PACT: G: Towards Effective Parallelization and Accelerator Offloading for Heterogeneous Multicore Embedded Systems

Miguel Angel Aguilar
Institute for Communication Technologies and Embedded Systems
RWTH Aachen University, Germany
aguilar@ice.rwth-aachen.de

PhD Advisor: Prof. Rainer Leupers

ABSTRACT

The use of Multiprocessor Systems on Chip (MPSoCs) is a common practice in the design of state-of-the-art embedded devices, as they provide a good performance, energy and cost trade-off. However, MPSoC programming is still an error-prone and time consuming task that currently involves several manual steps. This task becomes even more challenging for heterogeneous architectures, where general purpose processors are combined with accelerators. In this work, we describe an approach that jointly addresses extraction of multiple parallel patterns and accelerator offloading from sequential applications targeting heterogeneous MPSoCs. Results show the effectiveness of our approach, as we are able to speedup sequential benchmarks significantly on two commercial platforms.

CCS Concepts

•**Computer systems organization** → **Embedded software;** •**Computing methodologies** → *Parallel programming languages;* •**Software and its engineering** → **Compilers;**

1. PROBLEM AND MOTIVATION

MPSoCs have emerged as a response to the demands of applications in the embedded industry such as performance, power and cost. The current design trend of state-of-the-art MPSoCs is towards heterogeneous architectures, where cores with different attributes, such as Instruction Set Architectures (ISAs) and clock frequencies are combined. Typically these platforms integrate general purpose processors, such as ARMs, with special purpose cores, such as DSPs or GPUs. In order to exploit the full potential of an MPSoC, applications have to be optimized for these platforms. This is a challenging task that has been addressed in two complementary directions: *i*) *via new parallel programming paradigms*, and *ii*) *using automatic parallelization frameworks.*

Several programming paradigms have been proposed to express parallelism, such as the OpenMP industry standard [3]. In addition, in the embedded domain dataflow Models of Computation (MoCs), such as Kahn Process Networks (KPNs) [7] have gained acceptance, since they are suitable for embedded streaming applications [8]. These MoCs describe applications as a network of processes, which exchange data by means of FIFO channels. *C for Process Networks* (CPN) [2] is an example of a commercial solution that allows to implement KPNs.

Despite the efforts for providing a convenient parallel programming paradigm, the programmer still has the cumbersome task of writing correct parallel code. In practice, this task is even more challenging considering that in general software development is about evolution and transformation [12], where developers have to optimize legacy sequential code for MPSoCs. This is an extremely error-prone and time consuming process that involve several manual steps: identifying computational intensive code sections, identifying data dependencies, extracting profitable parallelism opportunities, deciding in which type of core to map code sections in heterogeneous platforms, and finally writing correct parallel code. The most prominent solution is to automate these steps by using parallelization frameworks.

Therefore, our research is focused on developing a joint parallelization and offloading method that automate the steps to optimize legacy sequential C applications for heterogeneous embedded MPSoCs, thus improving performance and productivity. Our ultimate goal is to provide an effective toolflow that fulfills the needs of a realistic industrial environment in the embedded domain.

2. BACKGROUND AND RELATED WORK

Research efforts on automatic parallelization have left valuable techniques and observations. However, there is not yet a widely accepted solution and many issues remain open, especially in the embedded domain. Early works focused on extraction of *data level parallelism* from loops with no carried dependencies. While this form of parallelism is abundant in scientific applications, works such as [14, 10] show that in the embedded domain other parallelism patterns should be explored. In addition, current parallelizing compilers rely only on *static information* obtained at compile time. However, this approach often fails in languages like C as it is extremely conservative with respect to the analysis of pointers, dynamic allocated memory and indirect function calls [16]. Therefore, the use of *dynamic analysis* to obtain runtime information is proposed as an alternative or a complement to static analysis [13, 15, 16]. Moreover, existing automatic parallelization frameworks for embedded MPSoCs either do not consider heterogeneous platforms at all [5], or are only capable of targeting platforms with cores of the same ISA that run at different frequencies [6].

