IPSJ SIG Technical Report Vol.2018-ARC-232 No.8
2018/7/30

Automatic Parallelization of a Diesel Engine Control software on an Infineon Multicore Processor

ISMAIL NAIT ABDELLAH OUALI[†1] CHRISTOPH SCHUMACHER[†2]
HIROKI MIKAMI[†3] SEBASTIAN KEHR[†6] BERT BÖDDEKER[†7]
KEIJI KIMURA[†4] HIRONORI KASAHARA[†5]

Abstract: This paper presents the compiler-based automatic parallelization of an engine management system executed by an AUTOSAR OS on the Infineon AURIX embedded multicore processor. The execution performance improved up to 1.8 times on two cores with automatic parallelization. By careful memory optimization, the execution time on two cores is up to 8 times short compared to the original execution on one core.

Keywords: MCU, Automatic Parallelizing Compiler, AUTOSAR, Infineon

1. Introduction

Due to the imminent end of Moore's law, processor manufacturers started developing general-purpose multicore architectures. However, in the domain of embedded systems, the shift to Multicore Control Units (MCUs) started a decade later: one reason that made the migration to embedded multi-core architectures necessary is the increase in software complexity. For example, by parallelizing applications, the surplus of computational power available to MCUs makes it possible to execute more complex algorithms or save energy by running the application on multiple cores with a reduced clock rate.

In the automotive industry, the migration of sequential applications to multi-core platforms is challenging [1, 2] as the software is designed and optimized for single-core execution. In this context, this article investigates the automatic parallelization of a sequential automotive application running on an AUTomotive Open System ARchitecture (AUTOSAR) [4] compliant OS.

Automotive control software is complex and contains several highly connected software components. The complicated data dependency in the software makes the manual migration to parallelized code complex and an error prone task. To simplify this process and generate reliable parallel software, the role of automatic parallelizing tools is crucial. In this research, we use an automatically parallelizing compiler based on technology of the Optimally Scheduled Advanced Multiprocessor (OSCAR) compiler [3], to automatically generate parallel code from a sequential Engine Management System (EMS) executed by an AUTOASR OS. The performance evaluation of the resulting software is conducted using an Infineon AURIX Tri-Core board. An AUTOSAR application is structured into Software Components (SW-C). Each SW-C is composed of a set of functions called *Runnables*, communicating with each other. These Runnables are executed periodically or triggered by an interrupt. Runnables with the same release period are grouped into tasks that will be scheduled by the OS.

Maintaining the original application configuration while migrating an AUTOSAR application to multicore Electronic Control Unit (ECU) is challenging. Re-arranging Runnables or changing the task scheduling will impose testing and validating the functional correctness of the application. To avoid such a tedious manual effort, a parallelization approach that will not change the application scheduling is required. To this end, an automatic Runnable-level parallelization approach is used. The Runnables inside a task are distributed automatically to cores while the original tasks scheduling is maintained.

In this paper we evaluate an EMS using OSCAR compiler on an Infineon Tri-Core board. To reduce memory interference, an automatic data mapping approach is presented as well as other optimizations for a better exploitation of the MCU. The rest of the article is structured as follows. Section 2 presents related works. A description of the characteristics of the application and hardware used for evaluation is presented in section 3. Section 4 describes additional optimizations to increase performance. In section 5, the approach used for mapping data is discussed. Extracting more parallelism using inline expansion is introduced in 6 and section 7 summarizes the evaluation results.

2. Related work

Using OSCAR, the automatic multigrain parallelizing compiler, Umeda et al. [5] parallelized a real automotive engine control software. Exploiting coarse-grain parallelism, the program is first decomposed into blocks, which are then analyzed for their data and control dependencies. A number of advanced restructuring techniques such as conditional branch duplication were introduced. Cordes et al. [6] presented a similar approach for the automatic parallelization of embedded software using an integer linear programming solver to parallelize the hierarchical task graph generated from sequential tasks.

Another approach particular to AUTOSAR applications is presented in [7] to migrate to multi-core systems. The introduced algorithm, called *RunPar,* considers Runnables as the scheduling unit. Still tasks are used to implement the schedule. An evaluation

†1, †2, †3, †4, †5 Waseda University.
†6, †7 DENSO AUTOMOTIVE Deutschland GmbH.

using an automotive EMS showed a 1.35 times shorter execution time on average for dual-core execution. As the inherent parallelism of functions used in today's engine management systems is estimated to be in the range of 60 to 70 percent [8], new approaches to extract more parallelism are needed. One approach is the *Supertask* approach proposed by Kehr et al. [9] to increase Runnable-level parallelism.

3. Evaluation Environment

This section describes the characteristics of the EMS application used to evaluate the effect of the automatic parallelization technology.

