COMPOSE: A Composite Embedded Software Synthesis Approach

Projjol Ray, Rainer Laupers, Gerd Ascheid
Institute for Communication Technologies and Embedded Systems (ICE)
RWTH Aachen University, Germany
{ray, leupers, ascheid}@ice.rwth-aachen.de

***Abstract*—Modern embedded software synthesis faces two distinct challenges. One one hand embedded software complexity continues to grow steadily, and at the same time they are often targeted to heterogeneous-MPSoC platforms, which are extremely difficult to program. Distinct families of tools have evolved, which specialize in tackling the specific issues associated to these two aspects of embedded software synthesis. However, in today's competitive environment a complete design flow is needed, which can seamlessly transition a complex high level design to a multicore centric implementation. Integrated approaches suffer from added complexity and maintenance issues, for having to solve these two related but somewhat different problems simultaneously. This paper introduces a composite design technique - COMPOSE - that connects the existing high level modelling environments with multicore oriented software synthesis frameworks to handle this two-way challenge. The resulting design flow proves to be a superior alternative than both the existing integrated tools as well as the decoupled approaches, since it combines the respective strengths of both of these tool families without being unduly complex. Our experiments show that use of such a composite approach opens up new possibilities for high level performance optimization and can shorten the application development life cycle by upto 20%.**

I. Introduction

With the increasing demand for features from mobile embedded platforms, complexity of embedded software has been on the rise [1]. To meet the conflicting goals of high performance and energy efficiency, these mobile platforms have evolved towards heterogeneous multicore architectures [2]. Together, these two factors have substantially complicated the process of embedded software development for such platforms.

High level design helps in tackling challenges like high software complexity, early verification, reduction of coding effort and coping with ever shortening Software Development Life Cycles (SDLC). High level design environments (HLDEs) like Simulink[3], Synopsys SPW[4], and COSSAP (later Cocentric System Studio)[5] aim to ease this complex design process by promoting modular development and massive code reuse. These tools provide graphical interfaces for *block diagram* based application modelling and come with large set of well tested libraries of readily implemented software *components*, which significantly reduce the development effort. HLDEs provide simulation support for highly abstract application models enabling early target agnostic analysis and verification. By explicitly modelling applications as coarse grain components and their interaction, they often expose latent parallelism and ease up the debugging process. By supporting easy (graphical) modification of an application at the algorithmic level, these tools provide unparalleled scope for design space exploration.

Modern heterogeneous-Multiprocessor Systems-on-Chip (he-MPSoC) target platforms present a trade-off between significant computation power and energy efficiency, but programming them efficiently remains a formidable task. Efficient implementation of embedded applications on such platforms involves issues like identification of a suitable Model of Computation (MoC) together with a programming model for implementing it, partitioning the application effectively into tasks, optimal mapping and scheduling of these individual tasks to the target architecture and dealing with multiple, often disparate native compiler tool-chains [8]. The extreme complexity of programming such platforms has kept software productivity significantly lagging behind the hardware productivity [7] and has been a subject of intense research [9]. This has led to the recent emergence of a specialized family of Multicore Programming Frameworks (MPFs). Examples of which include MAPS[12], CIC[13], DOL framework[14], Daedalus[15] etc. A detailed survey of such frameworks can be found in [8] and [9].

In future, embedded software will continue to grow in complexity with the emergence of complex standards like LTE [10] (in baseband processing) and WebP [11] (in multimedia domain) and will be targeted to increasingly complex he-MPSoC architectures. This makes high level design and efficient implementation on MPSoCs two important aspects of modern embedded software synthesis. Speedy software design for multicore targets requires bringing together the strengths of both the abovementioned tool families. But the emergence of these two disconnected classes of tools and their respective user communities with specific skill sets has led to a *gap* in the software synthesis process for multicore platforms.

One approach to bridge this gap would be to employ an integrated tool, which allows users to model at a high level as well as to generate efficient multicore oriented code. But such a tool would be very complex to develop and maintain, while keeping pace with the advancements at either end. This work proposes to bridge the aforementioned gap by connecting the HLDEs to the MPFs by automating the transition of high level application models to MoC based representation. This results in an efficient design flow, allowing the user to couple *existing tools* from these two families, thereby enabling synthesis of complex software for heterogeneous embedded platforms. This technique has been named **COMPO**site **S**ynthesis of **E**mbedded-software (**COMPOSE**) and has been illustrated here by coupling a commercial HLDE (Synopsys SPW) with a state-of-the-art MPF (MPSoC Application Programming Studio (MAPS)). The main contributions of this work are:

- A novel composite embedded software synthesis approach for multicore platforms.
- An extendable framework for automating the transition between high level design environments (HLDEs) and multicore programming frameworks (MPFs).