3. APPROACH AND UNIQUENESS

Figure 1 shows the main components of the proposed toolflow. It starts with a sequential C application, a model of the MPSoC and constraints provided by the developer to guide the analysis. Then a *program model* that contains both performance and control/data dependency information is built in a hybrid fashion by combining *static* (at compile time) and *dynamic* (at runtime) analyses. The program model is analyzed by algorithms that focus on computationally intensive code regions to identify multiple parallel patterns, and to decide which regions are good candidates for accelerator offloading. In first place the resulting information is used to generate source level hints that provide developers with a deep understanding of parallelization and

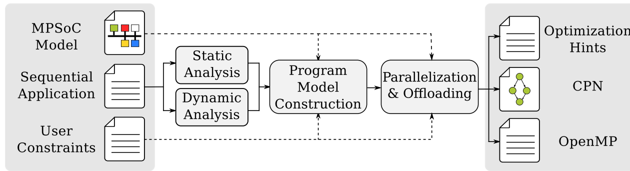


Figure 1: Parallelization and Offloading Toolflow

offloading opportunities. Moreover, the resulting information also enables code generation in a parallel paradigm, such as OpenMP or CPN. In the following sections the main components of our approach are described.

3.1 MPSoC Model

The MPSoC model provides a simplified view of the underlying platform in terms of processing elements and communication resources. This model allows to tailor our approach to the specific characteristics of the target MPSoC. The model describes first of all a *processor type*, which is characterized by properties such as: its role as *host* or *accelerator*, its ISA and its clock frequency. Then a *processing element* is defined as an instance of a given processor type that inherits its attributes. Finally, *communication elements* that allow to transfer data among processors are also modeled.

3.2 Program Model

The main data structure of our approach is the *Program Model*, which describes the input application in terms of: *i*) performance information, *ii*) a dynamic call graph that contains the executed functions and their calling relationships in a given profiling run, and *iii*) a set of graphs that contain control and data dependency information.

3.2.1 Performance Information

Performance information is a fundamental aspect for profitable parallelization and offloading optimizations. In this work it is used for two main purposes: identifying computationally intensive code sections (called here *optimization candidates*), and performing a cost-benefit analysis of the identified parallelization and offloading opportunities. Performance information can be estimated for every processor type in the target MPSoC by multiple methods, such as annotation based, table based, measurement based, analytical or estimation tools [5]. To ensure correct parallelization and offloading decisions we used an accurate commercial solution [2]. In our approach, we consider performance information at the C statement granularity, since this allows a high flexibility for the parallelization and offloading algorithms.

3.2.2 Dynamic Call Graph

A *call graph* enables a complete view of the applications, which is represented as a set of nodes that represent functions and edges that describe calling relationships. In our work we use a *Dynamic Call Graph* (DCG) since we want to focus on the functions that were executed in a given profiling run. In a DCG each function node is annotated with its execution cost and the edges are annotated with the number of times each call took place. From the DCG we extract the *optimization candidates* by using a user-defined threshold that specifies the minimum percentage of the total execution time of the application, which is required to consider a function as a candidate. By identifying the optimization candidates we reduce the problem space and improve the scalability of the approach.

3.2.3 Control and Data Graph Representations

The parallelization and offloading analyses relies on describing functions using two graph representations: *i*) the *Dependence Flow Graph* (DFG) and *ii*) the *Program Structure Tree* (PST) [11]. The basis of both graph representations is the *Single-Entry Single-Exit* (SESE) region, then we start by describing it.

Single-Entry Single-Exit (SESE) Region

A key challenge to optimize applications for MPSoCs is to choose the right granularity for code sections to be parallelized and offloaded to accelerators. In our approach it is used the concept of *Single-Entry Single-Exit* (SESE) region [11]. A SESE region is defined as a connected subgraph, such that there exits a unique incoming control edge *inside* the region, and a unique outgoing control edge *outside* the region. SESE regions are a convenient unit as they enable to perform independent analyses to each region in isolation allowing a divide-and-conquer approach. Moreover, SESE regions can be identified based on the language construct that they represent (e.g. *if-then-else* or *loop*).

Dependence Flow Graph (DFG)

In this work, we use the *Dependence Flow Graph* (DFG) to enable the control and data dependency analysis, which allows to keep the functional correctness of the application while optimizing it for MPSoCs. The DFG is an intermediate representation that incorporates control and data dependencies, and also the SESE regions. Figure 2b shows an example of a DFG resulting from the code in Figure 2a. In a DFG, *statement* nodes represent operations and functions calls, which are annotated with the execution count provided by the dynamic analysis. SESE regions are delimited by two artificial nodes: *switch* nodes that are conditional control flow jumps to multiple targets, and *merge* nodes that are the target of multiple control flow edges. *Control* edges are annotated with the control flow direction and execution counts provided by the dynamic analysis. *Data* edges are created based on static and dynamic information, and annotated with detailed information of the carried value and the dependency type, namely *Read-After-Write* (RAW), *Write-After-Write* (WAW) and *Write-After-Read* (WAR).

