A parallelization approach for resource-restricted embedded heterogeneous MPSoCs inspired by OpenMP

Olaf Neugebauer*, Michael Engel, Peter Marwedel

TU Dortmund University, Dortmund, Germany

ARTICLE INFO

Article history:
Received 13 May 2015
Revised 15 August 2016
Accepted 21 August 2016
Available online xxx

Keywords:
Parallelization
Heterogeneous multiprocessor system-on-chip
Embedded systems

ABSTRACT

Future low-end embedded systems will make an increased use of heterogeneous MPSoCs. To utilize these systems efficiently, methods and tools are required that support the extraction and implementation of parallelism typically found in embedded applications. Ideally, large amounts of existing legacy code should be reused and ported to these new systems. Existing parallelization infrastructures, however, mostly support parallelization according to the requirements of HPEC systems. For resource-restricted embedded systems, different parallelization strategies are necessary to achieve additional non-functional objectives such as the reduction of energy consumption. HPC-focused parallelization also assumes processor, memory and communication structures different from low-end embedded systems and therefore wastes optimization opportunities essential for improving the performance of resource-constrained embedded systems.

This paper describes a new approach and infrastructure inspired by the OpenMP API to support the extraction and implementation of pipeline parallelism, which is commonly found in complex embedded applications. In addition, advanced techniques to extract parallelism from legacy applications requiring only minimal code modifications are presented. Further, the resulting toolflow combines advanced parallelization, mapping and communication optimization tools leading to a more efficient approach to exploit parallelism for typical embedded applications on heterogeneous MPSoCs running distributed real-time operating systems.

© 2016 Published by Elsevier Inc.

1. Introduction

Today, powerful embedded systems used for high-performance embedded computing (HPEC) are more and more merging with traditional high-performance computing (HPC) systems in terms of requirements, programmability and bottlenecks. The Odroid-XU3 (Hardkernel Odroid-XU3, 2016) using the big.LITTLE architecture (ARM Limited, 2013) is an example of such system. This powerful embedded processor is used in mobile phones or mobile sensor applications like virus detection (Neugebauer et al., 2015b). Those systems can run full fledged multi process operating systems reducing the complexity of application development, and resource management. However, on the other side are low power and low-end embedded systems used e.g. in Internet of Things (IoT) applications or sensor networks. Those systems are typically extremely constrained in terms of computation power, memory, and energy availability, provided by batteries or solar panels. Nevertheless, increasing demand of more performance even for those limited systems leads to usage of multiprocessor system-on-chip (MPSoC) designs. These systems are composed of different processors running at different frequencies, DSPs, and special memories like scratch pad memory. Heterogeneity in these not only stems from the hardware side, but also from the software side. Especially for multiprocessor systems it is possible that each processor has its own operating system, if any. Communication between the cores and these distributed operating systems can be coupled tightly by sharing the same memory or loosely by using a communication infrastructure, e.g., network on a chip.

Exploiting the capabilities of low-end embedded MPSoC architectures efficiently requires software tools considering the system capabilities. These tools must avoid creating bottlenecks, which may result from ignoring the heterogeneity as well as the resource limitations of embedded processors, memory systems, and communication channels, while at the same time enabling developers to reuse large libraries of existing C code. Unfortunately, legacy code was mostly developed for single processors, so an efficient use of MPSoCs requires a tool that is able to create a parallel version of a sequential program with as little effort as possible.

* Corresponding author.
E-mail addresses: olaf.neugebauer@tu-dortmund.de (O. Neugebauer),
michael.engel@tu-dortmund.de (M. Engel), peter.marwedel@tu-dortmund.de (P. Marwedel).

http://dx.doi.org/10.1016/j.jss.2016.08.069
0164-1212/© 2016 Published by Elsevier Inc.
Existing parallelization support, however, commonly targets HPC or HPEC systems, which tend to be more homogeneous and at the same time much less resource-constrained compared to typical low-end embedded systems. As a consequence, using HPC/HPEC tools to create parallel software for constrained embedded systems may lead to wasted optimization opportunities and overallocation of resources.

A fundamental drawback of existing HPC approaches is that the underlying assumptions of given software structures do not fit to typical embedded applications, since embedded software exhibits different control flow structures and data dependencies compared to high-performance computing. Whereas many HPC applications can be executed efficiently using coarse-grained task-level and fine-grained data-level parallelism, many embedded applications exhibit more complex dependencies. This can lead to a substantial efficiency loss when applying HPC parallelization tools to embedded MPSoCs.

Embedded software often operates on streams of data, e.g., in media or signal processing applications, which enables the use of pipelined parallel processing. Thus, existing approaches using coarse-grained task-level and fine-grained data-level parallelism should be augmented to exploit the additional optimization opportunities that pipeline parallelism offers. Ideally, the parallelization system should simultaneously consider the variability of embedded hardware and software structures during the implementation of the parallel software.

