time to be accepted in industry.

Approaches using APIs

Unlike developing new languages, many projects use APIs to specify streaming applications. YAPI [7] and TTL [31] are abstract, task-level interface APIs for design and programming of embedded MPSoCs. While YAPI is used to model applications as process networks, TTL can be used both for parallel software specification and as a platform interface for integrating hardware and software tasks. In the DOL framework [28], processes of streaming applications are described in C with APIs to realize e.g. data communication. The structure of whole programs is specified in an XML file.

API approaches are in general attractive for C programmers due to little learning curve required. Another implication using APIs is that a possible huge investment on developing a compiler can be avoided. Nevertheless, using APIs usually makes code cluttered thus lowering the readability. It is also difficult to extract high-level information e.g. the structure of streaming programs from low-level APIs so that opportunities for high-level optimizations (e.g. process fusion) are reduced.

Approaches using language extensions

Between the approaches using new languages and those using APIs, there have been also many research proposals on various types of language extensions (mostly to C) to describe streaming programs. Brook [4] is a streaming programming environment for GPU-based computing. The language extends C to include data-parallel constructs called *streams* and defines functions by the keyword *kernel* to operate on streams. Other similar approaches include StreamC/KernelC [13] and Cg [19].

The commonality in those works is that the language design is specifically towards certain architectures like GPUs, media processors or network processors. The focus is mostly on enriching data parallelism representation and extending manipulation for streams in the languages.

Automatic parallelizing compilation techniques

Compilation techniques of streaming programs for MPSoCs have been extensively studied recently (see [21] for an overview). Those include automatic parallelization of sequential code (mostly loops) like Compaan in [15] (used in HEAP Project [17]), a parallelization technique of nested loop programs in [10], DSWP in [24] and pn in [32].

The Cetus tool [6] is a compiler infrastructure which targets C programs and supports source-to-source transformations. The original goal of Cetus was for research on multicore compiler optimizations that emphasizes automatic parallelization. Lately, it has been reported in [2] that Cetus was used to build translators for programs written in the OpenMP directive language for multicore compilation, e.g. an OpenMP to MPI translator for many-cores and a translator for OpenMP onto GPU architectures. Compared to our work, Cetus focuses more on automatic loop parallelization and supporting OpenMP programs for multicore machines.

7. SUMMARY

This paper describes our approach to tackle the embedded MPSoC programming challenge *from a practical perspective*. In our opinion, it is of paramount importance that the language and compiler design for MPSoCs has roots in practicality. MAPS, a compiler infrastructure for heteroge-

	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: Retargeting MAPS towards MPSoC platforms

neous MPSoCs, has been presented. CPN, a language extension to C to model streaming applications, is designed for portability and conciseness. The compiler *cpn-cc* which does source-to-source translation on the AST is able to perform code transformations at high-level and leverage the existing C compilers of processors on the target MPSoC platforms. We evaluated the design of MAPS by applying it to off-the-shelf MPSoCs from TI (both homogeneous and heterogeneous) and other platforms.

Several improvements are planned beyond this work, including integrating platform-specific heuristics for code optimization and enhancing CPN language by allowing combining several processes into a super process template (called process nesting or container). More case studies will also be carried out to evaluate the quality of generated code further.

8. ACKNOWLEDGMENTS

This work has been supported by the UMIC Research Centre, RWTH Aachen University (`www.umic.rwth-aachen.de`).

9. REFERENCES

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

[2] H. Bae, L. Bachega, C. Dave, S.-I. Lee, S. Lee, S.-J. Min, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. In *CPC*, 2009.

[3] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *DATE*, pages 983–987, 2012.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3), Aug. 2004.

[5] J. Castrillon, A. Tretter, R. Leupers, and G. Ascheid. Communication-Aware Mapping of KPN Applications onto Heterogeneous MPSoCs. In *DAC*, 2012.

[6] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *IEEE Computer*, 42(12):36–42, Dec. 2009.

[7] E. A. de Kock and et al. YAPI: application modeling for signal processing systems. In *DAC*, pages 402–405, New York, NY, USA, 2000. ACM.

[8] S. A. Edwards and O. Tardieu. SHIM: a deterministic model for heterogeneous embedded systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(8), Aug. 2006.

[9] J. Eker and J. Janneck. CAL Language Report. Technical report, University of California at Berkeley, December 2003. ERL Technical Memo UCB/ERL M03/48.

[10] S. Geuns, M. Bekooij, T. Bijlsma, and H. Corporaal. Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems. In *DATE*, March 2011.

[11] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress 74*, pages 471–475, North Holland, Amsterdam, 1974.

[12] W. Haid, K. Huang, I. Bacivarov, and L. Thiele. Multiprocessor SoC software design flows. *IEEE Signal Processing Magazine*, 26(6):64 –71, November 2009.

[13] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. Programmable stream processors. *IEEE Computer*, 36:282–288, 2003.

[14] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms. In *DATE*, 2005.

[15] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *International workshop on Hardware/software codesign*, 2000.

[16] C. Lattner. LLVM and Clang: Next generation compiler technology. The BSD Conference, Ottawa, Canada, May 2008.

[17] L. Lavagno, M. Lazarescu, I. Papaefstathiou, A. Brokalakis, J. Walters, B. Kienhuis, and F. Schaefer. HEAP: A Highly Efficient Adaptive Multi-processor Framework. In *DSD*, pages 509 –516, sept. 2012.

[18] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proceeding of the IEEE*, volume 75, 1987.

[19] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3), July 2003.

[20] G. Martin. Overview of the MPSoC design challenge. In *DAC*, pages 274 –279, 2006.

[21] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke. Multicore compilation strategies and challenges. *IEEE Signal Processing Magazine*, 26(6):55 –63, November 2009.

[22] H. Munk, E. Ayguadé, C. Bastoul, P. Carpenter, Z. Chamski, A. Cohen, and et al. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. *International Journal of Parallel Programming*, 38, Apr. 2010.

[23] M. Odendahl, W. Sheng, M. Aguilar, R. Leupers, and G. Ascheid. Automated Code Generation of Streaming Applications for C6000 Multicore DSPs. In *5th European DSP Education and Research Conference*, Sep. 2012.

[24] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38*, 2005.

[25] F. Schirrmeister and P. Sheridan. Optimizing Multicore System Performance. *Synopsys Insight, Issue 1*, 2011.

[26] Texas Instruments. OMAP35x Product Bulletin. [Online] Available `http://www.ti.com/lit/sprt457` (accessed 11/2010), 2009.

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

[28] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *ACSD*, 2007.

[29] W. Thies and S. Amarasinghe. An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design. In *PACT*, Vienna, Austria, Sep 2010.

[30] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr 2002.

[31] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *CODES+ISSS '04*, 2004.

[32] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1), Jan. 2007.