Low-Level Profiling and MARTE-Compatible Modeling of Software Components for Real-Time Systems

Konstantinos Triantafyllidis, Egor Bondarev, Peter H.N. de With
Eindhoven University of Technology
5600 MB, Eindhoven, The Netherlands
{k.triantafyllidis, e.bondarev, p.h.n.de.with}@tue.nl

***Abstract*—In this paper, we present a method for (a) profiling of individual components at high accuracy level, (b) modeling of the components with the accurate data obtained from profiling, and (c) model conversion to the MARTE profile. The resulting performance models of individual components are used at the component composition (design) phases for detailed evaluation of the performance of the designed system. Furthermore, the profiled models serve as a valid source for architecture optimization of the composed system. The presented method is a constituent part of our complete Design Space Exploration (DSE) methodology [1, 2], which involves modeling of individual components, component composition, performance analysis of the designed composition, and architecture optimization. The contribution of this new profiling method is attractive in various ways: (a) The profiling is fast and detailed, which leads to accurate models, (b) it is generic, since it allows multiplatform execution and (c) it can be used by MARTE-based analysis tools, due to the model compatibility. Our discussed experiment has resulted in cycle-accurate models in less than 1 hour profiling effort per component.**

Profiling, modelling, component-based, MARTE, real-time systems

I. INTRODUCTION

In software engineering of real-time systems, a component-based development is becoming an adopted practice, since it enables a rapid system prototyping and development by composing a system from existing component blocks. Such component-based real-time systems still should satisfy all the types of performance requirements (latency, throughput, etc). At the early component-composition phases, an architect needs assessment methods to evaluate performance of the real-time system, and those methods are commonly built upon and rely on the models of the individual components. In the domain of real-time systems, such models should be accurate and reasonably detailed in order to provide meaningful and accurate system-wide assessment results, to predict if the designed system will meet its latency deadlines. At present, the problem of modeling of individual software components at high accuracy level is not solved, due to complex intrinsic properties and a large variety of available hardware blocks and software architectures.

Moreover, due to increasing diversification of hardware units and architectures (i.e. processors, memories and buses), it is challenging for component engineers to make universal component models that support this hardware diversity. The clock frequency of a Central Processor Unit (CPU) does not play a major role in system performance anymore. Additional system specifications, like CPU-cache size, have their own influence on the performance of a real-time system. Besides this, multicore architectures should be taken into account with respect to how the SW components are executed on them. Another issue that makes the problem more arduous is the correlation between the data input and the dissimilar resources demanded by the SW component on the performance appraisal of the system. Due to the widespread early termination technique used in multimedia applications, different data inputs may lead to completely different performance results.

Additionally, recently reported heterogeneous systems show a clear increase of complexity. Workload is often executed on a Multiprocessor System on a Chip (MPSoC), or co-executed both in CPU and Graphical Processor Unit (GPU) cores (CUDA [3] and Open CL [4]). Both cases are indispensable for analysis, since it is difficult to identify which process is executing in what core. Although real-time systems exist that migrate their workload into MPSoCs and GPUs, a high-detailed profiling and modeling tool that is capable of analyzing their performance behavior is required for a structured development of the execution .

In this paper, we propose a uniform method for profiling of individual components at high accuracy level, and modeling components with more accurate data obtained from profiling. Moreover, our method can be applied to the large class of multicore heterogeneous platforms. Our method is supported by the developed tool ProMo that performs both the SW component's profiling and the semi-automated creation of performance models. The ProMo tool is based on the PAPI library [9]. This library is used for highly detailed performance profiling, while the auto-generated performance models are based on the XMI framework. The ProMo tool supports the large majority of the HW platforms available in the market and, additionally, our profiling is faster compared to the MULTICUBE and Sesame simulation approaches. This is due to the fact that simulation is always slower than a direct execution/profiling of a compiled source code. Moreover, the direct execution provides more accurate performance measurements than the code simulation. ProMo tool is able to adopt performance modeling of those

individual SW components that support their co-execution on GPU cores.

The obtained profiled values of the performance attributes are stored into a ProMo performance model. This model is MARTE-compatible, since the ProMo tool converts the data into the MARTE-based format.

The sequel of this paper is as follows. Section II records the related to ours work. Section III explains the overall DSE methodology. Sections IV and V describe the architecture of the ProMo tool. Section VI describes the conversion of ProMo performance model to a MARTE-compatible model. Section VII illustrates a case study used for the validation of the profiling and modeling approach. Section VIII concludes on the benefits and limitations of this toolkit.