In this paper, we present PICO (Parallelism Implementer and Communication Optimizer), our approach to support parallel program creation and execution for constrained embedded MPSoC platforms running multiple real-time operating systems e.g., the Real-Time Executive for Multiprocessor Systems (RTEMS) (RTEMS Operating System, 2016). Since establishing such a system requires a large-scale effort, we did not start completely from scratch. Rather, PICO is inspired by the well-known (OpenMP, 2016) syntax and semantics. This has a number of advantages, e.g., Using OpenMP concepts facilitates the accessibility of our framework, since the basic parallelization model, API style, and annotation syntax are familiar.

However, our implementation is not derived from existing OpenMP variants. The major reason for this is that PICO tries to apply an established design principle for constrained embedded systems, namely to accomplish as many decisions as possible at compile time, thereby reducing the required runtime overhead. Thus, decisions for scheduling and resource allocation are performed statically in PICO. Further, PICO does not require a shared memory architecture. Due to automatic dependency extraction, PICO can implement channel-based synchronization automatically. In addition, a modification of OpenMP to a system with potentially multiple real-time operating systems with its limited hardware resources is a complex and always a very promising task.

In a nutshell, PICO’s main contributions are:

- An OpenMP-inspired infrastructure to support pipeline parallelism efficiently for heterogeneous low-end embedded MPSoC systems with multiple real-time operating systems.
- Easy to apply source code annotations to describe coarse pipeline parallelism in-line.
- Automatic extraction of data dependencies and implementation of necessary synchronization.
- Interfacing state-of-the-art tools for extracting parallelism from sequential code, communication optimization and mapping to MPSoCs.

This paper gives an overview of the structure, syntax and semantics of PICO and evaluates its optimization capabilities by analyzing typical embedded benchmarks parallelized for different MPSoCs. 

**Section 2** discusses related work. **Section 3** describes the specifics of parallelization for embedded systems. Syntax and semantics details are given in **Section 4** followed by a description of PICO’s integration into a comprehensive parallelization framework in **Section 5**. **Section 6** evaluates PICO’s parallelization capabilities. **Section 7** proposes directions for future work while **Section 8** presents a conclusion of the paper.

**2. Related work**

So far, a number of tools have been developed that implement parallelism annotated in sequential application code. Below, we give examples of systems used in the embedded domain.

The MPSoC Parallelization Assist (MPA) by Baert et al. (2009) helps application designers to implement manually extracted coarse-grained parallelism from sequential C code. Thies et al. (2007) presented a program transformation technique using stream graphs which implements pipeline parallelism for annotated C code. Profiling information is used to detect communication between pipeline stages.

OpenMP (2016) allows developers to explicitly specify code regions which should be executed in parallel. Thus, a basic parallelized version is achievable with only small changes to the original sequential version, significantly reducing the required effort. In addition, the sequential code structure can be preserved which simplifies understanding of the application’s behavior. Several mechanisms for work sharing, scheduling or synchronization are available. The OpenMP standard enables the developer to distribute work to heterogeneous systems as well. Data dependencies have to be specified manually, which can be very complex and error-prone.

Even though OpenMP works well in the domain of HPC, its applicability to embedded devices is limited. Therefore, many approaches have been presented which extend OpenMP in various ways. An extension by Marongiu and Benini (2009) is able to utilize explicitly managed memory hierarchies which are often found in embedded devices. Directives were added to specify arrays to be mapped to local scratchpad memories. In addition, a compiler-based semi-automatic array partitioning (Marongiu and Benini, 2012) assists the developer to distribute the memory access to different memory locations is described. Burgio et al. (2013) developed a runtime layer for OpenMP tasks targeting embedded shared memory clusters. A work-queue is used to distribute the work. For such clusters, Marongiu et al. (2015) developed a programming model based on OpenMP. Their model is intentionally simple to increase the usability. For the target many-core systems, organized in a multicluster design, their presented offload directives achieved comparable speedups to hand-optimized OpenCL code. Chapman et al. (2009) support different memory architectures. They introduce a target clause for OpenMP tasks to execute a task on a specific target processing unit. A compiler for MPSoCs was presented by Liu and Chaudhary (2003) that extends OpenMP to enable work-sharing across multiple Digital Signal Processors. Here, several extensions were implemented to utilize platform-dependent features. Implementing OpenMP programs on MPSoCs without an operating system has been investigated by Jeon and Ha (2007). Wang and Chandrasekaran (2013) created a OpenMP runtime library leveraging the MultiCore Association APIs focusing the resource management. This runtime is also applicable for embedded systems with different operating systems. OpenStream by Pop and Cohen (2013) uses semantics and the semantics of the OpenMP to implement stream programming by applying a data-flow execution model.

