B. Juurlink, W.Karl (Hrsg.)
Proceedings 28th PARS Workshop

Generating Optimized FPGA Based MPSoCs to Parallelize Legacy Embedded Software with Customizable Throughput

Kris Heid[1], Christian Hochberger[1]

Abstract: Executing legacy software on newly developed systems can lead to problems regarding the required throughput of the software. Automatic software parallelization can help to achieve a desired exection time even if a single core version would be to slow. In this contribution, we present a toolset that automatically parallelizes a given legacy software and distributes it to multiple soft-cores forming a processing pipeline. As a goal for the parallelization, the user can provide a minimum throughput that has to be achieved. Although this concept is limited to repetitive tasks, it can be well applied to most embedded system applications. The results show that the tool achieves remarkable speedups without any manual intervention or code restructuring for a sprectrum of benchmarks.

Keywords: automatic parallelization; embedded; soft-core; MPSoC; µStreams

1 Introduction

Reusing existing software is often proposed as a good way to reduce the cost of newly developed embedded systems. Such existing code might implement an application that must obey (soft) real time conditions. To meet these conditions, the performance of a single processing core might not be sufficient and distributing the code over multiple cores could present a solution to this problem.

Automatic parallelization of software is a research topic for many years now. Yet, most of the resulting tools cannot be used for embedded systems, as they either demand a particular infrastructure or programming interface (like CUDA or MPI) or require special OS support.

In this work, we focus on a particular software structure that in turn can be parallelized fully automatically. We expect that the software executes a repetitive task that can be broken down into smaller execution units (thus forming kind of a processing pipeline). Such a structure is often found in embedded software.

To fully exploit this structure, the number of pipeline stages and the communication between different stages must be adapted to the needs of individual applications. Thus, the shown

[1] Technische Universität Darmstadt, Fachgebiet Rechnersysteme, Merckstr. 25, 64283 Darmstadt, Germany
heid@rs-tu-darmstadt.de, hochberger@rs.tu-darmstadt.de

approach is best suited for application specific System-On-Chip (SoC) solutions. These SoCs can either be implemented as a custom chip, which would demand very high quantities, or it could be implemented on a Field Programmable Gate Array (FPGA) which can easily be used even for small quantities. Leveraging FPGAs also allows easy future extensibility with more peripherals or cores.

On an FPGA, the processing cores should be realized as so called *soft-cores*, making it possible to select an arbitrary number of cores and a custom communication infrastructure depending on the application requirements.

In this contribution, we present our fully automatic tool flow that enables the user to generate a SoC architecture with multiple cores and distribute the legacy software to these cores, such that a user specified throughput is achieved.

The remainder of this paper is structured as follows: The next section discusses the related work, particularly other approaches to parallelize software. Section 3 explains our used target architectures. Section 4 then presents our tools and how they work together. An evaluation using multiple benchmarks from different domains is given in Section 5. Finally, a conclusion and an outlook follow.

2 Related Work

Parallelization approaches have already been studied for the past 30 years. For parallelization, different concepts have been proposed: Domain specific languages (DSLs), language extensions, application programming interfaces (APIs), libraries, annotation based parallelization and automatic parallelization. In the order of appearance, the concepts usually make the initial effort for parallelization smaller and smaller, since it is hard to learn new programming languages or explore possibilities of a new API. Nowadays, annotation based parallelizers are relatively popular due to small user effort and good results. OpenMP[CJvdP07] is the de facto standard in this domain and it is adapted to many platforms. This is mainly since OpenMP managed to integrate step by step many features proposed by former competitors like StarSs[La12] with heterogeneous platforms or the system presented by Yang et al.[YL09] with the focus on distributed-memory architectures.

However, automatic parallelization is still the holy grail since it requires no user effort. Many automatic compilers like SUIF[Ha96], Intel C Compiler, Pluto[Bo08] or Daedalus[Th07] only focus on parallelizing (affine nested) loops, which only meets the characteristics for some applications. Other tools like PIPS[KJA15], Eldorado[CMM10] and AutoPar[Li10] extend their parallelization capabilities to the whole program. AutoPar not only focuses on loop parallelization, but also on creating tasks from functions through annotating the source-code with OpenMP loop and task pragmas. PIPS creates a data dependency graph from the source-code. The runtime of the source-code parts is estimated through assembler code analysis and parts of the source-code are mapped to the hardware with a resource aware

list scheduling. PIPS extends the source-code with OpenMP pragmas or generates message passing interface (MPI) code. Eldorado generates a hierarchical control flow graph (CFG) and applies integer linear programming to find an optimal solution with respect to available processor threads, task creation overhead and communication overhead. Compared to PIPS, Eldorado uses a, presumable more inaccurate, average execution time per C source-code statement.