II. Related work

The most well-known profiling tool is Intel VTune [11]. Its main characteristic is that it provides accurate and highly detailed performance metrics. The VTune can collect information by two methods: by sampling events and by profiling the call stack. During the sampling method, low-level events can be measured, like cycles retired, cache misses, etc. During the profiling method, VTune counts how many times a function calls another function and determines the time spent on execution of each function. Moreover, the Intel VTune profiler supports a wide diversity of low-level events and enables a source code instrumentation, which makes it suitable for profiling and modeling of software components. However, it supports only the Intel CPU types and it is distributed via licensing. This limits its integration into a tool that should be able to profile and model an arbitrary set of component types on an arbitrary hardware infrastructure.

The AMD APP Profiler [13] is the profiler for AMD CPU families. Its main difference to the Intel VTune profiler is that it is an open source profiler and thus freely available. However, like VTune, it does not support CPU families other than AMD-based, with respect to low-level event profiling.

The most commonly used profiler for GNU applications is the *gprof* open-source profiler [20]. The gprof provides two types of performance analysis: (a) Line level Flat Profile and (b) Call Graph Analysis. Its major limitation is that the provided metrics are not sufficient for detailed component profiling and modeling, because it does not support low-level metrics extraction.

In the last decade, the real-time research community made several attempts to tackle the problem of high-detailed profiling and modeling, which resulted in a number of state-of-the-art methods. The MULTICUBE approach [5] deploys component profiling using the SCoPE tool [6]. During profiling, the SCoPE tool annotates source code of individual SW components with a computed cost during execution for each line of the source code. These costs can be mapped into processor-specific cost metrics, which are available in a repository xml-file. Unfortunately, to our knowledge, these cost metrics are available only for the ARM processor 926t, which limits the application area of the approach. Since the SCoPE tool uses a RISC-based cache simulator, it would be inaccurate to use the cache simulator for non-RISC processors. The usage of Single Instruction Multiple Data (SIMD) concept and the instruction set of the examined CPU are crucial factors influencing the Cycles Per Instruction (CPI) rate. However, the MULTICUBE approach does not fully address these factors. Taking the above-mentioned consideration into account, we can conclude that the MULTICUBE approach is applicable only to systems that use homogeneous processing platforms, i.e. deploy families of identical processors.

During our experiments with the SCoPE tool, we have found out that its core block, namely *scope-g++*, was unable to compile and instrument a couple of libraries, such as Eigen and Boost, since it does not recognize some complicated operations related to hash-based memory access and, therefore, is not able to instrument the cross-compiled source code while offering the corresponding cost of execution.

In the Sesame approach, Pimentel *et al.* [7] have proposed two different strategies to obtain the performance of SW components: the off-line and on-line calibration. Both strategies use the SimpleScalar simulator [8], which is a specific type of an Instruction Set Simulator (ISS). During off-line calibration and prior to the system simulation, SimpleScalar calibrates a table, that specifies the execution latency/cost of each instruction for a specific processor. The computed values are used for annotating the cross-compiled source code of a component with instruction latencies. After this, the simulator executes this code and computes the execution time of the functions that are provided by interfaces of the modeled SW component. Alternatively, during the on-line calibration, the tool traces the ISS execution and enhances the performance results obtained at the off-line calibration phase. Currently, Sesame supports only SimpleScalar ISS which, similarly to the MULTICUBE approach, limits the application area, due to the lack of support for multicore heterogeneous architectures. Summarizing, the above discussion reveals that the existing solutions cannot be broadly applied and are particularly necessary for multicore heterogeneous platforms. This will be the focus of our paper.

III. Architecture evaluation and optimization

In the last decade, we have conducted extensive research on Design Space Exploration (DSE) of various real-time systems [1, 2]. Our DSE methodology is subdivided into three phases, as depicted in Fig.1: (1) Profiling and Modeling of the SW components, (2) Architecture Composition and (3) Architecture Evaluation and Optimization of the composed system.

During the Profiling and Modeling phase, a component developer profiles and defines a highly accurate performance model of the implemented component. For each implemented function, the performance model describes the usage of hardware resources (processor, memory, bus and network). Such models are applicable to different hardware platforms. The performance models are placed into the repository, together with the component source code and/or executable modules.

The Architecture and Composition phase (in the middle) aims at the development of a number of architecture alternatives that would potentially satisfy the system requirements. The Analysis and Optimization phase (at the bottom) features prediction of the system performance by simulating the composed alternatives. The Pareto-optimal analysis aims at comparison of the alternatives accounting for the trade-offs between the following multi-dimensional criteria: latency, throughput, robustness and hardware resource load. After Pareto optimization, the results serve as guidelines for further optimization of the current optimal architectures, thereby enabling further iterative performance improvements of the designed system.