Program Structure Tree (PST)

The PST allows to express the hierarchy of the SESE regions. In a PST, nodes represent SESE regions and edges represent their nesting relationships, as Figure 2c illustrates. In our approach, the PST helps to traverse the regions and analyze each of them in isolation in terms of both parallelization and offloading. The results of the analysis are annotated on each PST node. This information is later used during the code generation phase.

3.3 Parallelization and Offloading Analysis

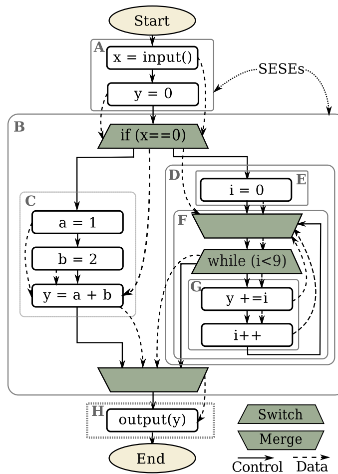
The optimization strategy follow by our approach is composed of two analyses: *i*) extraction of multiple parallel patterns from code regions, and *ii*) accelerator offloading of code regions.

3.3.1 Parallelism Extraction

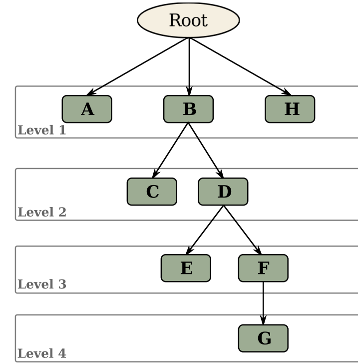
Figure 3 shows the forms of parallelism considered in this work: *Task, Data and Pipeline Level Parallelism.* One optimization candidate (function) is analyzed at the time. The goal is to extract concurrent processes by clustering C statements according to the parallelization algorithms within

```
1  void bar ()
2  {
3      int x = input();
4      int y = 0;
5
6      if (x == 0){
7          a = 1;
8          b = 2;
9          y = a + b;
10     } else {
11         int j = 0;
12         while (j < 10){
13             y += j;
14             j++;
15         }
16     }
17     output(y);
18 }
```

(a) Source Code



(b) Dependence Flow Graph



(c) Program Structure Tree

Figure 2: DFG and PST Example

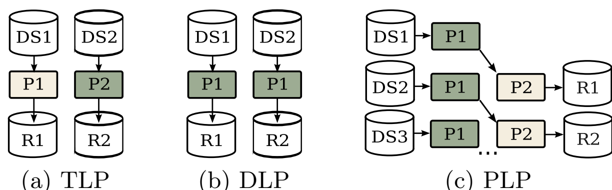
each SESE region. Using C as the smallest clustering granularity provides great flexibility. Furthermore, it enables a straightforward correlation between the extracted processes and the original source code, which enables an easy derivation of a parallel representation. In the following sections we described the algorithms used in our approach to expose multiple parallel patterns [4, 5].

Task Level Parallelism (TLP)

In TLP, a computation is divided into multiple processes that operate in parallel on the same or different data sets, as illustrated in Figure 3a. TLP typically exploits the concurrent execution of function calls or the parallel execution of several independent loops. The TLP algorithm used in this work takes each optimization candidate and then collapse all regions related to if-then-else blocks and loops as single clusters, and then leave single single statements (including function calls) as independent units. This results in a linear linear control flow. The algorithm analyzes two consecutive clusters in the control flow at a time to identify potential parallelism, until all clusters are analyzed.

Data Level Parallelism (DLP)

This is defined as a pattern where a data set is split into chunks to which the same computation is applied by multiple parallel processes, as Figure 3b shows. Exploiting DLP is the best strategy for a scalable parallelism. This pattern is typical found in multimedia applications. The algorithm used in this work for extracting DLP analyzes the most computational intensive loops within the optimization candidate functions. For every loop the following preconditions are evaluated: *i*) there are no true loop-carried dependencies, *ii*) the loop has only one induction variable and thus one single iteration space and *iii*) the ratio between the average trip count and the number of times the loop was entered during the profiling run is bigger than a fixed user-defined threshold. If these preconditions are met, the algorithm mark loops with DLP and estimates the speedup gains on the target MPSoC.