All aforementioned parallelization tools have the following flaws that we want to address in this work:

- Most tools try to extract as much parallelism as possible. Specifying a desired speedup and creating appropriate parallelizations in terms of hard- and software is not considered.
- Data and task parallelism is mainly used. Pipeline parallelism is rarely used even though it is very beneficial for regularly repetitive tasks (like in many embedded systems).
- Even parallelizers targeting embedded systems do not focus on conflicts through simultaneous peripheral access by multiple cores.

3 Target Platforms

As target platform we have used two soft-core multi-processor system-on-chips (MP-SoCs) kits:

SpartanMC is our own 18-Bit architecture where we have full control over the environment. The uncommon 18-Bits optimally leverage todays FPGAs, whose internal structures ideally fit 18 bits. The SoC kit contains a system-builder, simulator, GCC, Binutils, GDB, and synthesis toolchain support for Xilinx, Altera(Intel) and Lattice FPGAs.

MicroBlaze is Xilinx' 32-Bit soft-core. In contrast to other vendor soft-cores like Altera's Nios and LatticeMicro32, it is widely used and in our experience the support for MP-SoCs is better. The available tool environment has similar extend as SpartanMC.

A MP-SoC in both architectures is a distributed-memory system with communication through custom point-to-point as well as point-to-multipoint communication peripherals. Xilinx provides the Mailbox for bidirectional FIFO based communication. The Mailbox can either be configured with an AXI4-Lite interface, implementing memory-mapped IO or an AXI-Stream interface, requiring special processor instructions to access it. The Stream interface should be used for throughput demanding applications, since it has a significantly higher throughput and lower latency. For point-to-multipoint connections we use multiple Mailboxes in parallel.

SpartanMC has a 1-1 core-connector, 1-N dispatcher, N-1 concentrator and an optional shared memory region as communication peripherals. The core-connector is very similar to the Mailbox, except being unidirectional. The concentrator and dispatcher have a round-robin arbiter or software based arbitration to distribute or receive data in a fair manner. All SpartanMC interconnect peripherals are either memory-mapped FIFO buffers or implemented as DMA peripheral memory regions. DMA core-interconnects in SpartanMC require at least two dual-ported RAM primitives (BRAMs), one output of each RAM is connected to core 1, the other to core 2, integrating seamlessly into the core's address space. Only one RAM port to each core is active at a time and a data transfer simply changes the active port on both sides. Thus, the duration of a data transfer of arbitrary size takes 1 cycle, while the non DMA variant requires about 2 cycles per 18bit word. However, BRAM is often a very limited resource and the RAM size must be chosen big enough to contain the maximum message size, while message size is irrelevant for the FIFO based transmission.

4 Tool Flow

We proceeded in different steps with a dedicated tool for each step to realize a parallelized hard- and software design. This decision makes the approach easily adaptable to other Soft-Core MP-SoCs. Additionally, human readable and modifiable files are generated after each intermediate step. This allows the user to step in and modify the design to his wishes or requirements. The overall approach is shown in Fig. 1 and explained in the following.

4.1 AutoPerf: Application Profiling

The legacy sequential source-code is firstly profiled with AutoPerf[HWH18a]. The code is instrumented with calls to the SoCs performance-counter or timer. Each statement is embraced by a call to start the performance-counter and afterwards read and reset it. By default, the main function is profiled, but the user can decide to profile other functions through pragma annotations as-well. The instrumented source-code and an abstract hardware configuration is provided as output. The single-core design can be synthesized and run on the FPGA after automatically importing and building a design in the system-builder (jConfig). The user has to take care to provide an average or worst case environment for the peripheral interaction during measurement. A performance-profile is printed via UART or similar after the application finished.