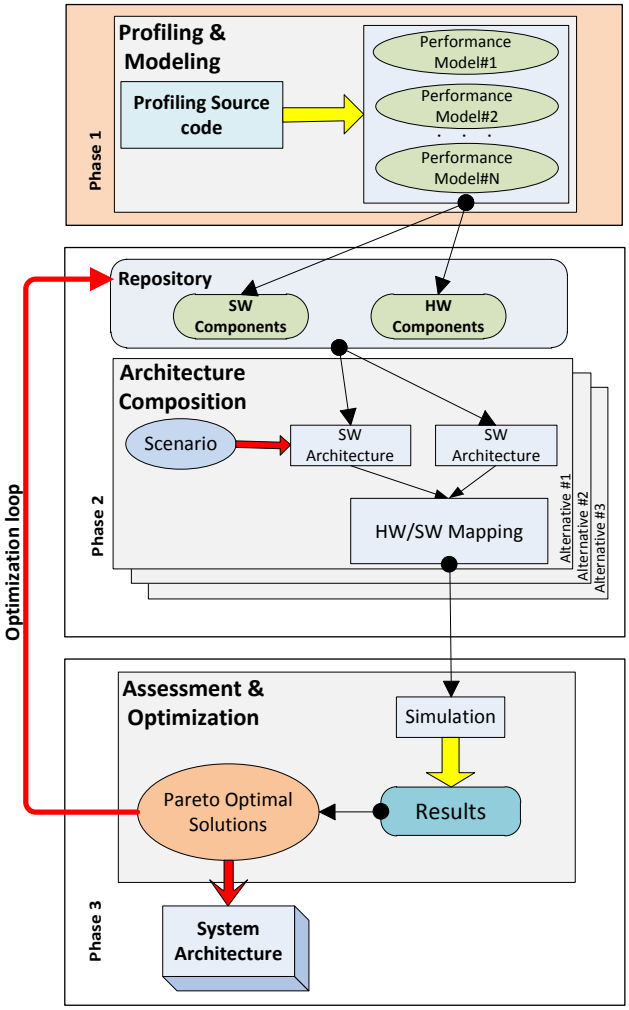

Figure 1. Overview of the DSE methodology, where are focus is on Profiling and modeling of components in the Phase 1.

IV. Detailed Profiling

In order to perform DSE on a real-time system, the architect needs a complete performance model, specifying execution behavior and hardware resource usage of each individual SW component. This model should be highly accurate and detailed, in order to enable valid analysis of the composed system. Furthermore, due to the diversity of available HW components, the performance model should be uniform and applicable to as many HW platforms as possible.

In order to achieve an accurate and detailed model, a careful profiling technique should be deployed. For this purpose, the ProMo tool integrates a cycle-accurate profiling technique, based on the PAPI library. The PAPI library is able to access the hardware performance counters of various CPUs. These counters are available in the majority of modern microprocessors, located at the so-called Performance Monitor Unit (PMU), where they count events and occurrences of specific signals related to corresponding processor functions. The main characteristic of the PAPI library is portability across different HW platforms [10], in contrast to other profiling tools, like Intel VTune [11], ARM Profiler [12] and the AMD APP Profiler [13]. It is feasible to profile an application or a component on different CPU platforms by just cross-compiling its source code and run the produced executable on the target system. Moreover, the PAPI profiler enables measuring a comprehensive set of events (cache misses, total instructions retired, branch instructions, etc.) already at the first execution. Finally, PAPI is widely supported by the majority of the Unix-based operating systems [10], which are broadly used for developing embedded and large-scale systems.

The number of the simultaneous events, collected by PAPI, depends on the CPU type used. In our case, the ProMo tool profiles the execution of an individual component and obtains the following performance metrics for each function in the provided interface of the component:

1. *Used number of instructions* and the *CPU cycles*,
2. *Cache-miss-rate* for the L2 and L3 caches,
3. Amount of *memory claimed* and *released*,
4. *Potential bus-load* due to R/W instructions.

The above metrics play the following roles in architecture analysis and optimization phases. The metric *used number of instructions* defines a component complexity, while the *CPU cycles* metric is used for computation of execution times of component functions. Additionally, by knowing the number of the *CPU cycles* retired, it is possible to estimate the execution time of this component, in case it is mapped on a similar CPU, but with a different clock frequency. Also, the division of *CPU cycles* by the *number of instructions* results in the *Cycles Per Instruction* (CPI) metric. This is a valuable metric for analysis and optimization phases, in case the hardware platform is heterogeneous and contains multiple types of CPU architectures (e.g. RISC-based and CISC-based).

The multi-level *cache-miss-rate* metric is used for (a) computation of the potential bus load, and (b) identification of the most appropriate CPU types during the optimization phase.

