# The Impact of Intra-core and Inter-core Task Communication on Architectural Analysis of Multicore Embedded Systems

Juraj Feljan, Jan Carlson
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
Email: juraj.feljan@mdh.se, jan.carlson@mdh.se

***Abstract*—In order to get accurate performance predictions, design-time architectural analysis of multicore embedded systems has to consider communication overhead. When communicating tasks execute on the same core, the communication typically happens through the local cache. On the other hand, when they run on separate cores, the communication has to go through the shared memory. As the shared memory has a significantly larger latency than the local cache, we expect a significant difference between intra-core and inter-core task communication. In this paper, we present a series of experiments we ran to identify the size of this difference, and discuss its impact on architectural analysis of multicore embedded systems. In particular, we show that the impact of the difference is much lower than anticipated.**

***Keywords*—software architecture; model-based analysis; multi-core embedded systems; task communication; measurement; cache**

## I. INTRODUCTION

The majority of computer systems in use today are embedded systems. An embedded system is a microprocessor based system with a typically single dedicated function (as opposed to general purpose computer systems), embedded in and interacting with a larger device. Embedded systems range from simple devices (e.g., MP3 players) to complex systems consisting of multiple nodes communicating over a network (e.g., process controllers), and are used ubiquitously, as we can find them in industry, transportation, medicine, communication, entertainment, commerce, etc.

Today, embedded systems have more complex functionality than ever. At the same time, pieces of functionality that were traditionally realized in hardware are instead implemented in software (e.g., software-defined radio [1]). This makes today's embedded systems increasingly performance intensive. Similarly to general purpose computer systems, there is a trend to tackle the increasing performance demands of embedded systems by increasing the number of processing units, for example by using multicore technology. A multicore processor is a single chip that contains two or more processing units (cores) that are coupled tightly together in order to increase processing power while keeping power consumption reasonable.

Introducing additional processing units increases the performance capacity, but on the other hand introduces the problem of how to best allocate (partition) the software to the available cores, as the allocation has a substantial impact on the performance. A possible way of determining whether a particular allocation of software to cores gives satisfactory performance is to implement, deploy and run the system, in order to collect performance measurements. However, rather than employing such a "fix-it-later" approach, in line with software performance engineering [2], a preferred approach would be to predict the performance with a sufficient accuracy early in the development process, based on architectural models of the system. That way we can get an indication towards good allocations, and avoid time-consuming and costly redeployment of the system when using an iterative measurement-based method. The earlier in the development process that a design fault is caught, the cheaper and simpler it can be fixed. Also, by using models of the system, it is possible to try a large number of candidate allocations in shorter time than by measuring.

In our current work [3], we are investigating an approach for optimizing the allocation of software modules to the cores of a multicore embedded system, with respect to performance. Here, communication time plays a significant role, as it impacts performance aspects relevant in the domain of embedded systems, such as throughput and response time. In a multicore system, the communication time is affected by the allocation of software modules to the available cores. If two communicating software modules run on the same core, the communication normally happens through the local cache and has thus the potential to be much faster than communication between two modules running on different cores, which happens through the shared cache or the main memory. As our work includes design-time model-based performance predictions, we have to take these differences in communication duration into account, in order for the performance predictions to be accurate.

In this paper, we investigate the impact that the allocation of software modules to the cores of a multicore system has on communication time. By performing measurements on a running system, we determine the difference between intra-core communication and inter-core communication under varying conditions. We show that in many situations the difference is significantly lower than we expected, and discuss the reasons and implications of this, namely that the impact of this difference on design-time model-based performance analysis is limited.

The paper is organized as follows. In Section II, we describe the preliminaries and present the motivation for investigating the difference between intra-core and inter-core communication, from the perspective of our current work. In Section III, we give an overview of related work. Section IV is the core of the paper: first it reasons about the expected difference between intra-core and inter-core communication in different scenarios, then it describes the setup of the performed experiment, and finally it gives an interpretation of the results. Section V concludes the paper with a discussion of what the experiment results mean in the context of architectural analysis of multicore embedded systems.