4.2 AutoStreams: Automatic Annotations

The produced performance profile and the original source-code is feed to AutoStreams[HH18]. The user then specifies the desired throughput of the application. AutoStreams will parse the source-code into an Abstract Syntax Tree (AST). A CFG of the

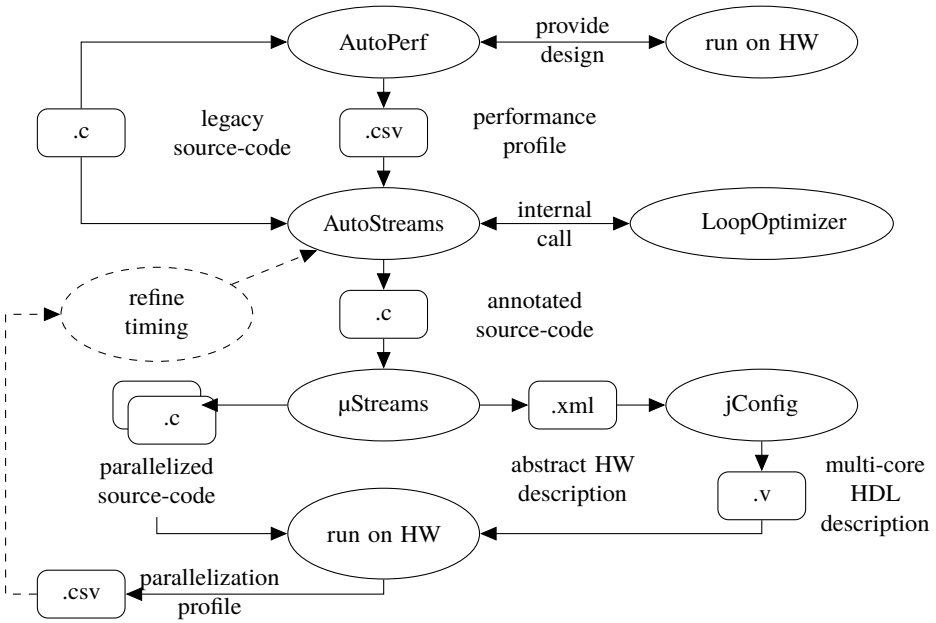

Fig. 1: Automatic parallelization toolflow

profiled source-code parts is created. A CFG node then represents a statement of the profiled function. Often, loops consume vast parts of the overall application runtime. Loops are automatically partitioned into multiple smaller loops with the LoopOptimizer (see Section 4.2.1) if required. This decreases the time granularity and increases amount of CFG nodes.

Afterwards, AutoStreams tries to partition the application to use as few hardware as possible to achieve the user defined throughput. To prefer the smallest possible hardware configuration, an analytical model for each core and core-interconnect has been previously evaluated and added to a AutoStreams hardware usage table. Additionally, the required communication time per communicated data size has been evaluated and is reflected in an approximation function for each core-interconnect. Through these analytical models, an approximation can be done whether it is better to use more cores with slow communication or less cores with fast, but hardware (BRAM) costlier communication peripherals for example.

Now, design space exploration can start. Evaluation of all possible solution takes to long, due to the vast amount of possible solutions. Thus, a branch-and-bound method was chosen. Firstly, a non optimal search heuristic is applied, delivering a solution limiting the search space in the second phase for an optimal solution (detailed description in [HH18]).

To select the best of all solutions fulfilling the timing requirements, firstly pipelines with unique peripherals per pipeline stage are selected, since access to the same peripheral

from different cores is not safely possible [HWH16b]. This behavior can be ignored via command line flags. Secondly, configurations with smallest hardware (LUT) overhead are preferred. Thirdly, amongst minimal hardware designs, the ones with the highest throughput are chosen. Thus, the most economic solution fulfilling user requirements is selected.

The used partitioning is reflected in injected source-code annotations which can be reviewed and if desired manipulated by the user.

4.2.1 LoopOptimizer: Loop Optimizations

Creating balanced pipelines doesn't work well when single statements or CFG nodes (often loops) dominate execution time. Placing pragmas inside loops will create pipeline backward dependencies, prohibiting the benefits of the pipeline. The LoopOptimizer[HWH18b] implements loop splitting and loop fission to provide AutoStreams with loop restructuring and more possibilities of split points.

Loop fission finds independent statements in a loop. All independent statements are partitioned into a separate loop while the loop condition remains identical for all loops.

Loop splitting is a method to distribute the iterations of the original loop to several loops handling a fraction of the original iterations. Thus, the iterations are partitioned while the body stays the same.