The computation of the *potential bus-load* is computed by utilizing the number of the cache misses and cache writes at the upper level of the cache hierarchy. For instance, in our

Intel Core i5 750 processor type, the upper level is the L3 cache, while the ARM Cortex-A8 has an L2 cache at the upper level. The bus-load is obtained by multiplying the cache misses and cache writes with the size of each cache line. Besides this, at the optimization phase, our DSE methodology uses the cache-miss rate to detect and avoid possible bottlenecks during automated generation of alternative architectures.

The *memory claimed* metrics is computed for each function of the provided interface of a component, and identifies the amount of memory used during the execution of the function. Similarly, the *memory released* metric identifies the amount of memory (delta) that has been released upon function completion. We compute the *memory claimed* and *released* metrics with the Linux-command library.

The obtained profiling values might differ for each individual profiling iteration, depending on the state of the system at the moment of execution (i.e. full cache of data, CPU and BUS usage from another application etc.). Therefore, to achieve model completeness, the engineer is advised to execute several profiling iterations, and use the worst and best case scenarios by modifying the input parameters that define the workload of the component. The iterations can stop at the saturation point, when the new values converge to the current value range.

As mentioned previously, the number of simultaneously collected events by the PAPI profiler depends on the type of CPU deployed. Commonly, this number ranges from 1 to 7. This limits the profiling metrics obtained by ProMo, while in an unlimited case the performance models could be even more detailed. That could be solved by iterating the profiling procedure several times for collecting different metrics at each iteration and integrate those into a more detailed performance model.

V. PERFORMANCE MODELING

We have defined a number of important requirements for model structuring. First, the component developer should be able to generate a performance model of his component with relatively low effort. Second, the model structure should enable model composition, i.e. during the component composition phase, the models of the individual components can be composed into a system-wide performance model. Finally, the performance model should be able to incorporate multiple profiling data, such that the model becomes applicable to a large variety of hardware platforms and CPU families.

In our approach, the source code profiling measurements should be made available for the later Architecture Composition phase. For this purpose, the ProMo tool stores the obtained measurements into a performance model that can be shipped to an architect along with the component executables, or stored into a repository for later (re)use.

The model structure complies with the XMI standard, so that, the developed performance models can be easily adopted by existing DSE tools, like CARAT [14], AQOSA [1], ArcheOpterix [15] and Sesame [7]. Moreover, the performance model can be converted into MARTE-compatible [16] or AADL-compatible [17] resource models.

Let us outline the structure of the proposed performance model in the next Section. The structural view on the model is depicted in Fig. 2. We model performance attributes for each function of the provided interfaces of the component. For each function, the attributes are grouped into four types: *cpuUsage*, *memoryUsage*, *busUsage* and *networkUsage*. An additional type, called *Condition*, is used to enable parameter-dependent (input-data dependent) modeling.

The *cpuUsage* node contains as many sub-nodes, as the amount of target CPUs. Each time the SW component is profiled on a different CPU type, the metrics are saved into a new sub-node of the *cpuUsage* primitive. Each sub-node includes the following attributes:

- instructions_avg, min, max,
- cycles_avg, min, max,
- L2_miss_avg, min, max,
- L3_miss_avg, min, max.

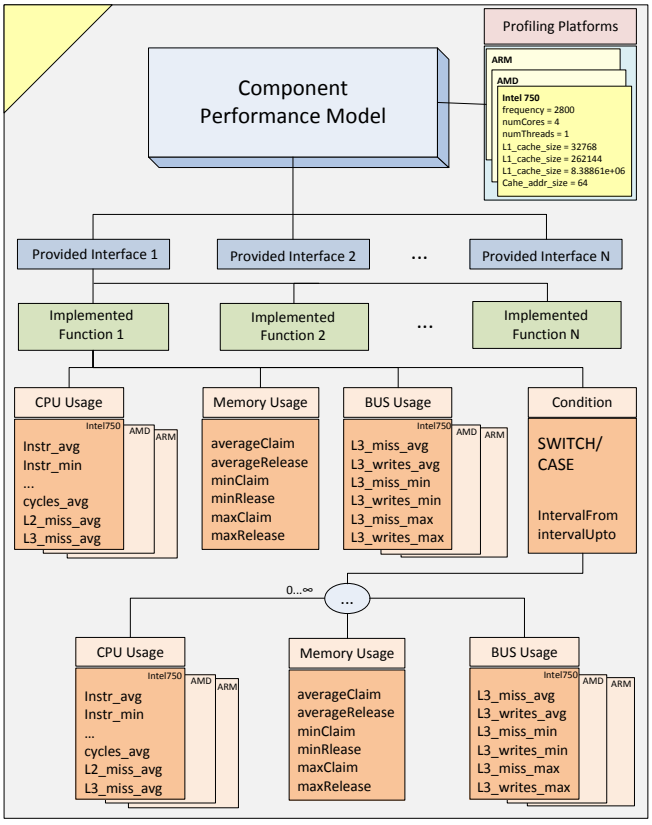

Figure 2. Structure of Performance Model.