While many parallelization approaches as well as programming support for parallel applications exist, support for pipeline parallelism and heterogeneous embedded systems with automatic data dependency extraction is still rare. In this paper we focus and present solutions for the parallelization of applications for embed-
ded multicore systems running distributed real-time operating systems with complex structured and not necessarily shared memories.

3. Parallelization for embedded systems

Using OpenMP’s syntax and semantics as the basis for a parallelization tool for embedded systems is attractive due to its familiarity and its easy and seamless integration into C code. However, in order to create efficient embedded parallel software especially targeting low-end systems, a number of different approaches and extensions to the feature set of OpenMP were required.

A conventional OpenMP runtime environment is often not available for typical embedded systems, for example, due to restricted APIs, employed process models or distributed operating systems. While porting an OpenMP implementation to a POSIX-compliant OS is possible, this obvious approach has a number of drawbacks. Common OpenMP implementations rely on runtime decisions to decide task schedules and additional resource allocations during the execution of a program. For low-end embedded systems, however, the amount of resources as well as the set of tasks to be executed is mostly known at design time. Accordingly, most resource allocation and scheduling decisions can already be taken at compile time, which allows to reduce the required overhead significantly at runtime. While the resources of an embedded MPSoC are known beforehand, instances of a resource type can nevertheless differ significantly in their parameters. For example, a modern MPSoC, like ARM’s big.LITTLE platform (ARM Limited, 2013), may contain processors running at different clock frequencies and featuring differing pipeline structures, cache sizes, bus systems, and memory hierarchies. For efficient parallel programs, execution times of tasks on different processors should be balanced well; thus, a static scheduling approach has to integrate knowledge about the resource characteristics of the different MPSoC system components. PICO provides the automatic insertion of communication and synchronization primitives as required in the code of the parallel tasks. In order to adapt them to a large number of possible on- and off-chip communication infrastructures of different MPSoCs, PICO not only supports communication over shared memories, but also using different communication infrastructures such as hardware FIFOs or networks on a chip.

Finally, one of the most relevant differences between HPC and embedded software is the more complex form of parallelism inherent in embedded software. While task-level and simple data-level parallelism can be easily analyzed and annotated by an experienced programmer, pipeline parallelism and its dependencies are much less obvious and, consequently, much harder to detect and annotate. Thus, PICO provides a data flow analysis which enables the automatic detection of data dependencies. In the remainder of this section, we demonstrate a typical embedded systems parallelization use case which exhibits complex data dependencies and the usefulness of pipeline parallelism for heterogeneous MPSoCs.

Fig. 1(a) shows the main computation loop of the Spectral benchmark from the UTDSP benchmark suite (Lee, 2016), which is a representative embedded application that calculates a power spectrum of an input speech sample. The outer loop contains two inner loops and a call to a fft (Fast Fourier Transform) function in between both loops. The second inner loop contains a loop-carried dependency to its previous iteration, indicated by a red dashed line, because it reads from mag[j] which was written in the previous iteration of the outer loop. Due to this dependency, task-level parallelism, where statements are bundled into independent tasks executed in parallel, is not applicable in this case. In addition, due to the loop-carried dependency, it is not possible to execute all loop iterations in parallel and benefit from data-level parallelism.

Pipeline parallelism is a possible solution to deal with loop-carried dependencies without rewriting the algorithm. Here, the outer loop is split into two pipeline stages and a task is created for each stage (T1 and T2). Data dependencies between T1 and T2 inhibit independent execution of both tasks resulting in the timing diagram depicted in Fig. 1(b). Interleaved execution can reduce the tasks’ idle time. Additional parallelism can be exploited by creating additional instances of the first pipeline stage (cf. Fig. 2(a)). Due to the fact that three results of the first pipeline stage are available at the same time, Task 2 can process all iterations in a row (cf. Fig. 2(b)). Idle time is reduced and performance is increased by balancing calculations onto available processing units.

Transferring a static iteration scheduling and processor mapping for a homogeneous system to a heterogeneous system usually leads to suboptimal results if runtime scheduling is not considered. Fig. 2(c) shows this unbalanced behavior for a heterogeneous platform with different processing speeds. Here, tasks 1.2, 1.3 and 2 are mapped to faster processors compared to task 1.1 which leads to long idle times. Therefore, support for offline loop iteration distribution across tasks with respect to the target platform is required. OpenMP’s runtime library provides runtime scheduling for loop iterations. However, the additional overhead introduced at runtime is usually not acceptable in the context of resource-restricted or timing-critical embedded systems. A possible balanced timing behavior is shown in Fig. 2(d).