## II. Background

The scope of our work are modern (and future) embedded systems whose hardware architecture resembles the one of today's general purpose computers. There is a recent trend of embedded systems moving from single core CPUs to complex multicore CPUs. For example, processors used in today's smartphones and microcontroller boards support up to 4 cores at 2.5 GHz (e.g., ARM Cortex-A15 MPCore [4], Qualcomm Krait 400 [5]).

Typically, each core of a multicore processor has a small on-chip memory (cache), while a larger off-chip main memory (RAM) is shared between the cores. The cache keeps a copy of a subset of data present in the RAM, in order to make this data available to the CPU at a much lower latency than when accessing data from the RAM. For this, the cache utilizes the fact that the same data is often re-accessed frequently (temporal locality of data), and that data being accessed close in time is often stored in adjacent memory locations (spatial locality of data). Other than the local cache (called L1 cache), modern processors typically have additional levels of cache. L2 cache is usually shared between pairs of cores, while L3 cache is shared between all cores. The latency of a particular memory grows in the following order: L1 cache, L2 cache, L3 cache, RAM. Even when having a particular CPU in mind, it is difficult to characterize these values with concrete numbers, but in general L2 cache latency is roughly two to three times larger than L1 cache latency, L3 cache latency is roughly ten times larger than L1 cache latency, and finally RAM latency is two orders of magnitude larger than the latency of L1 cache [6], [7]. When data is transferred between the different memories, it is done in bigger blocks of fixed size called cache lines. A cache line is usually several tens of bytes long.

The software architecture of embedded systems typically consists of a set of concurrent communicating software modules called tasks. The decision of which task to run on which core (i.e., the allocation of tasks) impacts the performance of the system. The extent of the impact depends on the particular performance aspect we consider. For example, schedulability is directly determined by the allocation. If too many tasks are allocated to a single core, the core will be overloaded. As a consequence, tasks will miss their deadlines which is not acceptable for systems with real-time requirements, which embedded systems often have. Similarly to schedulability, it can be expected that task allocation has a large impact on communication time. Two tasks running on the same core can communicate through the L1 cache, while two tasks running on different cores have to communicate through one of the shared memories. This means that intra-core communication should be considerably faster than inter-core communication.

Our current work [3] focuses on optimizing the allocation of tasks to the cores of a multicore embedded system. Already early in the development process, before the implementation, we want to be able to identify the allocations that will result in a system with good performance. We start with an architectural model of the system in terms of tasks and the connections between them, and a model of the hardware platform the system will run on. By an automatic model-to-model transformation, from the architectural and platform models we obtain an executable model of the system, and by simulating this model we get performance predictions for the system. This way we are able to test many allocations in search for the ones that give satisfactory performance. With the term performance, here we mean aspects like throughput and response time. These aspects depend on the communication time, which in turn depends on the allocation of tasks to the cores, as stated above. Therefore, in order to be able to give sufficiently precise performance predictions, we need to identify the difference in communication time depending on whether tasks communicate locally with other tasks running on the same core, or globally with tasks running on different cores. Due to the considerable differences in latencies between the different memories, we intuitively expect this difference to be significant.

## III. Related work

Even though the work presented in this paper touches upon research on caches in multicore systems and research on detailed performance evaluations of multicore systems, the context of the work lies in the field of architectural analysis and optimization of embedded systems. We therefore focus the discussion about related work to this research area.

Architectural analysis and optimization of embedded systems can be viewed as a subfield of software performance engineering [2]. Research in this field has a general goal of being able to reason about the performance of embedded systems, already prior to the implementation. At this early stage, embedded systems are typically specified as (more or less formal) models, which can be analyzed or simulated in order to get performance predictions. Often these approaches are complemented with architectural optimization — model-based assessment of particular architecture candidates is enhanced with a mechanism for finding a good architecture. For all but the most trivial embedded systems, evaluating all possible architecture candidates is not feasible, so typically architecture optimization involves a search process aided by heuristics, whose goal is to find near-optimal architectures. In the remainder of this section, we describe several prominent approaches for architectural analysis and/or optimization of embedded systems, both academic and industrial.