As can be noticed, each attribute can incorporate average, minimum and maximum values. For obtaining these three values, the profiler runs the component execution multiple times. The architect can select them, either for average-case worst-case analysis, or best-case performance analysis.

Besides this, each *cpuUsage* sub-node is annotated with the attributes of the CPU used during profiling (frequency,

number of cores, number of threads, L1, L2 and L3 cache-sizes). They are obtained during the initialization of the profiling tool.

The *memoryUsage* node specifies two main metrics: (a) the *claimed* and (b) the *released* amount of memory during the execution of each component function. Similarly to the *cpuUsage* node, the memory metrics incorporate the average, minimum and maximum values.

The *busUsage* node can be specified for different CPU types and contains the following attributes:

- L3_miss_avg, min, max,
- L3_cache_writes_avg, min, max.

The attributes of the *busUsage* node are important for analysis of the bus-load at the Architecture Assessment phase.

```
Name of SW component = PoissonReconstruction
    Profiling Platforms
        Intel (R) Core (TM) i5 CPU @ 2.67GHz
                numThreads = 1
                frequency = 2800
                number_of_cores = 4
                L1_cache_size = 32768
                L2_cache_size = 262144
                L3_cache_size = 8.38861e+06
                cache_addr_size = 64

        Intel(R) Xeon(R) CPU W3530 @ 2.80GHz
                numThreads = 2
                frequency = 2800
                number_of_cores = 4
                L1_cache_size = 32768
                L2_cache_size = 262144
                L3_cache_size = 8.38861e+06
                cache_addr_size = 64

    Name of function = setTree
        property = cpuUsage
            Name of CPU = Intel (R) Core (TM) i5 CPU @ 2.67GHz
                instructions_avg, min, max = 4.70831e+08
                cycles_avg, min, max = 2.99188e+08, 2.97153e+08, 3.0048e+09
                L2_miss_avg, min, max = 137610, 136976, 138480
                L3_miss_avg, min, max = 2723.53, 2038.88, 4267.54"

            Name of CPU = Intel(R) Xeon(R) CPU W3530 @ 2.80GHz
                instructions_avg, min, max = 5.08946e+08
                cycles_avg, min, max = 2.62671e+08, 2.59245e+08,2.69307e+09
                L2_miss_avg, min, max = 178060, 176518, 180955
                L3_miss_avg, min, max = 2942.94, 2668.64, 3599.31

        property = busUsage
            Name of BUS = bus_Intel (R) Core (TM) i5 CPU @ 2.67GHz
                L3_miss_avg, min, max = 2723.53, 2038.88, 4267.54
                L3_cache_writes_avg, min, max = 4940.33, 4919.92, 4964.36

            Name of BUS = bus_Intel(R) Xeon(R) CPU W3530 @ 2.80GHz
                L3_miss_avg, min, max = 2942.94, 2668.64, 3599.31
                L3_cache_writes_avg, min, max = 7115.8, 7025.43, 7317.74

        property = memoryUsage
            averageClaim, averageRelease = 44040
            minClaim, minRelease = 44036
            maxClaim, maxRelease = 44052

    Name of function = GetIsoValue
            ….
            ….
    Name of function = GetMCIsoTriangles
            ….
            ….
END PoissonReconstruction
```

Figure 3. Pseudo-xml code of the performance model of Poisson Reconstruction component.

The *Condition* node enables parameter-dependent modeling of the component performance. Different input data can lead to completely different performance results. The *Condition* node allows specifying the CPU, memory and bus usage for a specific value interval (*intervalFrom* and *intervalUpto*) of the input parameters.

The performance model example is shown by the model of the PoissonReconstruction component (see Fig. 3). This model is a part of our case study, presented in Section VII.

The ProMo tool profiles and constructs the performance models in a semi-automatic way. The component developer defines the blocks of code or functions that need to be profiled and modeled in the source code of the SW component (source code instrumentation). The remaining actions are done fully automatically. The ProMo tool profiles the instrumented source code and constructs a performance model.

VI. CONVERSION TO MARTE COMPATIBILITY

In order to adopt commonly-used modeling practices, we have developed a converter, which translates our performance model into a *MARTE-compatible* model, and as a consequence, into an AADL-compatible model (the majority of *AADL* models can be described by the *MARTE* profile [19]).

The term MARTE stands for Modeling and Analysis of Real-Time and Embedded Systems, giving a profile that is an extension of the UML profile, and it covers the field of embedded and real-time systems. In brief, MARTE offers the following fundamental features:

- QoS-aware Modeling,
- Architecture Modeling,
- Platform-based Modeling,
- Model-based QoS Analysis.