978-1-4673-8511-4/15 $31.00 ©2015 IEEE

- A demonstration of how COMPOSE design flow can be used to efficiently adapt applications for multiple MPSoC platforms.

COMPOSE allows utilization of platform specific performance hints generated by MPFs as feedback, enabling optimization at the algorithmic level in HLDEs. Since this approach only loosely couples the HLDEs to MPFs, it offers the great benefit of leaving an independent path of evolution for both the families of tools without breaking the design flow. Additionally, not tying a high level model tightly to specifics of multicore implementation leads to a very efficient co-exploration of design space and multicore platforms.

II. Related Work & Motivation

Several attempts have been made to create an integrated framework to support a *complete* design flow, which can synthesize code for a target platform from a high level specification. One of the earliest works was the Gabriel design framework [16], which allowed graphically modelling applications as hierarchical block diagrams. It targeted high performance multi-DSP architectures, but was not intended for rapidly evolving heterogeneous multicore targets. The Ptolemy framework [6] is capable of modelling systems that involve diverse MoCs and synthesizing software from dataflow graphs.

The DESCARTES [17] software synthesis framework proposed to integrate software synthesis for target platforms into the high level simulation platform COSSAP [5]. By doing so, it underwent a seamless transition from the simulation to the implementation levels of abstraction. However this framework primarily targeted unicore Digital Signal Processors (DSPs). Thoen et al. [24] focused on the issue of high level software synthesis for real-time platforms but did not deal with the issues of programming heterogeneous MPSoCs.

Bulk of the work done in this field aimed at creating *monolithic solutions* for efficient embedded software synthesis, primarily by creating an integrated environment that tightly couples modelling with implementation for targets. These integrated tools lack the extensive abilities of the specialized HLDEs and MPFs for dealing with high level modelling and multicore software synthesis respectively. The innovative composite approach proposed in this paper benefits from combining the strengths of both these families of *elite tools* in their respective areas of embedded software synthesis. Yet this approach bypasses the excessive complexity of developing and particularly maintaining an integrated tool. Thus this approach provides a smarter alternative to both the integrated environments as well as the disconnected approaches, which require a non-trivial manual effort for transitioning from a high level model to a representation tailored towards MPSoC oriented software synthesis.

III. The COMPOSE Design Methodology

The composite design methodology proposed in this work basically consists of modelling and verifying an application at a high level using a HLDE and then automatically transforming it to a representation usable by MPFs. The HLDEs comprise the *frontend* for the COMPOSE design flow, while the MPFs form its *backend*. In this paper, this design flow has been demonstrated by a concrete example, which uses SPW as the representative HLDE and MAPS as the MPF. In the rest of this paper, the COMPOSE design flow will be illustrated in terms of this example.

Figure 1 shows the schematic representation of the COMPOSE design flow - highlighting a specific instance of it, where SPW and MAPS are used as frontend and backend respectively. In this instance of the COMPOSE methodology, the process of developing an embedded application starts with modelling at high level and simulating it in SPW. Once a stable design is obtained, it is processed by the **M**odel **to I**mplementation (**M2I**) transformation framework to automatically generate its MoC based representation. Using the platform exploration features of the MAPS framework, the application designer can then obtain *target architecture specific* feedback about the behaviour of his design. This feedback can be used to optimize a design at the algorithmic level and to fine tune it further for a specific target platform; subsequently the design can be subjected to multiple iterations of performance tuning, as shown by loop in Figure 1. Thus, while operating at a highly abstract level, an application can be adapted for different MPSoCs using the COMPOSE design methodology.

Designers of DSP applications tend to think of their systems in terms of block diagrams. Particularly for complex DSP applications, a hierarchy of block diagrams is employed to manage complexity. HLDEs allow modelling of applications as hierarchical block diagrams, thus supporting this design philosophy which lets a system level expert to quickly capture the software requirements of a DSP application. Most HLDEs like SPW and Simulink come with *extensive* libraries of commonly used DSP functions and implementations of several well known communication standards. These collection of well tested software components can be used as *readily available* building blocks to stitch up new applications. This can dramatically reduce the development effort. In addition, HLDEs provide extensive (architecture agnostic) simulation and debugging support which can be used for stabilizing and verifying the application.