ProCom [8] is a component-based and model-based approach for embedded systems in the automotive domain. A ProCom a component is a set of code, documentation, models and extra-functional properties. By utilizing different modeling formalisms, ProCom can analyze worst-case execution times, end-to-end response times and resource usage of embedded systems.

DeepCompas [9] is an analysis framework for predicting performance related properties of real-time embedded systems. The basis of the approach are composable models of individual software components and hardware blocks, which are then synthesized into an executable model of the system. Simulation-based analysis of the executable model results with predicted performance properties for the system. DeepCompas also makes a step towards architecture optimization, by providing support for performing trade-off analysis between several architecture alternatives.

ArcheOpterix [10] is a framework for optimizing embedded system architectures modeled in the Architecture Analysis and Description Language (AADL) [11]. The quality attributes supported by the approach include reliability, performance and energy. One of the key characteristics of the approach is that (through its extension called Robust ArcheOpterix [12]) it takes into account the uncertainty of design-time parameter estimates, and can find architectures that reduce the impact of the uncertainties.

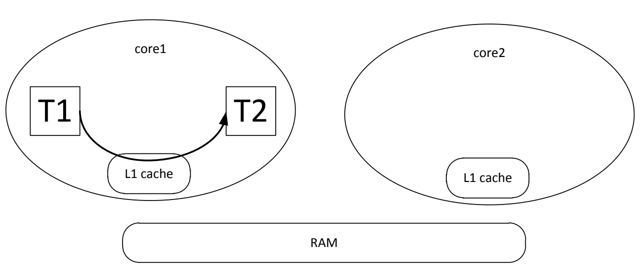
A defacto industry standard for model-based analysis of embedded systems is Mathworks Simulink [13]. It is a graphical tool that comes with built-in libraries of blocks (for instance the Stateflow toolbox for defining and executing state charts) that enable analysis and simulation of embedded systems, and ultimately code generation. It is also possible to define custom blocks using the Matlab programming language, which makes Simulink extendable with custom analysis and simulation techniques.

Additional approaches (not limited to embedded systems) can be found in Koziolek's survey of component-based approaches for performance evaluation [14], and in the survey of architecture optimization approaches by Aleti et al. [15].

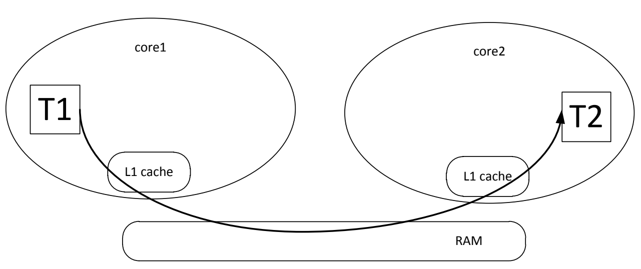
## IV. Intra-core vs. inter-core task communication

In this section, we first discuss in more detail about the expected difference between intra-core and inter-core communication in various scenarios. Then, in a separate subsection, we give details about the experiment setup — we describe the hardware and software environments, the general task model and the concrete task setup used in the experiment, and the variation points of the experiment. Finally, again in a separate subsection, we provide an interpretation of the experiment results.

Since many factors other than allocation influence the communication time, such as interruptions from other tasks, we start by identifying the case that has the highest potential of exhibiting a significant difference between intra-core and inter-core communication duration. Imagine the following scenario (Figure 1): a dual-core system, where each core has L1 cache, and the cores share the RAM. There are two communicating tasks: task $T_1$ produces (writes) data which task $T_2$ consumes (reads), and task $T_2$ runs immediately after task $T_1$ completes. The data fits in the L1 cache. If both tasks run on $core_1$ (scenario depicted in Figure 1a), task $T_2$ can obtain the data directly from the L1 cache on $core_1$, where it was written when task $T_1$ produced it. On the other hand, if task $T_1$ is running on $core_1$ and task $T_2$ on $core_2$ (scenario depicted in Figure 1b), the data produced by task $T_1$ is stored in the L1 cache of $core_1$ and not in the L1 cache of $core_2$. So $T_2$ will have to fetch the data from the RAM. Accessing the RAM is around a hundred times slower than accessing L1 cache, so inter-core communication should be significantly slower than intra-core communication. If the system also had shared L2 cache, the reasoning would still apply — since the latency of L2 cache is around two to three times larger than the latency of L1 cache, the difference in communication times should be smaller than in the case when there is no shared cache, but significant nevertheless.