Here we focus on Platform-based Modeling, which is composed from the following three profiles: GRM (Generic Resource Modeling), SRM (Software Resource Modeling) and HRM (Hardware Resource Modeling). Despite the fact that the MARTE profile allows specification of a wide diversity of performance attributes, it does not contain a few low-level attributes that we obtain during the profiling phase.

In order to be able to convert our performance model into a MARTE-compatible model, we add the missing low-level attributes to the ResourceUsage stereotype (GRM profile) as the following set of new performance properties:

- instructions_retired,
- cycles_retired,
- L1_cache_miss,
- L2_cache_miss,
- L3_cache_miss,
- L2_cache_writes,
- L3_cache_writes.

We assign these properties to the NFP_CommonType, supported by the MARTE library, since it contains several value types, like mean, max, min, etc., which is also used in our performance model specification.

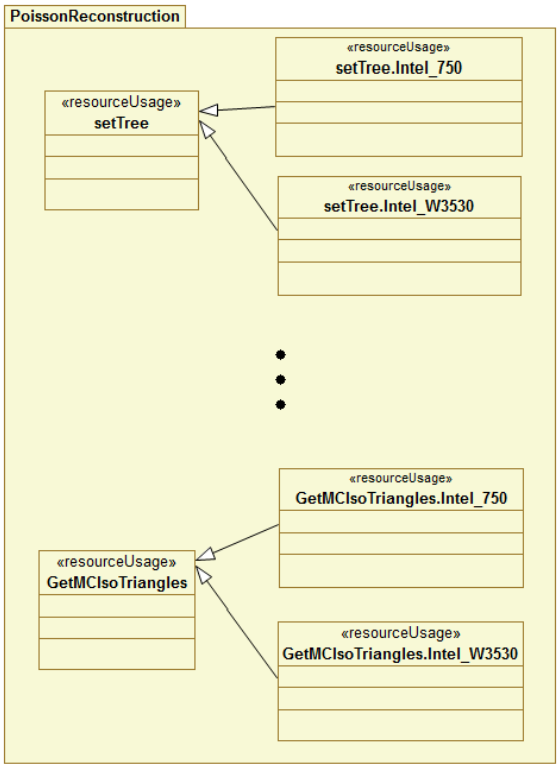

Figure 4. Representation of the Poisson Reconstruction SW component in the MARTE performance model.

Let us now describe the conversion results by outlining the MARTE-compatible performance model of the PoissonReconstruction SW component (initially depicted in Fig. 3).

The converted MARTE-compatible performance model is a package called PoissonReconstruction (see Fig. 4), and it includes several classes. The package represents the whole SW component, while the individual class represents exactly one ImplementedFunction of the component. For each class, we define at least one *instantiation* (like *implementation* in the AADL modeling approach) by using the Realization link [19]. Each individual *instantiation* is dedicated to a specific HW platform, on which the ImplementedFunction has been profiled. For instance, for the ImplementedFunction `setTree`, we model two class instantiations named: `setTree.intel_i5_750` and `setTree.XEON_W3530`. The classes that represent the ImplementedFunctions and their instantiations are of the MARTE stereotype ResourceUsage. Fig. 5 depicts the new properties that have been introduced into the ResourceUsage profile with the values transferred from the ProMo performance model.

```
<GRM:ResourceUsage xmi:id="id of setTree.Intel_750">
 <instructions_retired>
        4.70831e+08
 </instructions_retired>
 <cycles_retired>
        (2.99188e+08,mean),(2.97153e+08,min),(3.0048e+09,max)
 </cycles_retired>
 <L2_cache_miss>
        (137610,mean),(136976,min),(138480,max)
 </L2_cache_miss>
 <L3_cache_miss>
        (2723.53,mean),(2038.88,min),(4267.54,max)
 </L3_cache_miss>
 <L3_cache_writes>
        (4940.33,mean),(4919.92,min),(4964.36,max)
 </L3_cache_writes>
</GRM:ResourceUsage>

<GRM:ResourceUsage xmi:id="id of setTree.Intel_W3530">
 <instructions_retired>
        5.08946e+08
 </instructions_retired>
 <cycles_retired>
        (2.62671e+08,mean),(2.59245e+08,min),(2.69307e+09,max)
 </cycles_retired>
 <L2_cache_miss>
        (178060,mean),(176518,min),(180955,max)
 </L2_cache_miss>
 <L3_cache_miss>
        (2942.94,mean),(2668.64,min),(3599.31,max)
 </L3_cache_miss>
 <L3_cache_writes>
        (7115.8,mean),(7025.43,min),(7317.74,max)
 </L3_cache_writes>
</GRM:ResourceUsage>
```

Figure 5. Resource model of setTree function as part of the MARTE xml model.

