Work-in-Progress: Multi-Grained Performance Estimation for MPSoC Compilers

Miguel Angel Aguilar[1], Abhishek Aggarwal[1], Awaid Shaheen[1], Rainer Leupers[1], Gerd Ascheid[1], Jeronimo Castrillon[2] and Liam Fitzpatrick[3]

{aguilar, aggarwal, shaheen, leupers, ascheid}@ice.rwth-aachen.de, RWTH Aachen University, Germany[1]
jeronimo.castrillon@tu-dresden.de, TU Dresden, Germany[2]
fitzpatrick@silexica.com, Silexica GmbH, Germany[3]

ABSTRACT

Parallelizing compilers are a promising solution to tackle key challenges of MPSoC programming. One fundamental aspect for a profitable parallelization is to estimate the performance of the applications on the target platforms. There is a wide range of state-of-the-art performance estimation techniques, such as, simulation-based, measurement-based, among others. They provide performance estimates typically only at function or basic block granularity. However, MPSoC compilers require performance information at other granularities, such as statement, loop or even arbitrary code blocks. In this paper, we propose a framework to adapt performance information sources to any granularity required by an MPSoC compiler.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; • **Software and its engineering → Compilers**;

1 INTRODUCTION

The selection of a granularity is a major issue in parallelizing compilers, as it has a direct impact on the form and degree of parallelism that can be exploited. MPSoC compilers work at a wide variety of granularities: statement, basic block, loops, function and arbitrary code blocks, as shown in Figure 1. To achieve a profitable parallelization, compilers require performance information at all these granularities. This enables the identification of hotspots and the evaluation of potential performance improvements provided by the parallel patterns discovered in the application [1, 3].

There is a multitude of performance estimation techniques [3], such as, simulation-based, emulation-based, measurement-based, among others. However, these techniques typically provide the estimates only at the function and basic block granularities. For example, gprof [5] provides information at the function granularity only, while the performance estimation framework for embedded platforms proposed in [4], provides information at both function and basic block granularity. This is not sufficient for the requirements of modern compilers for embedded multicore systems. Therefore, in this paper we propose a framework to adapt the source performance information to any required granularity.

2 PROPOSED APPROACH

Figure 2 shows the proposed tool-flow. The first step is to take the performance information provided by any performance estimation technique and adapt it to a generic format. The main challenge here is to deal with multiple input performance information formats. In order to handle this variability and to keep the estimate generator generic, we extract from each input performance estimator only specific information and convert it into a generic interface. This task is performed by the pre-processing step of the tool-flow. The information extracted during the pre-processing step is the following: i) basic block self costs, and/or ii) function self costs. This cost information can be given either in terms of cycles or time. In addition, meta data is extracted to be able to correlate the performance information with the source code (e.g., source file names and line numbers), as well as the core type to which this information belongs to. Here the self costs refer to the costs of a code region excluding the costs of calling other functions within the region itself.

After extracting the relevant performance information during the pre-processing step, this information is stored in a generic interface for the rest of the tool-flow, as Figure 2 shows. Each of the input granularities have in common that they can be represented in terms of a sequence of source code line numbers. Therefore, we use line number intervals to generically describe the granularities. For example, in Figure 1 the function `foo` has a line number interval of 1–11. Similarly, basic blocks can be described by intervals.

Finally, the estimate generator module adapts on demand the cost information available in the generic interface, to the cost information at the granularities requested by an MPSoC compiler. These requests are described in terms of line intervals and core types. A valid interval goes from one single line to an interval of lines that does not go beyond any function in the application. To complement the information from the generic interface, the estimate generator also performs a source code profiling run to extract execution count information both at function and basic block granularities, which is also needed during the adaptation process.

The estimate generator handles three scenarios for the requested intervals. In the *first scenario*, the interval exactly matches information available in the generic interface, therefore, the estimate generator simply forward the information from the generic interface to the MPSoC compiler, as follows:

$$Cost_r(L_a, L_b) = Cost_{gi}(L_x, L_y) \tag{1}$$

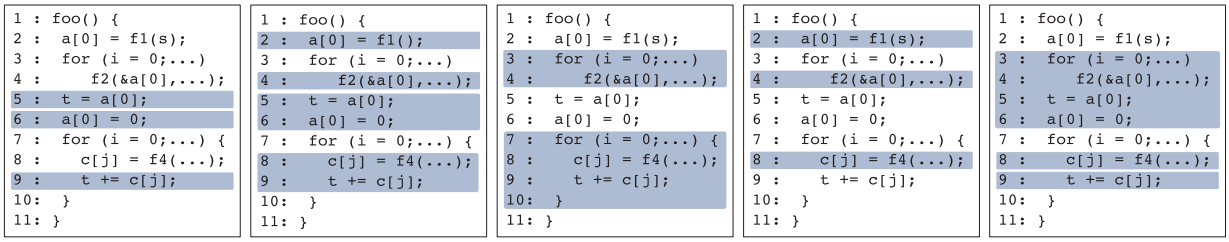
where, $[L_a, L_b] = [L_x, L_y]$. In the *second scenario* the requested interval is bigger than any interval available in the generic interface. Then, the requested cost information can be computed as the addition of all the non-overlapping intervals in the generic interface that are contained within the requested interval (e.g., function cost computed based on its basic block costs):

$$Cost_r(L_a, L_b) = \sum_{k=1}^{n} Cost_{gi}(I_k) \tag{2}$$

where $I_k \subset [L_a, L_b]$. Finally, in the *third and most interesting scenario* the requested interval does not match any interval in the generic interface, but there is an interval that includes the requested interval (e.g., statement cost computed based on its basic block cost). Therefore, the costs are estimated based on the smallest possible interval that contains the requested interval. This estimation relies on a weighted distribution approach, in which the