4. PICO API

This section describes PICO’s OpenMP-style C extensions to enable its use in an low-end embedded system context. Like OpenMP, we use #pragma directives as annotations to define parallel regions. Resulting parallel programs use the fork-join execution model, where a running task spawns new subtasks (fork) and waits until all subtasks have terminated (join), as depicted in Fig. 3. Here, it can be seen that implicit control-flow and necessary data synchronization are added automatically at the end of each parallel region. Data exchange between concurrently running tasks us-

---

Please cite this article as: O. Neugebauer et al., A parallelization approach for resource-restricted embedded heterogeneous MPSoCs inspired by OpenMP, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.08.069

3.
Fig. 3. Fork-join execution model.

Fig. 2. Spectral benchmark – pipeline parallelism.

Listing 1. Parallel sections.

```c
#pragma pico parallel sections num_threads(2) {
    #pragma pico section taskid=1 processor={1,3} 
    func1();
    #pragma pico section taskid=2 processor={2}
    func2(b);
    #pragma pico section taskid=1 processor={1,3}
    c = func3(a);
}
```

Fig. 4. Timing behavior: parallel sections.

4.1. Parallel sections

Task-level parallelism is expressed by parallel sections (pico parallel sections) where tasks are annotated with pico section pragmas. To group statements to tasks, a clause named taskid is used. This enables the developer to assign statements to tasks without reordering them. This simplifies the parallelization step, eliminates possible pitfalls and thus quickly leads to parallel prototypes. In addition, the processor clause enables static task mapping to a set of processing units of the target platform. A mapping tool can then decide which specific processing unit should be used.

An example with three statements executed in two tasks is given in Listing 1. Here, the functions func1 and func3 are mapped to task 1 and are executed in parallel to task 2 processing func2. Task 1 will be executed either on processor 1 or 3 and the mapping tool later decides which processor should be used. Task 2 is allocated to processor 2 by the designer. The timing behavior is depicted in Fig. 4.

4.2. Parallel loops

Pipeline parallelism is often the most efficient way to parallelize embedded software. The pico parallel pipeline for pragma is used to annotate loops which should be processed in a pipeline manner. Each pipeline stage is grouped by pico section directives similar to statements to tasks grouping (cf. Section 4.1). To duplicate a pipeline stage and execute its iterations concurrently, chunks or iterations clauses are used.

A chunk specifies a sequence of consecutive iterations executed on a single processor, sometimes also called workload or work size. The related chunk clause partitions the iteration space of a pipeline stage into continuous blocks of subsequent iterations. Each chunk definition implicitly creates one task for each chunk. The accumulated size of all blocks in one definition must in general be equal to the number of iterations of the loop. For example, chunks = 3,1,1,1 (6 iterations, without interleaved) creates four tasks, where the first task processes the first three iterations, the second one and so on. This ensures that the parallelized loop processes the same iteration space as the sequential version. By using the interleaved clause, the user can specify a more


```c
#pragma pico parallel for num_threads(4)
for(int i = 0; i<12; i++)
#pragma pico section taskid=1 chunks=3,2,1
    interleaved processor=(2,4),(2,4),(1,3)
a = func1();
#pragma pico section taskid=2 processor=(1,3)
b = func2(a);
```

**Listing 2.** Parallel pipeline for interleaved chunks.

Fig. 5. Timing behavior: parallel pipeline for interleaved chunks (cf. Listing 2).

```c
#pragma pico parallel for processor=(2),{1}
    chunks=10,2
for(int i = 0; i<12; i++)
    a = func1();
    b = func2(a);
```

**Listing 3.** Parallel for interleaved chunks.

general mapping. Here, each task processes chunks of the defined size until all iterations have been processed. An example is shown in Listing 2. The first instance of task 1 executes chunks with a size of 3, the second instance chunks with a size of 2 iterations and the third instance chunks of size 1. These assignments are repeated until the complete iteration space has been mapped to the available tasks. Fig. 5 shows the timing behavior of this example. Iterations are mapped to tasks at compile time and do not require additional scheduling overhead at runtime.

Using the chunks clause, iteration spaces can be tailored for heterogeneous architectures very easily. However, in some cases it is beneficial to map specific iterations directly to tasks or processors to create an even more fine-grained mapping. This can be achieved by the iterations clause.

The processor clause is used to distribute tasks to processing units. In contrast to parallel sections, the clause for pipeline stages accepts multiple groups of processing elements. The size of the group correlates to the number of created tasks defined by chunks or iteration mappings. For Listing 2, the first two instances of the first stage will be mapped to processors two and four where the last instance is mapped to processor one or three which will be decided later on by the mapping process. Parallel loops with only one pipeline stage realize data-level or loop-level parallelism. This can be shortened with the pico parallel for pragma as depicted in Listing 3. The num_thread clause can be omitted if the number of tasks can be derived from the chunks or iteration mapping. If no iteration mapping is specified, the num_thread clause is used and the iteration space is split into equally-sized parts. The timing behavior is shown in Fig. 7. This mapping might be beneficial if processor 1, to which task 2 is mapped, is much slower than processor 2 executing task 1.