VII. CASE STUDY

In order to validate our profiling and modeling approach and to test the robustness of the ProMo tool, we have applied it to our currently developed image processing system for 3D volumetric reconstruction. The system features intensive computation of 3D point clouds, polygonal meshes and HD image processing. The automatically obtained performance models are validated and tested during the architecture evaluation phase of the system development. Fig. 3 illustrates the performance model of our Poisson 3D reconstruction algorithms in a pseudo-code format, where the model contains the actual performance values obtained by profiling of the Poisson algorithm [18]. Fig. 3 presents a partial model for a *setTree* function.

The profiling has been performed on two different platforms: Intel i5 and Intel Xeon (see the specification in Table 1).

Comparing the component performance metrics obtained by profiling on the two different processors, we have revealed that, even if the processors belong to the same processor family Nehalem and have identical clock frequencies, the performance results of the SW component executions are different. This indicates a conclusion that the clock frequency does not play a definitive role in the component execution performance. Instead, other attributes, such as cache-hierarchy, pipeline architecture and bus frequency, influence the performance significantly and therefore, such factors should be carefully considered.

TABLE I. HW PLATFORMS USED FOR PROFILING

Computer HW						
CPU				***BUS***	***RAM***	
Model	Freq.	Cache	#cores/ #thrds	Trfr. rate	Size	Freq.
Intel Core i5 750	2.8 GHz	8MB	4/4	2.5 GT/s	4GB	1333 MHz /dual channel
Intel Xeon W3530	2.8 GHz	8MB	4/8	4.8 GT/s	6GB	1066 MHz /triple channel

Summarizing, we note that the ProMo tool has provided cycle-accurate performance values in a consistent way, since these values have been multiple times validated by several execution runs. We can also report that the average time spent on profiling and modeling of each component of the composed 3D reconstruction system equals 50 minutes (including the multiple profiling iterations).

This case study (as well as the system development) is still ongoing, so that the input data sets are still rather limited and the real-time deadlines/throughput requirements are not yet set clearly. We will provide more comprehensive reports on the completed case study in a succeeding publication.

VIII. CONCLUSIONS

In this paper, we have presented a method for profiling of individual components at high accuracy level, and modeling of the components with the accurate data obtained from that profiling. This has resulted to a new tool called ProMo for execution a larger framework for DSE. The tool profiles the cross-compiled source code of the SW component and generates a highly-accurate performance model in an automated way.

The performance models can be composed into a system-wide performance model, which can be used for analysis of component-based real-time systems. Moreover, the performance models can play an important role in the architecture optimization (DSE) phases, since they enable rapid prediction of potential bottlenecks, latencies and resource loads for multiple alternative architectures.

The *advantages* of the ProMo tool become manifest in multiple ways. Firstly, the profiling phase is *generic*, supporting the majority of the CPUs available, so that it can execute at multi-core heterogeneous processor platforms. Secondly, the performance results are characterized by their cycle-accurate level, since the measurements are directly collected from the PMU of the attached CPU. Hence, the execution delivers the computation cost directly. Thirdly, the obtained performance models adhere to the XMI standard and they are compatible with MARTE or AADL resource models. Therefore, they can be easily adopted by existing DSE tools. Fourthly, the highly detailed metrics obtained by profiling, serve as a valid source for accurate architecture assessment and succeeding optimization phases. Fifth and finally, the ProMo tool features rapid profiling and model generation, due to its execution (vs. simulation) nature.

The ProMo tool imposes a number of *limitations* that need further research. Firstly, it is supported only by Unix-based (open-source) operating systems, since it deploys the PAPI library. This limits the architectural choice for operating systems to be used during the design space exploration of a real-time system. Another limitation is that the component developer is advised to profile his component on multiple hardware platforms, to facilitate the component deployment on heterogeneous systems. Finally, a low-level performance model of a complex component may become very large, which makes it difficult to understand for an architect.

This paragraph describes the *observations* and *lessons learned* during the case study experiments. Firstly, we have realized that the component performance (execution time) on the two different deployed platforms has shown a variation difference of 10%, even despite the fact that the CPU platforms are very similar in the clock frequency and cache hierarchy. This occurs due to the different BUS Transfer Rate that each CPU offers and due to the different pipeline architectures. Secondly, an interesting observation is the large difference (more than 12%) between the cache misses of the two CPUs. Since the two CPUs adopt the same cache hierarchy, and both platforms have only executed this single case study application, we initially expected these values to be similar. In this case, we can make the hypothesis that the CPUs use different cache policies. Thirdly, due to our observation that clock frequency did not play a definitive role in the component performance, we have started looking more carefully at other factors, like cache-hierarchy, pipeline architecture and bus frequency. Finally, we have found that it is difficult to identify the number of iterations required for achieving a complete and accurate performance model. This is due to the non-consistent system execution behavior. As a consequence, the formal identification of the saturation point in the iterative execution profiles is a challenging task, which we leave for future research.