MPF backends like MAPS, DOL, CIC etc. specify applications in terms of MoCs like KPN [19] or SDF [18]. Such an application specification is inherently well suited for multicore targets since it can express the application as a independent set of tasks or processes that can be run concurrently and communicate through FIFO channels. Using the architectural

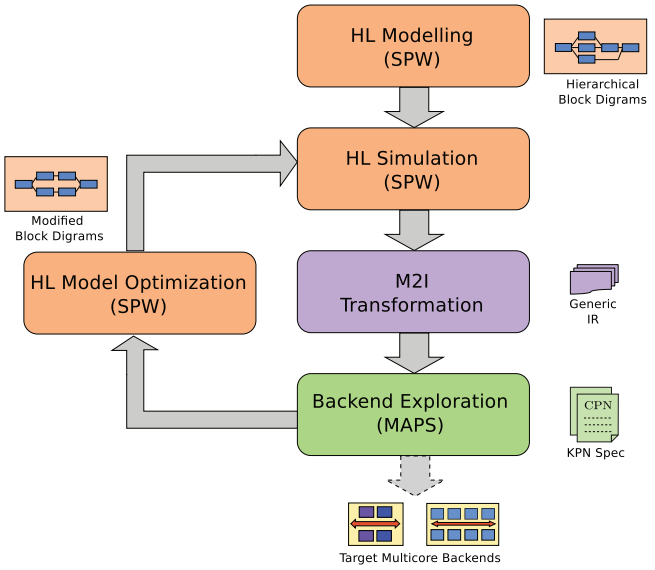

Fig. 1. The COMPOSE design methodology.

details of the target platform, most MPFs can estimate the performance characteristics at task level. This is one of the most crucial platform exploration features of backends, which can be exploited to determine how each task will perform on each type of processing element of the target platform [21]. These performance results can be utilized for optimal mapping and scheduling of the application for specific targets. MPFs also take care of some of the important multicore software synthesis issues like schedule computation, resource allocation, computing the minimum FIFO depths and implementing inter-task communication. In particular, the MAPS MPF provides the flexibility of choosing between manually mapping the tasks to desired processing elements or generating the mapping automatically, while obeying certain system constraints. Furthermore, being a source-to-source transformation framework, it enables retargetable code generation, making it possible to use native compiler toolchains for the target processing elements and linking with optimized target specific libraries.

The M2I framework *enables* this composite design flow by automating the transition from high level model to a MoC based specification suitable for multicore programming. This is achieved by transforming the input model into an intermediate representation (IR) generic enough to capture high level application models from various HLDEs (see section IV-B). This IR is then used to generate the specific multicore-centric representation depending on the chosen backend MPF. The details of M2I framework are presented in section IV-C, where extendibility of this framework is discussed for accommodating different frontends and backends.

IV. Model to Implementation (M2I) Transformation

A. The M2I Overview

As explained in section III, the M2I source-to-source transformation framework is the *key component* underpinning the COMPOSE design flow, which automates the conversion of a high level application model to a MoC based specification. For the M2I transformation to be possible, the programming model employed by the frontends and the backends need to be semantically consistent. For a particular instance of M2I transformation to be valid, the MoC employed by the frontend needs to be equally or more restrictive than the one used by the backend. If the frontend tool employs a semantically incompatible programming model, which can not be interpreted as a dataflow model, or is less restrictive than the MoC supported by the backend (typically KPN) - then M2I transformation becomes infeasible.

Most HLDEs employ dataflow programming models, for instance both Simulink and SPW programming models are based on Homogeneous Synchronous Dataflow (HSDF) [18]. On the other hand MPFs like MAPS, DOL and CIC allow KPN based application modelling. HSDF being more restrictive dataflow model than KPN satisfies this condition and allows a valid M2I transformation from SPW and Simulink models to KPN based specifications.

Like most HLDEs, both Simulink and SPW employ a hierarchical syntax as shown in figure 2. KPN and its more restrictive forms like SDF and HSDF are however non-hierarchical by default. This necessitates a flattening of the hierarchical process nework of the input model, while preserving the dataflow properties. This requires a careful renaming of the inner blocks and channels like those within the `Audio Filter Chain` block shown in figure 2 to avoid namespace conflict for the case of multiple instantiations.