(a) TLP (b) DLP (c) PLP

Figure 3: Patterns. DS: Data Set, P: Process, R: Result

Pipeline Level Parallelism (PLP)

In PLP a given computation within a loop is broken into a sequence of processes (known as *stages*), as Figure 3c illustrates. This is an important pattern in the embedded domain, since many applications follow a streaming based processing approach [8], where there are serially dependent tasks, such as audio and video encoding and decoding. Similar to DLP, the algorithm used in this work to extract PLP starts by checking some preconditions for each loop region within the optimization candidate functions. Then, all the SESE sub-regions within the loop body are collapsed as clusters to get a linear control flow. This set of clusters are the initial candidates for pipeline stages. Afterwards, the algorithm is going to try to identify the best balanced configuration where new clusters are created based on the initial clusters. A pipeline configuration describes the final number pipeline stages and the mapping between statements in the loop body and the stage to which they are mapped to.

3.3.2 Accelerator Offloading

In this analysis the execution time of every SESE region within an optimization candidate is compared against the execution time on every type of accelerator in the target MPSoC. It is assumed that there is only one core type acting as a host. The execution time obtained when running a SESE region on a given accelerator is estimated as follows: $t_{off} = t_{acc} + t_{data}(bytes)$. Here t_{acc} represents the actual execution time on the accelerator, and $t_{data}(bytes)$ is the offloading time required to map a given amount of bytes to and from the accelerator. The potential performance improvement after offloading is estimated according to the Amdahl's law: $speedup = t_A/(t_A - t_{host} + t_{off})$. Here t_A is the execution time of the application on a single host core, t_{host} is the execution time of the region on the host core, and t_{off} is the execution time of the region on the accelerator. If the speedup is bigger that an user-defined threshold, the SESE region is marked as an offloading candidate. Finally, the outermost offloading candidates (called here *maximal*

offloading regions) are extracted from the PST. This helps us to avoid nested offloading. Furthermore, to avoid nested offloading across function boundaries, we mark functions invoked within offloaded regions and their callees chains as fully offloaded functions.

3.3.3 Source Level Hints

Our approach generates multiple source level hints based on the information collected during the parallelization and offloading analyses. This allows an easy interaction with the user and an intuitive presentation of the results. The goal of these hints is to provide the developers with a deep understanding about the applications and its optimization potential. Examples of the hints are: list and workload of the optimization candidates, details about the identified parallel patterns (DLP, PLP or TLP), expected speedups, among others.

3.3.4 Parallelization and Offloading Implementation

This section describes how the discovered parallelization and offloading opportunities are implemented.

C for Process Networks (CPN)

The first alternative considered here to implement parallelism is the use of Kahn Process Networks (KPNs), which allow to exploit all the parallel patterns described in Section 3.3.1. In order to implement KPNs we used *C for Process Networks* (CPN). CPN is an extension to C that allows to specify applications as KPNs, which is commercially provided by Silexica [2]. It was designed with the goal of making the description of KPNs intuitive, while keeping its syntax close to C. In order to derive a CPN specification, our approach generates special source level hints, which provide skeletons to help the developers in the process of deriving a CPN specification from the input application.

OpenMP

OpenMP is a parallel programming paradigm for shared-memory multicore systems based on compiler directives. It is now widely supported by multiple compilers, and it has been gaining acceptance in the embedded domain. The most common use of this paradigm is to parallelize loops with no loop-carried dependencies to exploit DLP, thus this is the focus of our work. OpenMP was extended for heterogeneous systems with the introduction of the *accelerator model* in the 4.0 specification [3]. This is a host-centric model, which allows to offload code regions and data to accelerators. The accelerator model is portable across multiple types of target devices with different instruction set architectures (ISAs), such as DSPs or GPUs. The information about parallelization and offloading opportunities identified by our approach, allow us to automatically generate OpenMP pragmas.

4. RESULTS AND CONTRIBUTIONS

In this section, we present the experimental results, and also we summarize the most important contributions of our work.

4.1 Results

In order to evaluate the effectiveness of our approach we selected two commercial embedded devices from different application domains: *i*) the Android based Nexus 7 tablet and *ii*) the KeyStone II platform from Texas Instruments. Each platform allows to evaluate different aspects of our framework.