4.3 µStreams: Annotated Source-Code Transformation

The target of µStreams[HWH16a] is to split up the original source-code at the pragma annotations into several chunks, forming a processing pipeline. Each pipeline stage will do a fraction of the work of the original application and pass results to the next pipeline stage proceeding in the same way. Thus, the first core is able to handle new data inputs in shorter intervals, increasing the applications throughput. Currently, µStreams has only one pragma: `#pragma microstreams task` with the option to specify `replicate` **number of replicas** to make non dividable pipeline stages superscalar. Pragmas can be placed before any statement, function and inside called functions, but not inside loops or recursive functions. Dependencies between the partitioned source-code chunks are automatically identified and communication infrastructure in software and hardware is automatically created. µStreams is also able to detect used peripherals at the different source code parts based on used APIs and variable types [HWH16b]. One firmware file per core is written as C source-code at the end of modification. Additionally, an abstract hardware description (XML) is created, specifying processor cores, core-interconnects and peripherals. The user also has the option to add evaluation peripherals, which automatically measure the performance of the parallelized design. The abstract hardware description and the firmware sources can be imported into the system-builder (jConfig), automatically connecting and

building the components. The system can then be synthesized and compiled to be run on an FPGA.

4.4 Refine Timing Constraints

The user can review the optionally, but automatically generated parallel performance profile after execution on the FPGA and match it against the requirements. We identified two causes for not fulfilled throughput requirements: Different GCC compiler optimizations and different complex loop iterations of partitioned loops. Varying GCC compiler optimizations can be applied when compiling the parallelized and partitioned code separately, compared to the profiled initial single core compilation, resulting in slower or faster program execution. Partitioned loops in AutoStreams are treated as if each iteration has the same runtime share of the overall loop runtime. This is of course not always the case but a valid approximation for most loops. In both cases, restricting throughput requirements will create a valid design after a few refinement iterations.

4.5 Contributions of this Work

Compared to previous publications of the presented tool-chain, we have made advancements in the following sections:

- We have extended all tools from SpartanMC support to Xilinx MicroBlaze, another soft-core MPSoC. This should show the wider applicability of the approach.
- For the SpartanMC, DMA like core-interconnect peripherals have been created to further decrease critical inter-core communication overhead. The peripherals have been integrated in Verilog and the peripheral characteristics have been added in the analytical model of AutoStreams.
- The pipeline replication mechanism has been integrated in µStreams and AutoStreams to enable superscalar pipeline stages. For replicated pipeline stages the 1-to-N and N-to-1 peripherals are used to distribute and collect application state. We want to analyze the benefits and drawbacks of these constructs and how well the superscalar pipeline improves throughput compared to the previously used pipeline.

5 Evaluation

5.1 Used Benchmarks

To determine the usefulness of the proposed toolchain, we evaluate four benchmark programs: ADPCM, MJPEG2000, IIR filter and a firewall.

Adaptive differential pulse-code modulation (ADPCM) is a digital signal compression algorithm widely used in mobile low-power wireless sensing applications. Infinite impulse response (IIR) Butterworth Filter represents a 5th order high-pass filter. IIR filters are typically used instead of FIR filters for filtering time continuous signals in embedded environments since they require significantly less processing power at an equivalent accuracy. Motion JPEG 2000 Encoder (MJPEG2000) is a video compression algorithm. Compared to other video compression algorithms MJPEG2000 has no inter frame dependencies making it well suitable for parallelization. The firewall implementation contains static as well as dynamic filter rules (managing open TCP connections). We used our research group's (~20 people) firewall as sample for the number of static rules and open TCP connections, since the performance of a firewall mainly depends on the number of rules.

5.2 Results

Tab. 1: Results of automatic parallelization and according hardware cost

		SpartanMC				MicroBlaze		
Benchmark	req.[1]	act.[2]	LUTs	DMA	Cores	act.[2]	LUTs	Cores
ADPCM	2	2.2	3.1	x	3	2.7	3.0	3
	4	3.4	5.2	x	5	4.4	5.0	5
	8	7.2	10.3	x	10	8.0	9.1	10
	12	10.8	15.6	-	17	15.7	20.3	20
IIR Filter	2	2.1	2.7	-	3	2.0	2.3	3
	4	3.9	5.2	x	5	3.8	9.3	12
	8	6.0	12.5	x	12	exceeds FPGA res.		
	12	8.1	25.2	x	24	exceeds FPGA res.		
MJPEG	2	2.6	3.0	-	3	2.2	3.0	3
	4	4.4	6.3	-	6	3.5	5.0	5
	8	exceeds FPGA resources						
Firewall	2	3.1	2.6	-	4	1.8	2.3	4
	4	4.5	3.0	-	5	3.5	3.2	6
	8	7.4	3.5	-	7	5.1	5.1	10
	12	7.4	3.5	-	7	5.2	6.0	12