Since PICO uses a pessimistic static dataflow analysis it tends to add more communication channels as really necessary. Thus, PICO provides a pragma to relax this behavior by marking a variable as shared.

To summarize, our framework enables application developers to benefit from pipeline parallelization without requiring extensive modifications of the existing sequential applications. Furthermore, offline load-balancing techniques are provided to take heterogeneous embedded systems into account. Details can be found in Section 5.1 With free statement to task mapping, the original source code structure is preserved. Finally, our processor class mapping enables tailored processor-dependent implementations to increase the benefit obtained from heterogeneous platforms.

5. Framework

We implemented the proposed features into a tool called PICO, embedded into the PA4RES (Parallelization For Resource RestrictedEmbedded Systems) framework which enables a tight interaction with several other tools as shown in Fig. 6.

PICO transforms annotated sequential source code into a statically scheduled parallel program with necessary communication and synchronization primitives between tasks. The tool is based on the MACC framework (Pyka et al., 2010) using the ICD-C (ICD Compiler, 2016) compiler framework’s abstract syntax tree and versatile program and data flow analyses. These techniques enable an automated extraction of dependencies for synchronization.

Annotated source code can either be provided by the user or generated with the automatic parallelizer (Cordes et al., 2012; 2011) called PAXES (Parallelism eXtraction for Embedded Systems). PAXES provides ILP- or Genetic Algorithm-based approaches to extract the nested parallelism, resulting in a set of Pareto-optimal solutions based on a static model. In this set are solutions resulting in high speedups but also high energy consumption or solutions with the opposite behavior. The user of the framework then selects a solution which fits best in the current scenario. The parallelized application generated by PICO is then passed to a mapping tool which finally maps the application onto the target platform.

To summarize, the user of the framework can either pass a sequential program without annotations or an annotated application to the PA4RES framework. In the following, a detailed description of PICO’s internal workflow is given.

5.1. Internal workflow of PICO

In a first step, the source code of the sequential annotated application is transformed into an inter-procedural program dependency graph (PDG) containing control and data dependencies. Each C statement is represented by a node. Control flow and data dependencies are represented by edges between the nodes. A static pointer-aware data analysis is applied to discover data dependencies nested in the source code. Typical dependency analysis for C code tends to over-approximate in order to guarantee safe results. The user of the PICO can improve the results of the analysis by following coding guidelines like, e.g., the rules proposed by Parallelizable C (Mase et al., 2010). In a third step, the source code annotations (pragmas) are processed. Based on all collected information, a hierarchical task graph (HTG) is constructed. This hierarchical graph reflects the fork-join model also used by OpenMP. According to the annotations, fork and join nodes are inserted into the graph. Then the graph between the fork and join nodes is duplicated for data-level parallelism, for task-level parallelism the nodes are mapped to the tasks, or mapped and duplicated for pipeline parallelism. Control flow and data-dependencies are adjusted as required. Communication and synchronization nodes required between tasks are determined and added by the framework automatically. Therefore, knowledge about task partitioning and data dependencies are exploited. In contrast to OpenMP, the developer does not have to define private and shared variables explicitly, which is often a time-consuming and error-prone task. Implementing communication is efficiently a complex task. For example the software FIFO implementations used in PICO can be mapped to several memories. In addition, determining the size of the FIFO, other implementation details like busy waiting or interrupted-based approaches, and the position in the source code for the com-
munication is extremely complex. Thus, a Genetic Algorithm-based communication optimizing (Neugebauer et al., 2015a) phase can be applied to map communication to the target platform automatically. The algorithm creates a set of Pareto-optimal solutions in terms of energy consumption, runtime and memory consumption. Manual load balancing, in terms of iteration mapping, can be very complex, thus PICO offers two heuristics to map iterations to parallel tasks. One round-robin approach divides the iteration space by the number of tasks and then assigns the workload accordingly. This approach typically achieves good results for homogeneous systems and loops with identical behavior across several iterations. For heterogeneous systems, a heuristic can be used which takes the capabilities of the target processors into account and thus assigns different workload sizes to the cores. Performance characteristics of the target processors can be accessed through the MACC framework. In this paper, the performance characteristic can be described by the frequency of the processor. For more complex systems, MACC estimates the performance of each processor. To distribute the workload of a parallel loop, PICO then assigns more iterations to the processors with the higher performance. This process is transparent to the user and requires no additional annotations, only PICO’s configuration needs to be adopted.