A critical discrepancy often found between the frontend and the backend tools is their different interpretations of the channel accesses. In dataflow models like KPN, SDF and HSDF, the read operations are strictly blocking and destructive. This means the read operations block in the absence of sufficient tokens and upon reading, the tokens are removed from the channels. MoC based backend tools tend to enforce these channel access semantics. However, frontend tools like SPW are designed to model both hardware and software systems, where it is convenient for the channels to behave analogously to the *signals* of HDLs. So they do not implement stringent token based behaviour like blocking and destructive channel reads. This may result in unbalance between reading and writing of tokens on the channels when an application undergoes M2I transformation and result in deadlocks/overflows. Although this can be detected in case of HSDFs, where fixed number of tokens are read and written for every firing of a task, it is not possible to track this unbalance in case of KPNs, where the number of tokens produced and consumed per firing of an actor is not fixed a priori. Thus, for high level models based on KPNs, these unbalances must be avoided by the developer.

If the prerequisites are met, the input model is transformed into an IR generic enough to capture the application details from the high level models employed by different HLDEs, which can be efficiently transformed into MoC based representation for the backends.

B. The M2I IR

In general, a hierarchical block diagram based application model essentially has two components: the *application topology description* (structural) and the *atomic process definitions* (behavioural). In a hierarchical model, a process may be either primitive (or atomic) or composite. While a composite process represents a *sub-block diagram* like the `Audio Filter Chain` block shown in figure 2, a primitive or atomic process can not broken down any further into block diagrams and is defined imperatively. The application topology consists of instances of both primitive and composite processes and the channels connecting them. Because of hierarchy, a process can be instantiated within another composite process, where the latter acts as *container* for its constituent processes and channels.

In the M2I IR, the structural information is represented by two sets $\mathcal{I}_\mathcal{P}$ and $\mathcal{I}_\mathcal{C}$ and the relation Γ. The composite process instantiations are stored as a set of relations $\mathcal{I}_\mathcal{C} = \{I_c, C_i, C_o, \Phi_c\}$. The relation $I_c = \langle c_{id}^{'}, inst_{id}, inst\_name, c_{id} \rangle$, denotes the container composite

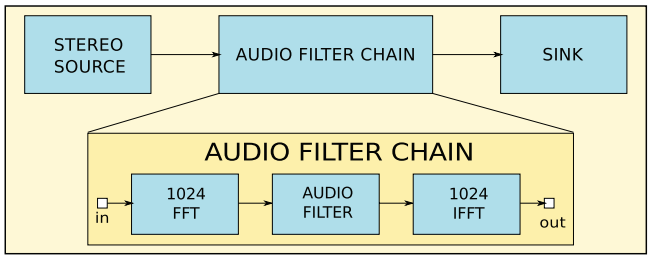

Fig. 2. Hierarchical block diagram specification of an application.

process id (id of the composite process, within which this process is nested), the unique instance id of this composite process instance, the composite process instance name and its type (as defined in T_c below). Both the relations C_i and C_o have tuples of the format $\langle c_{id}^{'}, inst_{id}, \gamma_{id} \rangle$, which denote the container composite process id, the process instance id and the id of the channel to which the input and output ports are connected respectively. These two relations maintain the connectivity information, while the relation $\Phi_c = \langle c_{id}^{'}, inst_{id}, value \rangle$ stores the custom parameter values in the $value$ field. These values can be used to further customize the individual instances of the processes like specifying the port widths. Analogously, the set of relations $\mathcal{I_P} = \{I_p, P_i, P_o, \Phi_p\}$ is used to save the information on primitive process instantiations. Finally the channels are represented by a relation $\Gamma = \langle c_{id}^{'}, \gamma_{id}, name, type \rangle$, which saves the channel name and data type along with its id and container information. The relation Γ along with the sets $\mathcal{I_P}$ and $\mathcal{I_C}$ complete the application topology description.