Table 1: Parallelization of the Benchmarks with CPN

Benchmark	Processes	Channels	Patterns
Beamformer	5	2	DLP
Edge Detection	8	10	TLP, DLP
JPEG Decoder	2	1	PLP
LTE	5	2	DLP
PNG Decoder	2	1	PLP
Webp Decoder	2	1	PLP

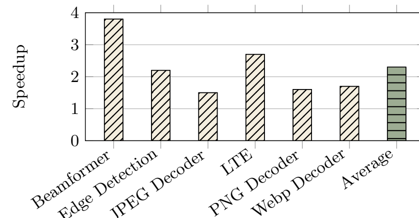


Figure 4: CPN Speedups on the Nexus 7 Tablet

4.1.1 Evaluation on Android Devices

Android became the "cornerstone" of the major vendors in the market of mobile devices. This success relies on its Java based programming model that enables to develop portable applications across multiple hardware platforms. However, this comes at the cost of performance. To overcome this limitation, new APIs were introduced in Android that allow to develop computationally intensive application sections in native code, thus enabling to exploit specific characteristics of the target MPSoCs. Therefore, we applied our framework to analyze native C code in Android applications [4].

The selected Android device for this evaluation is the Nexus 7 tablet [1], which is based on a 1.5GHz ARM Krait Quad-core Snapdragon MPSoC. The goal of this evaluation is to parallelize embedded streaming applications across the ARM cores, using CPN to implement the identified parallel patterns, namely TLP, DLP and PLP.

Table 1 shows the characteristics of the process networks implemented in CPN using the parallelization information provided by our framework. The process networks are described in terms of the number of processes, number of FIFO channels and the parallel patterns exploited. It is interesting to highlight, that decoding applications such as *JPEG*, *PNG* and *WebP* benefit from PLP. Such applications have to process a compressed input bitstream by performing a sequence of steps within a loop. The way how the bitstream is processed creates loop-carried dependencies that prevent DLP. On the other hand, the extraction of PLP is straightforward, as these steps can be represented as pipeline stages.

Figure 4 shows the speedup results using as the baseline the execution time of the sequential version of each benchmark running on a single ARM core. The best result was achieved by the *Beamformer* benchmark with a speedup of 3.8×, since its workload is dominated by a loop that can be parallelized with DLP, thus it scales well on the four ARM cores. On average, we obtained a speedup gain of 2.3×.

4.1.2 Evaluation on TI Keystone II Platform

The 66AK2H MPSoC [9] belongs to the Keystone II family from Texas Instruments. This MPSoC is composed of a cluster with four ARM Cortex-A15 cores (running up to 1.4GHz) and a cluster with eight C66x DSP cores (running up to 1.228GHz). The applications of this MPSoC include Communications, Space, Video and Image Processing.

The goal of this evaluation is to parallelize and offload loop

Table 2: Loop Parallelization and Offloading with OpenMP

Benchmark	Manual		Automatic	
	Parallelized	Offloaded	Parallelized	Offloaded
BP	2	0	2	1
BFS	2	1	2	1
Hotspot	1	1	1	1
Kmeans	1	0	1	0
LUD	2	1	2	1
NN	1	0	1	0
ParticleFilter	11	0	2	0
PathFinder	1	0	1	0

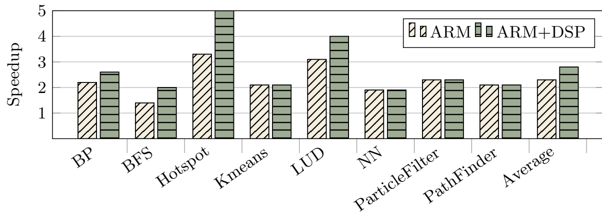


Figure 5: OpenMP Speedups on Keystone II Platform

regions on the Keystone II platform using OpenMP. For this purpose we selected the Rodinia benchmark suite, which targets heterogeneous platforms. Rodinia is a set of data parallel benchmarks manually annotated with OpenMP pragmas and other paradigms as well. This is beneficial to verify the effectiveness of our approach, since it allows to compare the automatically generated annotations by our toolflow against the manual annotations by expert programmers.

After analyzing the Rodinia benchmarks our framework discovered abundant DLP. In the case of *LUD*, PLP was discovered besides DLP. However, in this case our profitability analysis selected DLP as the most beneficial pattern. Table 2 shows the number of manually versus automatically parallelized and offloaded loops with OpenMP. From these results and by comparing the generated code versus the handwritten code, we could verify that our framework was able to parallelize and offload the same loops in almost all benchmarks, just with two exceptions. This first one was in the *BP* case in which we offloaded one loop in contrast to the manual version. As we will see in the speedup results this leads to a performance improvement. The other exception was in the *ParticleFilter* case in which we annotated only two loops, in contrast to ten loops in the manual version. The reason for this difference is that in the manual version of *ParticleFilter* several non-intensive loops were annotated.