Finally, the parallelized program is combined with a lightweight runtime library managing tasks and providing communication interfaces.

6. Evaluation

To investigate the capabilities of PICO in combination with the PA4RES framework, we selected applications from the UTDSP benchmark suite (Lee, 2016) and SNU real-time benchmark suite (SNU Real-time Benchmark Suite, 2016) containing representative real-world embedded applications. Table 1 summarizes all used benchmarks. Further, we evaluated the framework with other applications commonly used in the embedded systems domain, such as a JPEG encoder. For simplicity, PAXES is used to identify parallel region in a sequential application. It generates a set of Pareto-optimal parallelization decisions, basically annotated source code. For each benchmark, one solution was selected randomly. Thus, potential speedup of the benchmarks could be higher but then, the energy consumption would be higher as well. This trade-offs are very important in the design of low-end embedded systems and software. At the end, the user decides which solution fits best to the scenario.

In the first experiment, we present results focusing on a homogeneous multicore system with an ARM11MPCore composed of four ARM cores running at 500 MHz comparable to Fig. 8. On each core, a distributed RTEMS (RTEMS Operating System, 2016) instance is running. $R^2G$ (Heinig, 2010) is used as a lightweight library-based middleware. This library simplifies, among others, task creation and memory allocation for the target system with its multiple operating systems. As usual in low-end embedded systems, task mapping and memory allocation is done statically.

In the second part we discuss how heterogeneous systems can be targeted by our framework in detail. The heterogeneous platform consists of four ARM processors running at different frequencies comparable to Fig. 8. As in the homogeneous case, each core executes its own operating system instance.

Both platforms use the same memory architecture with four different memories. A 8 MB memory is used for the boot loader of the operating systems, a 1MB scratch pad memory (SPM) is partially used by the operating systems and could be used by the programmer, a partitioned 64MB DRAM is used to store processor...

![Fig. 7. Timing behavior: parallel (cf. Listing 3).](image)

![Fig. 8. Basic platform structure.](image)
Table 2

Results for PICO-based parallelization executed on a homogeneous system with four ARM cores, simulated with Virtualizer. Identification of parallelizable loops done by PAXES. Sequential execution assumes a single core system.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Com channels</th>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adpcm</td>
<td>0</td>
<td>2.55</td>
<td>76.67</td>
</tr>
<tr>
<td>Boundary-value-problem</td>
<td>0</td>
<td>45.49</td>
<td>1242.66</td>
</tr>
<tr>
<td>Compress</td>
<td>0</td>
<td>120.17</td>
<td>4223.08</td>
</tr>
<tr>
<td>Edge detect</td>
<td>0</td>
<td>147.31</td>
<td>7582.11</td>
</tr>
<tr>
<td>Filterbank</td>
<td>3</td>
<td>119.11</td>
<td>4595.60</td>
</tr>
<tr>
<td>FIR-filter</td>
<td>0</td>
<td>8.35</td>
<td>344.30</td>
</tr>
<tr>
<td>Lattice-filter</td>
<td>0</td>
<td>3.07</td>
<td>121.12</td>
</tr>
<tr>
<td>Matrix Multiplication 80 × 80</td>
<td>0</td>
<td>225.69</td>
<td>9709.25</td>
</tr>
<tr>
<td>Spectral analysis</td>
<td>6</td>
<td>6.90</td>
<td>214.52</td>
</tr>
<tr>
<td>JPEG</td>
<td>3</td>
<td>76.75</td>
<td>2696.46</td>
</tr>
</tbody>
</table>

Table 2 Results for PICO-based parallelization executed on a homogeneous system with four ARM cores, simulated with Virtualizer. Identification of parallelizable loops done by PAXES. Sequential execution assumes a single core system.

6.1. Homogeneous platform experiments

For the homogeneous platform, PAXES chooses data-level parallelism for the benchmarks without necessary communication (cf. Table 2) and pipeline parallelism for the others. Since PICO extracts data dependencies automatically, just simple pragmas are needed to parallelize the applications. Scheduling decisions are not added, thus the framework decides how to distribute the workload. For this experiment, PICO’s communication optimization phase was disabled. Applying this phase can further improve the performance as reported in Neugebauer et al. (2015a). For energy consumption, the sequential version of the benchmark is always executed on a single core and only energy consumption of one core is measured. This reflects the design exploration process of embedded systems where different combinations of software and hardware are investigated. This can also be seen as a single core system or a multicore system where cores can be disabled. The Matrix Multiplication benchmark is a good example that PICO is able to achieve a super-linear speedup. The other benchmarks are not always parallelizable well, e.g. in Adpcm only one small loop could be found to be parallelized leading to a speedup of just 1.11.