The behavioural aspect of the application is represented by primitive and composite process definitions. The composite process definitions are characterized by their input and output ports and the sub-process network that they represent, while the primitive processes require the imperative process definition. In M2I IR, an atomic or *primitive* process structure is represented by the set of relations, $\mathcal{P} = \{T_p, I_p, O_p, P_p\}$. Here, the relation $T_p = \langle id, type, path \rangle$ denotes the id, type and library location of a primitive process respectively. The relations I_p and O_p represent the input and the output port information and have the format $\langle p_{id}, name, type, rate \rangle$, denoting the primitive process id, port name, port data type and channel width respectively. And the relation $P_p = \langle p_{id}, name, type \rangle$ denotes custom parameter names and data types associated with a primitive process respectively. Analogously, a composite (non-atomic) process definition is translated into a similar set of relations, $\mathcal{C} = \{T_c, I_c, O_c, P_c\}$. These sets of relations can represent the skeleton of a general process modelled by HLDEs. Each of the primitive process definitions is reduced to an AST. An additional relation $\Psi = \langle p_{id}, ptr_{AST} \rangle$ is used to conect the ASTs of atomic process implementations to primitive process types in T_p.

Since the M2I IR captures all the essential aspects of general hierarchical block diagrams, it can capture the characteristics of high level models generated by different frontend tools. This representation scales well with the input size and can accommodate very large and complex inputs. Furthermore, since the IR does not impose any semantic restrictions of the process network model that it represents, upon flattening, it can act as the precursor to KPN based specifications or even more relaxed dataflow models, making is amenable to code generation for MPFs.

C. M2I Implementation

The Figure 3 shows the architecture of the M2I framework. The frontend of M2I consists of *high-level model parsers* (or simply *model parsers*) for each high-level modelling environment brought under for M2I transformation. Each model parser comprises a *topology parser* and a *source parser*. These model parsers generate the IR described above. The M2I backend comprises a *code emitter* for each of the targeted MPSoC programming frameworks. The components that are not yet implemented are shown in white.

The M2I transformation framework can be extended to accommodate coupling of wide range of tools from the two *tool families*. The interface between the *frontend* and the *backend* of this framework is based on a generic IR. As a result, extending the framework for accommodating new frontends amounts to the development of appropriate model parsers, which can generate this IR. Similarly, bringing more backends under the perview of this transformation requires the development of appropriate code generators, which can process this IR.

Topology Parser: Topology parsers analyse and parse the topological representation of an application used by the HLDEs. They save the topology of the input application by populating all the relations of the M2I IR except Ψ.

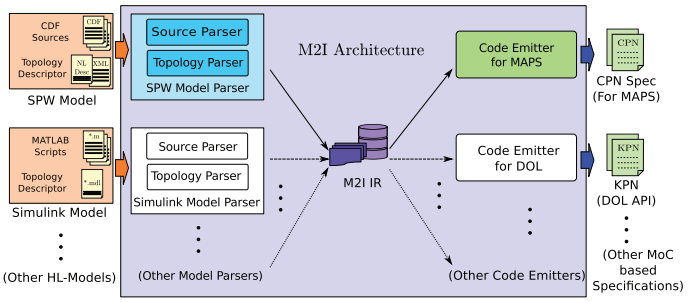

Fig. 3. The architecture of the M2I transformation framework.

Source Parser: The source parsers parse the source files that define the behaviour of the atomic processes. The source code is translated to an AST and the relation Ψ is populated by saving a pointer to the AST for each atomic process present in an application. Along with the set $\mathcal{P}$, the relation Ψ completes the process definition of each atomic process.

Code Emitter: Code emitters generate MoC based specification of the application from the M2I IR. For example, the code emitter for the MAPS backend generates CPN specification which is based on the KPN MoC. Code emission has two major subtasks, viz. instantiation of the process network topology and generation of the process definitions.

Network Topology Instantiation: This stage involves flattening the hierarchical information saved in the IR. The complete application is initially viewed as a single *composite process*, which has exactly one instance and the sets $\mathcal{I_P}$ and $\mathcal{I_C}$ provide its complete definition of its internal structure. Starting from this top level composite process, each composite process is recursively replaced by the sub block diagrams that it represents, until the process network consists only of atomic processes using information saved in $\mathcal{I_P}$, $\mathcal{I_C}$, $\mathcal{P}$, $\mathcal{C}$ and Γ.

Process Definition Generation: The set $\mathcal{P}$ in conjunction with ASTs saved by the relation Ψ are used to generate the process definitions. The semantic differences between the frontend and backend specifications need to be handled during this task. This includes the differences in interpretation of channel accesses mentioned in section IV-A. In case, an unbalance is detected between reading and writing rates of tokens for a channel (only possible if HSDF MoC is used by the frontend), an error is reported.