[1] user requested speedup | [2] actual speedup after parallelization

For the benchmarks, we set a user performance requirement to AutoStreams that should be achieved. The timing requirement reflects a 2x, 4x and if possible 8x and 12x higher throughput compared to the single core performance throughput of SpartanMC or MicroBlaze respectively. We allowed AutoStreams to use DMA and non DMA core-interconnects and superscalar core replication. After parallelization, we've synthesized the generated design for a Xilinx Artix-7 XC7A200T FPGA and finally ran the design while evaluating hardware overhead and achieved speedup.

As seen in Tab. 1, the performance demands of the 2x speedup requirements has almost always been fulfilled. To achieve a 2x speedup, at least three processing cores are leveraged. In the ideal case, two cores should be sufficient to achieve this speedup if we neglect communication overhead. Thus, a quite high overprovisioning was chosen. As speedup requirements increase, a smaller overprovisioning is chosen, since AutoStreams takes the smallest possible hardware just fulfilling the throughput requirements. However, due to the applied single core compiler optimizations which might not be applicable anymore on the partitioned parallelized code, AutoStreams thinks the requirements are met while the measured throughput is below the requirement. Surprisingly, the SpartanMC observed throughput was often 20% to 30% below the the expected throughput, while for MicroBlaze systems the observed throughput was only around 10% below the expected. Most likely the cause lies in the different compiler optimizations for both systems, since deactivating compiler optimizations on one benchmark revealed no discrepancy between the estimated and the actual throughput. Thus, for all benchmarks the user could make a second parallelization run with accordingly tightened timing restrictions.

Considering the hardware cost of parallelized design, one can see that for a higher parallelization unproportionally more cores are required. Firstly, it is often impossible to generate a balanced pipeline where each core has the same workload due to the finite granularity of the working packages to distribute. Secondly, the application state has to be passed through the pipeline which is the communication overhead. The longer the processing pipeline gets, the smaller are the workloads per core and thus communication overhead grows relatively. Comparing the IIR benchmark (much application state information) and the ADPCM benchmark (few application state information) one can see that for a 8x and 12x speedup, IIR needs a longer processing pipeline to fulfill the requirements. However, looking at the increase of actual used hardware (LUTs) on the FPGA, one can see that this number does not always grow proportionally with the amount of cores. For example the SpartanMC 17-core ADPCM example only uses 15x more LUTs compared to the single core variant, since the synthesis is able to reuse some hardware. In this evaluation we've just focused on the used LUTs but the other FPGA components (BRAMs, registers and DSPs) grow at similar rates.

We have restricted the available FPGA's resources for the ADPCM benchmark's SpartanMC builds to 200 BRAMs (since BRAMs are often the limiting resource), simulating a smaller FPGA and intentionally triggering AutoStreams to use BRAM preserving non DMA core-interconnects on resource shortage. This resulted in the 12x speedup case to use non DMA core-interconnects and thus still be synthesizable on a smaller FPGA.

The IIR example reveals a different aspect. The 2x speedup requirement is realized though non DMA core-interconnects for the SpartanMC. However, for tighter user requirements (4x,8x,12x), DMA variants are chosen. Compared to the workload size, the communication overhead grows, favoring the faster DMA core-interconnects. To achieve the same speedup with non DMA core interconnects an additional processing core would be required costing more hardware than the DMA-core interconnects. Also, a 12x speedup parallelization

without DMA interconnects is not possible since the communication overhead is bigger than the pipeline stages duration requirement. For the MicroBlaze variant, even a 8x speedup is not possible anymore since this would result in a 50 core design vastly busting the bounds of our used FPGA.

The MJPEG2000 benchmark requires a lot of hardware resources (BRAMs) since the full frame is transferred through the pipeline and stored in each core. Also, the communication overhead is quite big through the transferal of the whole frames. DMA core-interconnects for the SpartanMC system can not be used since the available address space would be exceeded. Through the large communication overhead, the necessary core number quickly increases and through the required memory per core, the FPGA's available BRAM is exceeded, preventing further parallelization. Even though not shown, AutoStreams suggests the usage of replicas/superscalar pipeline stages for a 8x speedup since inseparable code parts would exceed the timing requirements. Comparing the hardware cost of a regular pipeline with a superscalar pipeline revealed that a regular pipeline uses fewer hardware at the same throughput. This fact is also reflected in AutoStreams observed behavior to only use replication if no further sequential pipeline stages are possible.