(a) Intra-core communication



(b) Inter-core communication

Fig. 1: Task communication in a dual-core system

If two communicating tasks do not run immediately after each other, or if they get preempted by a higher priority task, the data they share might be evicted from the cache, due to other data taking its place. The longer the duration between producing and consuming a particular piece of data, the more likely other data will occupy the cache. In such cases even intra-core communication will have to go through the shared memory, thus reducing the communication time gain from allocating communicating tasks to the same core. Similarly, if the data being communicated does not fit in the local cache, the communication will have to go through the shared memory and the difference between intra-core and inter-core communication is reduced.

### A. Experiment setup

We use a system with an Intel Core 2 Duo E6700 processor [16]. Each core of this dual-core processor has 32 kB of local L1 cache, while 4 MB of L2 cache is shared between the cores. The cache lines in all caches are 64 bytes long. The system runs the 32-bit version of the Ubuntu 12.04 LTS operating system (kernel version 3.2.29) patched with the PREEMPT RT patch (version 3.2.29-rt44) [17], which turns the stock Linux kernel into a hard real-time kernel. By reducing the overall jitter and enabling the tasks to run at the highest priority, in combination with a high resolution timer of nanosecond granularity, this contributes to reducing unwanted interference in the experiments and increasing the precision of the measurements.

Next, we describe the task model used in the experiments. Tasks are implemented as Posix threads [18], and have read-execute-write semantics, meaning that they first read input data, then preform calculations and finally write output data. A task can either be periodic or event-triggered. A periodic task is activated at regular time intervals, while an event-triggered task is activated when the task it receives data from finishes executing. We assume that the tasks exchange data through shared memory, and that each core has access to the whole main memory. Other models (e.g., distributed memory, where each processor has its own local main memory), are possible but since they are not common in embedded systems, they are out of the scope of this paper.

As identified above, the biggest difference between intra-core and inter-core communication should happen in the case of two communicating tasks that share data which fits into the L1 cache, and the reader task runs immediately after the writer task finishes. We therefore use two tasks in the experiment, a periodic task that writes data, and an event-triggered task that reads the data. The event-triggered task is activated by the periodic task immediately after it has written the data. The data shared between the tasks is an array of integers (integer size is 4 bytes), and each task holds a pointer to it. We use bound multiprocessing, i.e., each task is allocated to a particular core and cannot move to a different core during the execution of a particular experiment. In order to reduce jitter, we run the tasks at the highest priority and prevent memory from being paged to the disk.

In the experiment, we measure the time it takes the reader task to read the shared data. Between the different experiment runs, we vary the allocation of the tasks to the cores, the pattern of accessing the data, and the size of the data the tasks share. Regarding the allocation, in the case of intra-core communication both tasks run on core 1, while in the case of inter-core communication the periodic task runs on core 1 and the event-triggered task runs on core 2.

In order to represent different data access patterns, we vary the stride of accessing the shared data. In other words, the tasks access the data array with different increments (see Figure 2 for an example of different strides; the grey elements are accessed, while the white ones are skipped). In the experiment runs, we use the following strides: 1, 2, 3, 4, 8, 12, 16, 24 and 32. The idea behind using different strides is to compare the reading times in the following cases: (i) when the data is read sequentially (stride 1), (ii) when the data is read nonsequentially with an increment smaller than the cache line (stride 2, 3, 4, 8 and 12), and finally (iii) when the data is read nonsequentially with an increment larger than the cache line (stride 16, 24 and 32).