D. Limitations of M2I Transformation

Presently the M2I transformation requires access the complete source code of the application under development, restricting linking to third party pre-compiled libraries, for which

sources are not accessible. However this limitation can be bypassed, if those pre-compiled libraries are made available for the architectures for which the implementation is to be targeted.

V. Experimental Validation

To demonstrate the efficacy of the COMPOSE design flow in design space exploration, performance enhancement and productivity boosting, an application with some of the commonly occurring embedded multimedia computation kernels was explored. The goal was to start with a simple high level representation of the application and adapt it to three different targets, viz. on a workstation, the TI OMAP 3 [22] and the TI Keystone [23] platforms. The workstation runs on an AMD Phenom II X4 965 quad-core processor (a 4-core homogeneous CISC platform). The TI OMAP 3 is a heterogeneous MPSoC, comprising an ARM Cortex A8 core, along with a TMS320C64x+ DSP and a 2D/3D graphics accelerator and the TI Keystone is a homogeneous platform based on TI C6678 processor, featuring eight C66x DSP cores.

The process network structure of the application used for the experiment is shown in Figure 4. The `source` process reads time domain audio signal from a file; the `FFT` process transforms that signal to frequency domain; the `filter` process performs the actual signal processing; the `IFFT` process recovers the filtered time domain signal from its spectrum; the `sink` process writes it back to a file. This initial structure of the application and will be referred to as "configuration 0", to distinguish it form the derived configurations.

The application was modelled and verified in SPW and then subjected to M2I transformation, to generate the CPN code (see [12] for details) for the MAPS framework. For adapting the application best to the needs of the specific target platforms, it was first profiled against all processor types on the target platforms using the task level performance estimation feature of MAPS. The profiling results shown in Figure 5 show that irrespective of the target processor type, the `FFT` and `IFFT` processes consume about 47% of the total computational time each, accounting for about 95% of the total application workload. The profiling results are particularly interesting for the OMAP platform, as they highlight the heterogeneity of the platform.

Since the two most computation intensive processes form a decently balanced pipeline, the profiling results hint that mapping them to separate cores can enhance the application performance. The profiling results also encourage exploiting the data level parallelism (DLP), since the `FFT-filter-IFFT` chain process each token independently. Based on the profiling hints, the system designer can create variants of the basic application by considering "algorithmic restructuring" at the model level using SPW to better suit different multicore targets. The variants or the "configurations" that we generated for this case study are shown in Figure 6.

The configurations 1 and 2 were created to exploit DLP to different extents by incorporating multiple `FFT` and `IFFT` instances to take advantage of the extra processing power

Fig. 4. The basic audio filter application (Config 0).

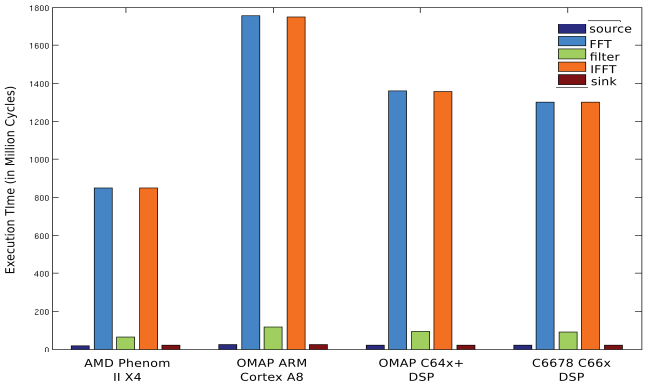

Fig. 5. Task level profiling results for the audio filter application.

offered by the additional cores on the host (4-cores) and TI Keystone (8-cores) platforms respectively. These configurations allowed multiple instances of the most computationally intensive processes in the application to run in parallel, enabling performance enhancement. Configuration 3 was created keeping in mind that the filter process runs much faster than the bottlenecking processes in the application. Configuration 4 was derived particularly for the OMAP platform which aimed at reducing the inter-process-communication overhead by minimizing the number of channel accesses. All these *configurations* of the application were then validated through model simulation in the SPW HLDE. Then they were transformed through M2I and evaluated for performance on the target platforms.

The maximum speed-ups obtained with respect to a sequential implementation for each *configuration* on each platform are listed in table 1. The number of cores utilized is also mentioned alongside. It was observed that a speed-up of upto 2.6X could be obtained on the TI OMAP platform by parallelizing the application and optimally mapping of the tasks to the cores. The reported speed-up for OMAP platform exceeds the expected 2X mark because the sequential application was benchmarked on the ARM core. Expectedly, greater speed-up of almost 6X was obtained for the TI Keystone platform, which offers 8 DSP cores. In contrast, despite a greater degree of parallelism, the maximum speed-up achieved on the quad core host platform was only 2.8X. This demonstrates the effetiveness of mapping most computationally intensive tasks to DSPs.