The firewall benchmark's generated hardware consists of a packet receiving core, multiple filter cores connected through a 1-N dispatcher and a packet sending core. Thus, all filter cores form a superscalar pipeline stage through the replicate pragma. The filter rules are stored in a common memory module connected to all cores. The core receiving and the one sending Ethernet packets become the bottleneck, visible through a saturation effect for a 8x and 12x speedup requirement. The sending and receiving cores can't be parallelized since they are directly interfacing the Ethernet hardware modules and AutoStreams even throws a warning that the desired speedup is unreachable. But the saturation effect also comes from the common filter table memory where all filter cores compare the packet header against.

6 Conclusion

In this contribution we have presented a toolset for the fully automatic parallelization of legacy software. The user only has to specify a required throughput and the tool then finds a suitable HW setup of a multi-core system with minimal area requirement and the corresponding distribution of the legacy software. Even if the expected software structure is limited, the tool is applicable to most embedded systems applications.

In the future, we want to improve the communication primitives that are used together with the MicroBlaze core. The Mailbox provided by Xilinx exhibits quite a large latency. Also, we think that the handling of global variables in the legacy code can be improved.

References

[Bo08] Bondhugula, Uday; Baskaran, Muthu; Krishnamoorthy, Sriram; Ramanujam, J.; Rountev, A.; Sadayappan, P.: Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In: ETAPS CC. 2008.

[CJvdP07] Chapman, Barbara; Jost, Gabriele; van der Pas, Ruud: Using OpenMP. Mit University Press Group Ltd, 2007.

[CMM10] Cordes, D.; Marwedel, P.; Mallik, A.: Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS). 2010.

[Ha96] Hall, M. W.; Anderson, J. M.; Amarasinghe, S. P.; Murphy, B. R.; Liao, Shih-Wei; Bugnion, E.; Lam, M. S.: Maximizing multiprocessor performance with the SUIF compiler. Computer, 29(12):84–89, Dec 1996.

[HH18] Heid, Kris; Hochberger, Christian: AutoStreams: Fully Automatic parallelization of Legacy Embedded Applications with Soft-Core MPSoCs. In: ReConFig 2018; International Conference on Reconfigurable Computing and FPGAs. Dec 2018.

[HWH16a] Heid, Kris; Weber, Jan; Hochberger, Christian: µStreams: A Tool for Automated Streaming Pipeline Generation on Soft-core Processors. In: 2016 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC). pp. 25–30, May 2016.

[HWH16b] Heid, Kris; Wirsch, Ramon; Hochberger, Christian: Automated Inference of SoC Configuration through Firmware Source Code Analysis. In: FSP 2016; Third International Workshop on FPGAs for Software Programmers. pp. 1–9, Aug 2016.

[HWH18a] Heid, Kris; Wenzel, Jakob; Hochberger, Christian: Fast DSE for Automated Parallelization of Embedded Legacy Applications. In: Applied Reconfigurable Computing. Architectures, Tools, and Applications. pp. 471–484, 2018.

[HWH18b] Heid, Kris; Wenzel, Jakob; Hochberger, Christian: Improved Parallelization of Legacy Embedded Software on Soft-Core MPSoCs through Automatic Loop Transformations. In: FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers. Aug 2018.

[KJA15] Khaldi, Dounia; Jouvelot, Pierre; Ancourt, Corinne: Parallelizing with BDSC, a resource-constrained scheduling algorithm for shared and distributed memory systems. Parallel Computing, 41:66 – 89, 2015.

[La12] Labarta, Jesús; Marjanovic, Vladimir; Ayguadé, Eduard; Badia, Rosa M; Valero, Mateo: Hybrid Parallel Programming with MPI/StarSs. IOS Press, May 2012.

[Li10] Liao, Chunhua; Quinlan, Daniel J.; Willcock, Jeremiah J.; Panas, Thomas: Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions. International Journal of Parallel Programming, 2010.

[Th07] Thompson, M.; Nikolov, H.; Stefanov, T.; Pimentel, A. D.; Erbas, C.; Polstra, S.; Deprettere, E. F.: A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: CODES+ISSS. 2007.

[YL09] Yang, Chao-Tung; Lai, Kuan-Chou: A Directive-based MPI Code Generator for Linux PC Clusters. J. Supercomput., 50(2):177–207, November 2009.