Let us briefly outline the other *future research* steps. Due to the increasing popularity of applications which target and can be executed on the Graphics Processor Units (GPU) cores, it is vital to support profiling and modeling of SW components from such applications. By integrating the PAPI-CUDA library, the ProMo tool is able to profile and model the SW application components that dynamically migrate their workload into CPU and GPU cores. The challenge is that the profiling needs to be iterated several times, in order to collect all the required events and to construct a consistent performance model. During these iterations, the point of profiling saturation cannot be currently identified in a fully automatic way. Moreover, we are working on creating automatic high-detailed resource models for HW by obtaining information from the applied execution platform. Finally, since our approach requires the user-based code instrumentation and saturation-point detection, we are implementing a GUI which facilitates the complete procedure.

REFERENCES

[1] Li R., Etemaadi R., Emmerich M.T.M., Chaudron M.R.V., "An evolutionary multiobjective optimization approach to component-based software architecture design," Evolutionary Computation (CEC), 2011 IEEE Congress on , vol., no., pp.432-439, 5-8 June 2011 doi: 10.1109/CEC.2011.5949650.

[2] L. Grunske, P. A. Lindsay, E. Bondarev, Y. Papadopoulos, and D. Parker, An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems, Architecting Dependable Systems IV, Lecture Notes in Computer Science, Heidelberg, Germany, ISBN 978-3-540-74033-9, 2007. Springer Berlin, pp. 188–209.

[3] CUDA: http://www.nvidia.com/object/cuda_home_new.html

[4] Open CL: http://www.khronos.org/opencl

[5] Silvano, C. et al. "MULTICUBE: Multi-objective Design Space Exploration of Multi-core Architectures," VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on , vol., no., pp.488-493, 5-7 July 2010, doi: 10.1109/ISVLSI.2010.67.

[6] Scope tool, user manual 1.15: http://www.teisa.unican.es/gim/en/scope/scope_web/scope_home.php

[7] Pimentel A.D., Thompson M., Polstra S., Erbas C., "On the Calibration of Abstract Performance Models for System-level Design Space Exploration," Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006.

[8] Austin, T., Larson, E., Ernst, D., "SimpleScalar: an infrastructure for computer system modeling," Computer , vol.35, no.2, pp.59-67, Feb 2002, doi: 10.1109/2.982917.

[9] Terpstra D., Jagode H., Haihang Y., Dongarra J., "Collecting Performance Data with PAPI-C", Tools for High Performance Computing 2009, Springer Berlin Heidelberg, 2010, vol., no., pp.157-173, do: 10.1007/978-3-642-11261-4_11.

[10] PAPI tool, Supported Platforms: http://icl.cs.utk.edu/papi/custom/index.html?lid=62&slid=96

[11] Intel VTune Performance Analyzer: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe

[12] ARM Profiler: http://www.arm.com/products/tools/software-tools/rvds/arm-profiler.php

[13] AMD APP Profiler: http://developer.amd.com/tools/AMDAPPProfiler/Pages/default.aspx

[14] E. Bondarev, M. R. V. Chaudron, and P. H. N. de With, Carat: a toolkit for design and performance analysis of component-based embedded systems, DATE '07: Proceedings of the conference on Design, automation and test in Europe, San Jose, CA, USA, ISBN 978-3-9810801-2-4, 2007. EDA Consortium, pp. 1024–1029.

[15] Aleti A., Bjornander S., Grunske L., Meedeniya I., "ArcheOpterix: An extendable tool for architecture optimization of AADL models," Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES '09. ICSE Workshop on , vol., no., pp.61-71, 16-16 May 2009, doi: 10.1109/MOMPES.2009.5069138.

[16] Taha, S., Radermacher, A., Gerard, S., Dekeyser, J.-L., "An Open Framework for Detailed Hardware Modeling," Industrial Embedded Systems, 2007. SIES '07. International Symposium on , vol., no., pp.118-125, 4-6 July 2007, doi: 10.1109/SIES.2007.4297325.

[17] Feiler Peter, H., Lewis, Bruce A., Vestal, Steve., "The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems," Computer Aided Control System Design, 2006 IEEE International Conference on Control.

[18] Poisson code: http://www.cs.jhu.edu/~misha/Code/PoissonRecon

[19] UML Profile for MARTE: Modeling and Analysis of Real-Time embedded systems: http://www.omg.org/spec/MARTE/1.1/PDF/

[20] GNU gprof Profiler: http://www.gnu.org