In a particular experiment run, the writer and the reader tasks access the same amount of data and with the same stride. The amount of data shared between the tasks in different runs is the following number of integers: 128, 256, 512, 4 096, 8 192, 16 384, 262 144, 524 288, 1 048 576, 1 310 720. In order

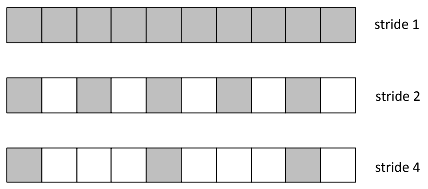


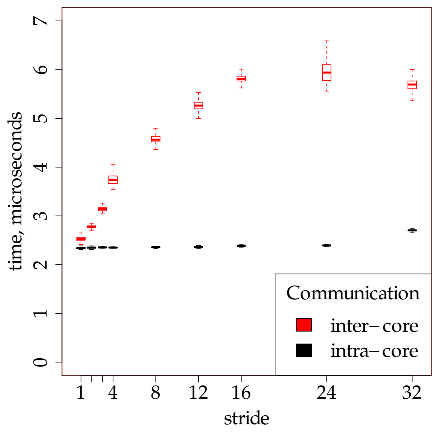
Fig. 2: Stride examples

to access N integers with stride S, we allocate a block of data whose size is N * S * 4 bytes. This means that the data we allocate in the different runs varies from 512 B (128 integers with stride 1) to 160 MB (1 310 720 integers with stride 32), and thus covers data that fits into the L1 cache, data that is too large for the L1 cache but fits into the L2 cache, and finally data that is too large for the L2 cache but fits into the RAM.

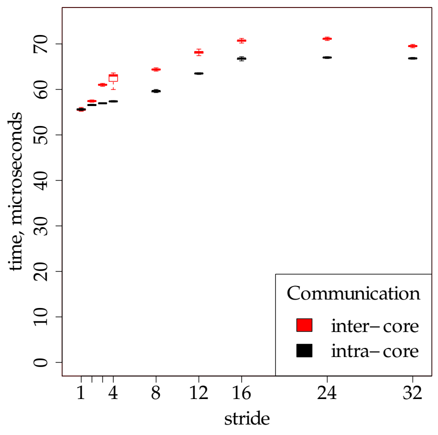
## *B. Experiment results*

We varied 2 allocations, 9 strides and 10 data sizes, which means that 180 experiment runs were performed in total. In each run, we collected 10 000 measurements of the time it took the event-triggered task to read the data sent by the periodic task. The complete experiment results are available in [19]. Here, we illustrate the results by focusing on three representative data sizes: one that fits into L1 cache (256 elements: from 1 kB for stride 1 to 32 kB for stride 32), one that fits into L2 cache (8 192 elements: from 32 kB for stride 1 to 1 MB for stride 32) and one that fits into RAM (1 048 576 elements: from 4 MB for stride 1 to 128 MB for stride 32). In Figure 3, we show the results as three graphs, one for each data size. As the data size increases, so does the reading time, which is the reason for the difference in the time scales between the graphs. Each graph has two entries for every stride: one for intra-core communication (depicted in black) and one for inter-core communication (depicted in red). Each entry is a boxplot describing the 10 000 measurements. The ends of the boxes show the first and third quartiles, the band inside the box is the second quartile (median), while the whiskers extend to the most extreme data point which is no more than 1.5 times the interquartile range away from the box. For the sake of readability of the graphs, the outliers are omitted.