The COMPOSE design methodology allowed the derivation of target specific versions of the application with very little effort on the part of the application designer (low design

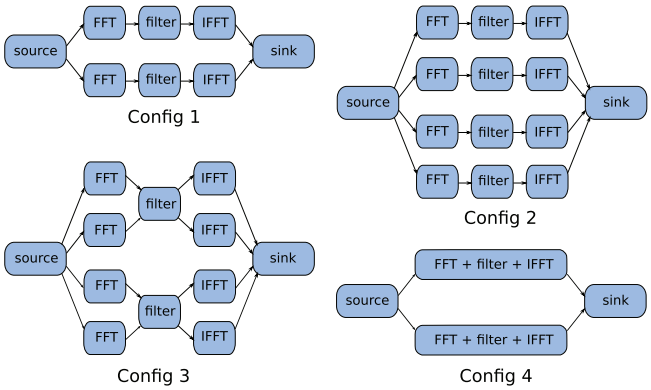

Fig. 6. The various configurations of the audio filter application.

effort of HLDE), which could best utilize the target hardware capabilities. This demonstrates the potential of the COMPOSE approach for opening up new possibilities of target-specific optimizations while exploring the design space.

Config	AMD Phenom II	TI OMAP 3	TI Keystone
0	1.90X *(4 cores)*	1.41X *(2 cores)*	1.91X *(3 cores)*
1	2.77X *(4 cores)*	2.47X *(2 cores)*	3.51X *(8 cores)*
2	2.80X *(4 cores)*	2.61X *(2 cores)*	5.95X *(8 cores)*
3	2.79X *(4 cores)*	2.66X *(2 cores)*	5.89X *(8 cores)*
4	1.86X *(4 cores)*	2.71X *(2 cores)*	2.19X *(2 cores)*

Table 1. Summary of speed-up obtained for the different target platforms.

Productivity Gain

A composite design methodology, when carried out manually will essentially consist of four phases, viz. high level modelling, model simulation and verification, manual transformation to MPSoC friendly syntax and probing the application on the target. Denoting the time taken by these phases by $T_{hl,i}$, $T_{sim,i}$, $T_{man,i}$ and $T_{be,i}$ respectively, where i in the subscripts denotes the iteration, the total time taken for software synthesis will amount to $T_{man,sdlc} = \sum_{i=0}^{n}\{T_{hl,i}+T_{sim,i}+T_{man,i}+T_{be,i}\}$, where $i = 0$ denotes the initial phase. The breakup of these four phases for the development of the audio filter application in our experiment using manual transition from SPW to CPN is shown in the lower portion of Figure 7.

However, by automating the transition from the high level model to MPSoC friendly specification, the transition time is greatly reduced (see Figure 7). The productivity gain (η) thus obtained is computed by using the expression $\eta = (T_{man,sdlc} - T_{m2i,sdlc})/T_{man,sdlc}$, where $T_{m2i,sdlc}$ is the length of the SDLC with automated transition using M2I transformation, which can be computed by the expression $T_{m2i,sdlc} = \sum_{i=0}^{n}\{T_{hl,i} + T_{sim,i} + T_{m2i,i} + T_{be,i}\}$. Here $T_{m2i,i}$ denotes the time for M2I transformation for the *i*-th iteration.

In case of the audio filter application, this transition phase shrinks by 88% raising the productivity of the complete SDLC by 20% as shown in the upper part of Figure 7. In general, the productivity gain achieved by COMPOSE design flow increases with the complexity of the application, because it takes much longer to manually transform more complex applications.

VI. Conclusion

In this paper, a novel approach to embedded multiocre software synthesis was presented in the form of a composite design methodology. The COMPOSE design methodology presented here allows a developer to use the high level application specification to rapidly generate code for multiple MPSoC platforms, resulting in better architecture specific optimization and productivity gain. It was shown how this synthesis approach could be used as a vehicle for adapting complex embedded applications to different target platforms and performing target specific optimizations using a high level model. This design flow provides a powerful mechanism for co-exploring design space and heterogeneous MPSoCs targets, without creating an overly complex tool. Future work will consist of automating the process of algorithmic optimization based on platform exploration hints.