Figure 5 shows the speedup results obtained after automatically parallelize and offload loops within the Rodinia benchmarks with our framework. The baseline for speedup computation is the sequential execution time of each benchmark on a single ARM core. We evaluated two scenarios: *i*) only parallelization on the ARM cluster, and *ii*) parallelization and offloading considering both ARM and DSP clusters. In the first scenario the performance was increased by 3.3× in the best case, and on average by 2.3×. In the second scenario, results show that the performance improvements for benchmarks with offloaded loops outperform the first scenario (*BP*, *BFS*, *Hotspot* and *LUD*). These results highlight the benefit of our joint parallelization and offloading approach. Here the performance of the benchmarks is increased in the best case by 5×, and on average by 2.8×.

4.2 Contributions

The main contributions of our work are summarized as follows:

- A parallelization framework for extraction of multiple parallel patterns from sequential applications targeting heterogeneous multicore embedded devices.
- Realization of the parallelization and offloading opportunities by using multiple parallel paradigms, such as CPN and OpenMP.
- Evaluation of the approach by optimizing multiple realistic benchmarks for two commercial platforms: an Android based Nexus 7 tablet and a heterogeneous MPSoC from Texas Instruments.
- Successful deployment of our research in state-of-the-art industrial tools (see Section 5).

5. IMPACT ON INDUSTRY

The research described in this paper has been successfully deployed in the industry through Silexica Software Solutions GmbH [2]. Silexica offers an award-winning *Software Design Automation* tool suite for effective multicore development of high performance embedded systems. The tool suite includes facilities for parallelization of legacy code (*SLX Parallelizer*), automatic software distribution to multicore platforms (*SLX Mapper*), and code generation for a variety of platforms (*SLX Generator*). Our research is in particularly at the core of the *SLX Parallelizer*. This close collaboration with Silexica represents an extremely valuable input to drive our research towards the real needs of the current embedded multicore industry.

6. REFERENCES

[1] Nexus 7 (2013). [Online] Available http://www.asus.com/Tablets_Mobile/Nexus_7_2013/ (accessed 01/2015).

[2] Silexica Software Solutions GmbH. [Online] Available http://www.silexica.com (accessed 4/2015).

[3] OpenMP Application Programming Interface. Version 4.0. http://www.openmp.org, July 2013.

[4] M. A. Aguilar, J. F. Eusse, P. Ray, R. Leupers, G. Ascheid, W. Sheng, and P. Sharma. Parallelism extraction in embedded software for Android devices. In *Proc. SAMOS XV*, July 2015.

[5] J. Castrillon and R. Leupers. *Tool Flows to Close the Software Productivity Gap*. Springer, 2014.

[6] D. A. Cordes. *Automatic Parallelization for Embedded Multi-Core Systems using High-Level Cost Models*. PhD thesis, TU Dortmund University, 2013.

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

[8] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of ASPLOS*, pages 151–162, 2006.

[9] T. Instruments. 66AK2H14/12/06 Multicore DSP+ARM Keystone II System-On-Chip (SoC). SPRS866.

[10] M. Islam. On the limitations of compilers to exploit thread-level parallelism in embedded applications. In *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*, pages 60–66, July 2007.

[11] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proc. of PLDI*, pages 78–89, NY, USA, 1993.

[12] R. E. Johnson. Software development is program transformation. In *Proc. FoSER*, pages 177–180, 2010.

[13] I. Karkowski and H. Corporaal. Overcoming the limitations of the traditional loop parallelization. *Future Gener. Comput. Syst.*, 13(4-5):407–416, Mar. 1998.

[14] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkarmark, X. Tian, and H. Saito. Challenges in exploitation of loop parallelism in embedded applications. In *Proc. CODES+ISSS*, pages 173–180, NY, USA, 2006.

[15] M. Kim. *Dynamic Program Analysis Algorithms to Assist Parallelization*. PhD thesis, Atlanta, GA, USA, 2012.

[16] G. Tournavitis. *Profile-driven Parallelization of Sequential Programs*. PhD thesis, University of Edinburgh, 2011.