CASES '17 Companion, October 15–20, 2017, Seoul, Republic of Korea

ACM ISBN 978-1-4503-5184-3/17/10…$15.00
https://doi.org/10.1145/3125501.3125521



(a) Statement (b) Basic Block (c) Language Construct (d) Function (e) Arbitrary

Figure 1: Granularity examples

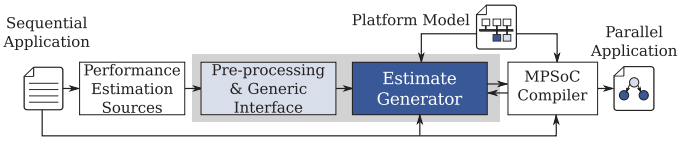


Figure 2: Estimate Generator Tool-flow Overview

weights are taken from a platform model (similar to [2]), as Figure 2 shows. In the platform model there is a cost table with latencies of each instruction for each core type in the target MPSoC. The weight for a given interval is computed as follows:

$$w(L_x, L_y) = \sum_{k=1}^{n} Latency(S_k) \cdot Count(S_k) \tag{3}$$

where S_k is the instruction, $Latency(S_k)$ is the latency of the S_k and $Count(S_k)$ is the execution count of S_k obtained from the profiling run. Then, the cost of the requested interval is as follows:

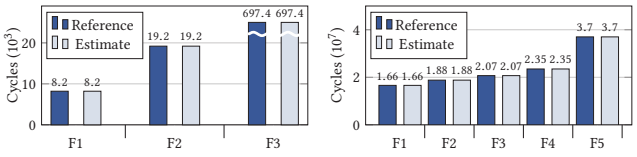
$$Cost_r(L_a, L_b) = \frac{w_r(L_a, L_b)}{w_{gi}(L_x, L_y)} \cdot Cost_{gi}(L_x, L_y) \tag{4}$$

where $[L_a, L_b] \subset [L_x, L_y]$, $w_r(L_a, L_b)$ is the weight of the requested interval, and $w_{gi}(L_x, L_y)$ is the weight and $Cost_{gi}(L_x, L_y)$ is the cost from the interval in the generic interface. The estimations presented in the described scenarios provide all the flexibility required by embedded parallelizing compilers to extract multiple patterns (e.g., task, pipeline and data level parallelism [1, 3]), especially when they work simultaneously at multiple granularities.

3 EXPERIMENTAL EVALUATION

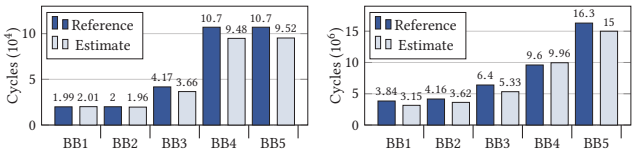
The focus of the evaluation is on the accuracy of the proposed framework. For the input performance estimation, we used the framework presented in [4], which provides estimations both at basic block and function granularities. The platform used was the Multi-core DSP Keystone C6678 from Texas Instruments [6], for which a model was built using its documentation. As case studies we used the `ADPCM` and `Trellis` applications from the UTDSP benchmark suite [7], which were evaluated using the optimization level `-O2` by the input performance estimation framework.

First we evaluated the second scenario according to Equation 2. Here, we took the input performance information at the basic block granularity and let the tool-flow estimate the function self costs. Then, we compared the estimations generated by our framework with the input reference estimations at the function granularity. As it can be observed from the results in Figure 3, there is no estimation error in both applications, which is an expected outcome since the estimation is simply the addition of the costs of all basic blocks within each function.



(a) ADPCM (b) Trellis

Figure 3: Estimation Results of Function Granularity



(a) ADPCM (b) Trellis

Figure 4: Estimation Results of Basic Block Granularity

Then we evaluated the third scenario according to weighted distribution approach described in Equation 4. Here we took the input performance information at the function granularity and estimated the basic block self costs. Figure 3 shows the results considering the most computational intensive basic blocks. We can observe that the error in the estimation for the `ADPCM` benchmark is between 1.13% and 11.12%, while for the `Trellis` benchmark the error is between 3.97% and 17.91%. The estimation errors in this scenario can be attributed to: i) inaccuracies in the latency values in the platform model, ii) awareness of the optimization level used while generating the input performance estimates.

4 CONCLUSIONS

In this paper, we presented a tool-flow that adapts performance information from state-of-the-art techniques, to the granularities required by modern parallelizing compilers for multicore embedded systems. The results show an estimation error under 18% for the evaluated benchmarks. In the future, we plan to further evaluate the accuracy of the proposed approach, and to assess the impact of our approach on the quality of the parallelization decisions.

REFERENCES

[1] Miguel Angel Aguilar et al. 2016. Towards Parallelism Extraction for Heterogeneous Multicore Android Devices. *IJPP* (2016), 1–33.
[2] Multicore Association. 2013. Software-Hardware Interface for Multi-Many-Core (SHIM) V1.00. [Online] http://www.multicore-association.org. (2013).
[3] Jeronimo Castrillon and Rainer Leupers. 2014. *Tool Flows to Close the Software Productivity Gap.* Springer.
[4] Juan Fernando Eusse et al. 2014. Pre-architectural Performance Estimation for ASIP Design Based on Abstract Processor Models. In *SAMOS 2014*. Greece.
[5] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 2004. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.* 39, 4 (April 2004), 49–57.
[6] Texas Instruments. 2016. Keystone Multicore Devices. [Online] Available http://processors.wiki.ti.com/index.php/Multicore. (2016).
[7] Corinna Lee. 1998. UTDSP Benchmark. [Online] http://www.eecg.toronto.edu/corinna/DSP/infrastructure/UTDSP.html. (1998).