To give a more detailed view, Listing 4 shows an outline of the annotated Filterbank benchmark. In this benchmark a set of filters is applied to an input vector. For simplicity most of the filter implementations are not shown in the listing. As can be seen, pipeline parallelism is applied. Due to extracted data dependencies, PICO will add communication of Vect_F between the first stage and the second stage. In addition, the first pipeline stage is duplicated so that 3 instances share the workload resulting in four parallel tasks.

Listing 4. Annotated filterbank application.

#pragma pico parallel pipeline for num_threads(4)
for (i = 0; i < 8; i++) {
    float Vect_B[256]; //output of the H
    float Vect_D[32]; //output of the down sampler
    float Vect_Up[256]; //output of the up sampler
    float Vect_F[256];
    #pragma pico section chunks=1,1,1 interleaved
    {
        //convolving H
        //Down Sampling
        ... //Up Sampling
        ...
        //convolving F
        for (j = 0; j < 256; j++) {
            Vect_F[j] = 0;
            for (k = 0; ((k < 32) && ((j - k) >= 0)); k++){
                Vect_F[j] += F[i][k] * Vect_Up[j - k];
            }
        }
        //adding the results to the y Matrix
        #pragma pico section
        for (j = 0; j < 256; j++) {
            y[j] += Vect_F[j];
        }
    }
}

6.1.1. Comparison to OpenMP

The development of the PA4RES framework and PICO started when OpenMP 3.0 was available. Today, OpenMP 4.5 also supports dependent tasks and thus enables pipelines inside of loops to deal with loop-carried dependencies. Nevertheless, since an OpenMP implementation for our target platforms is not available we would like to investigate which results OpenMP could achieve on a high performance embedded system with the used benchmarks. We selected the Odroid XU-3 system. An Exynos 5422 processor with four ARM Cortex-A15 and four Cortex-A7 processors is the main component of the Odroid-XU3. In addition, a shared 2GB low power DDR3 memory is available. Four INA231 (Texas Instruments Incorporated, 2013) sensors are used to measure current and power separately for the “big” cores, the “little” cores, the GPU, and the memory. The sensors are connected via an I2C interface to the MPSoC.

We used the same parallelization strategies as in the previous experiment but added necessary private and firstprivate clauses. Compiler optimizations were disabled and the number of threads was set to four. Each benchmark was executed 1000 times and run time and energy consumption are averaged. Two setups were analyzed, one without any scheduling restrictions (cf. Table 3) and one were the tasks are mapped to the Cortex-A7 (cf. Table 4). With-

Please cite this article as: O. Neugebauer et al., A parallelization approach for resource-restricted embedded heterogeneous MPSoCs inspired by OpenMP, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.08.069.
out restrictions, the operating system decides where to execute the threads. This can be beneficial if execution is delayed, e.g. due to dependencies, then this task could be mapped to the energy efficient cores. Of the other side, limiting the threads to the Cortex-A7 should result in good energy efficient but maybe longer execution times.

As can be seen, the observed speedups are lower compared to the speedups in Table 2. We believe this stems from the large overhead introduced by OpenMP for these small benchmarks. Thus, we increased the dimensions in the Matrix Multiplication to 800x800 elements. Here, a comparable speedup with OpenMP can be observed similar behavior is expected for the other benchmarks. In terms of energy consumption, it is beneficial to restricted the execution to the energy efficient Cortex-A7 leading to longer run times.

Pipeline parallelism with several duplicated pipeline stages are not directly supported by OpenMP. This explains the basically not exiting speedups for the pipeline benchmarks using OpenMP task dependencies. This could be achieved by rewriting of essential parts of the application.

Considering the amount of clauses and knowledge about dependencies which are needed to ensure a semantically equivalent execution in OpenMP, PICO shows its potential for rapid prototyping and parallelization of sequential applications, e.g., in Listing 5 a comparison for the Matrix Multiplication benchmark is depicted. This emphasizes PICO's potential not only as an intermediate stage in the PA4RES framework but also as for rapid prototyping when porting sequential applications to low-end embedded multiprocessor systems.

6.2. Heterogeneous platform experiments

Targeting the special requirements of modern heterogeneous embedded systems, in this section we demonstrate PICO's capabilities to support programmers for adapting their sequential applications to such systems.

In the following, we show with three benchmarks how to meet the requirements of heterogeneous embedded systems in detail. The Spectral benchmark from Section 3 and the JPEG benchmark are discussed in detail. In addition, the Filterbank benchmark presented in the previous section is used. These benchmarks are selected since they use pipeline parallelism with communication. For the presented benchmarks, the parallelization decisions are a result of the automatic parallelization process by the PAXES parallelizer. Both, PICO’s iteration mapping heuristic described in Section 5 as well as PAXES precise iteration mapping are analyzed.