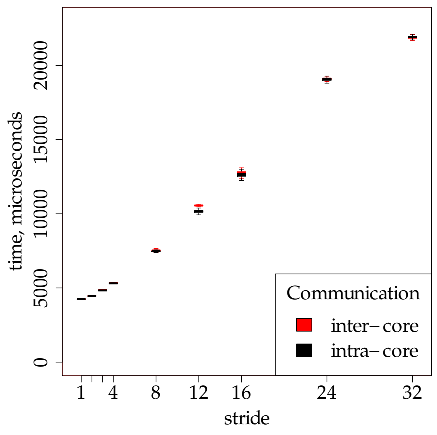
Comparing the three graphs, we can identify a trend of a relative decrease in the difference between intra-core and inter-core communication when increasing the amount of data shared between the tasks. If we take stride 16 as an example, intra-core communication is 144% faster than inter-core communication when the tasks share 256 integers. When the tasks share 8 192 integers, this difference decreases to 6%, and finally when 1 048 576 elements are shared the difference is 1%. As identified in the beginning of the section, this is expected behavior. If the shared data is bigger than the L1 cache, only the end portion of the data will be present in the L1 cache after the writer task has finished writing the data. Since the reader task reads the data from the beginning, it has to be fetched from one of the shared memories (the L2 cache or the RAM, depending on the size of the shared data), regardless of whether the tasks run on the same core or on different cores.



(a) 256 elements



(b) 8 192 elements



(c) 1 048 576 elements

Fig. 3: Experiment results

Looking only at the case where the shared data fits into L1 cache (Figure 3a), we see the expected significant difference between intra-core and inter-core communication. The inter-core communication takes roughly three times as long, which corresponds to the difference in latency between L1 and L2 cache. However, the difference is present only at the higher strides. If the shared data is accessed sequentially (stride 1) there is no significant difference between intra-core and inter-core communication. The reason lies in the way data is transferred between cache and RAM — as mentioned in Section II, this is done at cache line granularity. One cache line of 64 bytes corresponds to 16 integers. So even in the case when the data is not present in the L1 cache, as soon as the reader task reads the first integer from one of the shared memories, one whole cache line is transferred to the L1 cache, containing the currently read element and the 15 subsequent elements. Thus, the next 15 elements will be read from the L1 cache. This continues in the same fashion: after reading one element not present in the L1 cache, the next 15 are read from the L1 cache. In other words, in the case of inter-core communication where we intuitively expected 16 cache misses, we got one cache miss followed by 15 cache hits. Increasing the stride increases the share of the elements that create cache misses and decrease the share creating cache hits. This explains the increase of the times it takes to read the shared data in Figure 3a as we increase the stride. When the stride reaches 16, and thus the difference between two read elements reaches the length of the cache line, then reading every element creates a cache miss. The same happens with the strides higher than 16. Therefore, the reading times stay roughly the same even with further increasing the stride. On the other hand, in the case of intra-core communication, the data being read is always present in the L1 cache, regardless of the stride, and the reading times are roughly the same.

## V. Conclusion

The experiment confirmed that when tasks share data that is bigger than the local cache, we do not see a significant difference between intra-core and inter-core communication time. On the other hand, when the shared data does fit into the local cache, the experiment only partially confirmed the intuitively expected difference in communication times. Inter-core communication took roughly three times as long as intra-core communication (which conforms with the difference between the latencies of the L1 and L2 caches), but only when the shared data was not read sequentially. In the case of sequential data access, the difference between intra-core and inter-core communication was marginal, due to the way data is transferred between the different memories. It can be expected that data would in fact typically be accessed sequentially, meaning that even in the case of data that fits into the local cache, we would not witness a significant difference between intra-core and inter-core communication.

When the tasks do not share a set of data elements, but rather a very small amount of data (for instance only one integer), then inter-core communication would be significantly slower than intra-core communication. However, this would likely not have a large impact on the response time, since the time it takes to access one data element is typically negligible in comparison with the time that a task spends performing calculations.

In summary, we have seen that the difference between intra-core and inter-core communication in most cases is smaller than what could be anticipated from the difference in the latencies of the local and the shared memory. This was shown for the case when the tasks that share data run immediately after each other, which is the most favorable case for exhibiting a significant difference between intra-core and inter-core communication. A typical application would consist of a set of tasks, meaning that tasks that share data would not always run in immediate sequence, and that the difference between intra-core and inter-core communication would be further reduced.