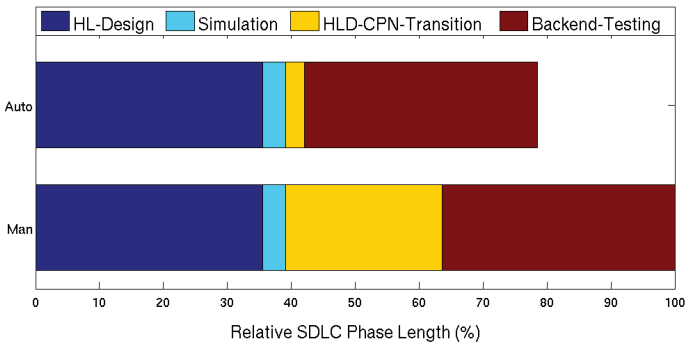

Fig. 7. Relative lengths of the SDLC phases: (Auto) if automated SPW to CPN transtion is used, (Man) if manual SPW to CPN transtion is used.

References

[1] W. Maydl, L. Grunske, *Behavioral Types for Embedded Software - A Survey*, Component-Based Software Development for Embedded Systems p-82-106, Springer 2005.

[2] The International Technology Roadmap for Semiconductors, *International Technology Roadmap for Semiconductors 2011 Edition Emerging Research Devices*, http://www.itrs.net/Links/2011ITRS/Home2011.htm.

[3] Math Works, *Simulink*, http://www.mathworks.de/products/simulink/.

[4] Synopsys, *SPW*, http://www.synopsys.com/Systems/BlockDesign/DigitalSignalProcessing/Pages/Signal-Processing.aspx.

[5] S. A. Edwards, *Languages for Digital Embedded Systems*, Springer, 2000.

[6] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, *Ptolemy: A Platform for Heterogeneous Simulation and Prototyping*, European Simulation Conference, 1991.

[7] W. Ecker, W. Mueller and R. Doemer, *Hardware-dependent Software - Principles and Practice*, Springer, 2008.

[8] J. Castrillon, L. Thiele *et. al.*, *Multi/many-core programming: Where are we standing?*, DATE, 2015.

[9] J. Castrillon, W. Sheng and R. Leupers, *Trends in Embedded Software Synthesis*, SAMOS, 2011.

[10] 3GPP Group, *LTE Standard*, http://www.3gpp.org/article/lte.

[11] Google Inc., *WebP Home*, https://developers.google.com/speed/webp/.

[12] J. Castrillon, R. Leupers and G. Ascheid, *Mapping Concurrent Dataflow Applications to Heterogenous MPSoCs*, IEEE Transactions on Industrial Informatics, Vol. X, 2011.

[13] S. Kwon *et. al.*, *A Retargetable Parallel-Programming Framework for MPSoC*, ACM Transactions on Design Automation of Electronic Systems, 2008.

[14] L. Thiele *et. al.*, *Mapping Applications to Tiled Multiprocessor Embedded Systems*, IEEE Computer Society, 2007.

[15] H. Nikolov, *System-Level Design Methodology for Streaming Multiprocessor Embedded Systems*, PhD Thesis, Universiteit Leiden, 2009.

[16] E. A. Lee *et. al.*, *Gabriel: A design Environment for DSP*, IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, 1989

[17] S.Ritz, M. Pankert, V. Zivojnovic and H. Meyr, *High-Level Software Synthesis for the Design of Communication Systems*, IEEE Journal on Selected Areas in Communications, Vol. 11, 1993.

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

[19] G. Kahn, *The Semantics of a Simple Language for Parallel Programming*, Information Processing, 1974, Proceedings of IFIP Congress, 1974.

[20] S. Tripakis *et. al.*, *Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs*, ACM Transactions on Embedded Computing Systems, Volume 12 Issue 3, 2013.

[21] J. Castrillon *et. al.*, *Trace-based KPN Composability Analysis for Maping Simultaneous Applications to MPSoC Platforms*, DATE, 2010.

[22] Texas Instruments, *OMAP 3*, http://www.ti.com/omap3.

[23] Texas Instruments, *TI Keystone*, http://www.ti.com/product/TMS320C6678.

[24] F. Thoen *et. al.*, *Real-Time Multi-Tasking in Software Synthesis for Information Processing Systems*, Proceedings of International System Synthesis Symposium (ISSS), 1995.