6.2.1. Spectral benchmark

Listing 6 shows the main function of the application annotated with PICO pragmas. The statements of the first loop are grouped into two disjoint pipeline stages. Data dependencies exist between both stages. The PICO framework automatically extracts the dependencies and adds necessary communication between tasks. The first pipeline stage is divided into three concurrent tasks, mapped to processor 1 or 2, and 3 and 4. The second pipeline stage is mapped the one of the 500 MHz processors (2 or 3). As can be
seen, only a small number of modifications to the original code were required to implement a pipeline structure. The second parallel region uses the parallel for construct to distribute the loop over three tasks, thus can be power gated. This emphasizes another difference between parallelization for embedded systems compared to HPC. For embedded systems, it might be beneficial not to extract as much parallelism as possible in order to achieve additional, non-functional objectives such as the reduction of energy consumption as shown above. Only four pragmas are necessary to describe such a complex parallelization.

The iteration scheduling generated by the heuristic is identical to the precise mapping generated by PAXES. The mapping clauses (chunks) were added to Listing 6 to show the final mapping. In the file passed to PICO the clauses were not included.

6.2.2. JPEG benchmark

The JPEG benchmark encodes images into jpeg files. The main computation loop of the parallelized source code is depicted in Listing 7. These modifications were generated by the PAXES parallelizer. The main loop is sliced into a pipeline with two stages. The first pipeline stage is duplicated into 3 parallel tasks. Here, a very fine grained iteration to task mapping and scheduling is generated by PAXES for the heterogeneous platform. The second pipeline stage is not split into parallel tasks, because only one processor is left and we consider systems without runtime scheduling.

6.2.3. Filterbank benchmark

To take the structure of the target platform into account, the Filterbank benchmark was slightly modified. The basic pipeline structure (cf. Listing 4) is the same but now the first stage is executed on two fast cores and the second stage on the other cores. This introduces additional data dependencies leading to 8 communication channels.

6.2.4. Evaluation

To examine the behavior of a suboptimal parallelization for a heterogeneous platform, we executed the homogeneous version of the benchmarks on the heterogeneous platform (see Table 5, hom2het cases). Here, it is obvious that suboptimal load balancing results in reduced performance. The automatic iteration scheduling heuristic provided by PICO and the precise iteration mapping of PAXES were used. Overall, the runtime could be reduced drastically and, as expected, the energy consumption could be optimized. For example, the speedup of the Filterbank benchmark could be increased from 0.68 to 2.37 even with increased communication channels. In addition, for the conducted experiments, the heuristic achieves almost the same speedups as the precise iteration mapping provided by PAXES.

The slowdowns of Spectral hom2het and Filterbank are similar but the energy consumption differs drastically. Deeper analysis shows, that the differences stems from different memory access behavior. In case if Spectral large data is communicated through the memory whereas Filterbank is dominated by calculation and thus profits from parallelization.

7. Future work

Using software-based simulation has a lot of advantages, but to further analyze the framework’s capabilities, experiments with real heterogeneous embedded are necessary.

The PAXES parallelization process currently works on a static model. Knowledge gathered by PICO, like precise communication cost, could be propagated back to improve the parallelization.

In addition to multi-media applications, we are going to parallelize new applications from the medical image processing domain by using PA4RES. The virus detection framework (Neugebauer et al., 2015b) developed in the SFB 876-B2 project (SFB, 2016) is a promising application.

We plan to investigate the benefits of extensions for PICO to deal with requirements of reliable systems in the context of the software-based fault tolerance for embedded real-time systems (Schmoll et al., 2013) project. For example, we plan to add pragmas to identify safety critical regions. Then, PICO could implement safety checks or redundant execution to increase the reliability of the system.

8. Conclusion

Creating efficient parallel software for modern resource-restricted MPSoCs is a complex and error-prone task. This paper presents an approach to parallelize legacy applications to benefit from modern heterogeneous low-end embedded systems. The proposed framework enables developers to express task-level, data-
level and pipeline parallelism without rewriting the entire application. Loops can be balanced offline according to the performance characteristics of the target processors. In addition, data dependencies are detected automatically by our framework which relieves the developer from the burden of manually specifying dependencies and synchronization points. We demonstrated the strength of our approach using typical embedded applications and conducted experiments on a homogeneous and a heterogeneous platform showing that load balancing has a crucial impact on the performance.

Acknowledgments

The authors would like to thank DFG for supporting part of this work within the Collaborative Research Center SFB 876 “Providing Information by Resource-Constrained Data Analysis” (SFB, 2016), project B2, and Synopsys for the provision of the virtual prototyping IDE Virtualizer (Synopsys, 2014).

References

Texas Instruments Incorporated, 2013. High- or low-side measurement, bidirectional current/power monitor with 1.8-V P interface.