In the context of design-time architecture-level analysis of multicore embedded systems, this has the following consequences. In order to identify whether a particular case exhibits a significant difference between intra-core and inter-core communication, we need detailed information about data access patterns. This information is typically not available prior to the implementation, when we envision the analysis to be performed. However, as seen from the experiments, in the typical case the difference between intra-core and inter-core communication is not significant enough to hinder performing early performance predictions. Early analysis relies on a set of abstractions and estimates, and for a sufficiently precise performance prediction, a small difference in a particular input to the analysis (in this case the difference between intra-core and inter-core communication time) can normally be ignored.

In the future, we plan to investigate other types of task communication, e.g., message passing. Furthermore, we want to perform a similar experiment on distributed embedded systems, consisting of several interconnected multicore units. Here, the difference between local (intra-node) and global (inter-node) communication should be significant, as intra-node communication uses shared memory, while inter-node communication is preformed over the network.

## Acknowledgment

This work was supported by the Swedish Foundation for Strategic Research via the Ralf3 project, and by the Swedish Research Council project CONTESSE (2010-4276). We would like to thank Tiberiu Seceleanu and Aneta Vulgarakis (ABB Corporate Research, Västerås, Sweden), and Michael Wahler (ABB Corporate Research, Baden-Dättwil, Switzerland) for valuable input and discussions.

## References

[1] T. Ulversoy, "Software defined radio: Challenges and opportunities," *Communications Surveys Tutorials, IEEE*, vol. 12, no. 4, pp. 531–550, 2010.

[2] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *Workshop on the Future of Software Engineering*, 2007, pp. 171–187.

[3] J. Feljan, J. Carlson, and T. Seceleanu, "Towards a model-based approach for allocating tasks to multicore processors," in *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2012, pp. 117–124.

[4] Cortex-A15 MPCore, http://www.arm.com/products/processors/cortex-a/cortex-a15.php, [Accessed: 2013-08-20].

[5] Qualcomm Krait 400, http://www.qualcomm.com/snapdragon/processors/800, [Accessed: 2013-08-20].

[6] D. Levinthal, "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors", http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, [Accessed: 2013-08-20].

[7] U. Drepper, "What every programmer should know about memory", http://lwn.net/Articles/250967, [Accessed: 2013-08-20].

[8] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković, "A component model for control-intensive distributed embedded systems," in *Proceedings of the 11th International Symposium on Component-Based Software Engineering (CBSE)*, 2008, pp. 310–317.

[9] E. Bondarev, "Design-time performance analysis of component-based real-time systems," Ph.D. dissertation, Eindhoven Universty of Technology, 2009.

[10] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "ArcheOpterix: An extendable tool for architecture optimization of AADL models," in *ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2009, pp. 61–71.

[11] SAE standard, no. AS5506, "Architecture Analysis & Design Language (AADL)," 2012.

[12] I. Meedeniya, A. Aleti, I. Avazpour, and A. Amin, "Robust ArcheOpterix: Architecture optimization of embedded systems under uncertainty," in *2012 2nd International Workshop on Software Engineering for Embedded Systems (SEES)*, 2012.

[13] Mathworks Simulink, http://www.mathworks.se/products/simulink/, [Accessed: 2013-08-20].

[14] "Performance evaluation of component-based software systems: A survey," *Performance Evaluation, Special Issue on Software and Performance*, vol. 67, no. 8, pp. 634–658.

[15] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.

[16] Intel Core 2 Duo E6700 processor, http://ark.intel.com/products/27251/Intel-Core2-Duo-Processor-E6700-4M-Cache-2_66-GHz-1066-MHz-FSB, [Accessed: 2013-08-20].

[17] PREEMPT RT patch, https://rt.wiki.kernel.org/index.php/Main_Page, [Accessed: 2013-08-20].

[18] POSIX, http://pubs.opengroup.org/onlinepubs/9699919799, [Accessed: 2013-08-20].

[19] J. Feljan and J. Carlson: "The Impact of Intra-core and Inter-core Task Communication on Architectural Analysis of Multicore Embedded Systems — Experiment Results", http://www.idt.mdh.se/~jcn01/research/multicore, [Accessed: 2013-08-20].