3.1 Diesel EMS Application

We selected a diesel EMS realized using AUTOSAR as use case. The large amount of communication makes the EMS an ideal use case for parallelization. The application contains 12 independent tasks with different release periods, eleven are periodic tasks with periods ranging from one millisecond to one second, plus one sporadic task (crank-angle). The examined EMS comprises approximately 700 Runnables that implement the behavior of numerous SW-C. The Runnables exchange data via send-receive (SR) and Inter Runnable Variable (IRV) communication. The internal state of the SW-C is updated at different rates, e.g. sensor values are polled with a greater or equal frequency than they are processed. Therefore, Runnables with the same released period are mapped to the same task.

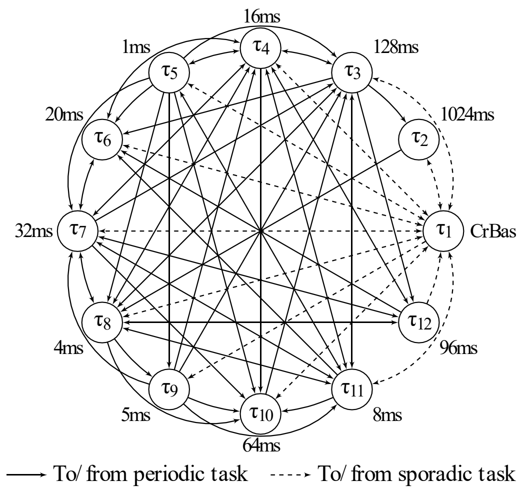

Figure 1: Communication in a diesel EMS.

The tasks communicate frequently with each other; Figure 1 provides a simplified description of the communication between the tasks of the examined diesel EMS. τ_1 executes after an interrupt from the camshaft sensor (crank-angle task). The tasks τ_2 to τ_{12} execute with the period denoted by the label close to the node, e.g. task τ_5 has a period of 1ms. An arrow represents communication between the attached tasks, which is imposed by the Runnables mapped to this task. Thus, communication takes place with different frequencies, but with a repetitive pattern.

3.2 Hardwar Platform

We employ the Infineon AURIX TC277 [10] with three cores as platform for our evaluation; this is a typical representative for an automotive embedded multicore processor. This microcontroller is fabricated in a 65nm technology and the maximum clock rate is 200MHz.

The AURIX microcontroller runs the EMS application, which is parallelized to run on two of its three cores. An AUTOSAR stack hosts the application. Additionally, one of the cores runs an Ethernet stack, which is used to exchange data with a host PC. The data exchange is carried out to monitor the behavior of the application, e.g. to check if all parameters are calculated correctly.

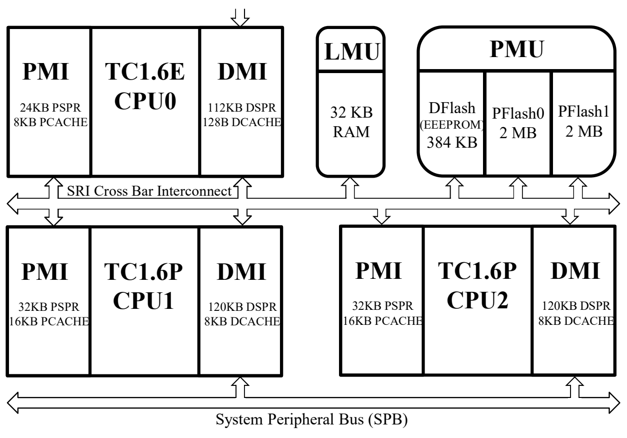

Figure 2: Infineon TC277 memory architecture [10]

The AURIX TC277 has three cores: one power efficient scalar Tri-Core CPU (TC1.66E) and two 32-bit super-scalar Tri-Core CPUs (TC1.6P) with a fully pipelined Floating point unit (FPU). Each core has closely coupled memory areas: Program Scratch-Pad RAM (PSPR) up to 32 KB, and up to 120KB of Data Scratch-Pad RAM (DSPR). Besides, a shared Local Memory Unit (LMU) of 32KB is available. The Program Memory Unit contains 4 Mbyte Program Flash Memory (PFLASH) distributed over two 2MB banks. Data read and write accesses to own DSPR/PSPR introduce zero clock cycles latency. But data read to remote DSPR/PSPR produces 5 clock cycles latency. PMU accesses produces larger latencies [10].

3.3 OSCAR compiler

OSCAR Compiler is an automatically parallelizing compiler technology that converts a sequential program written in parallelizable C-code [11] into a parallel program. To exploit parallelism, the compiler firstly segments a sequential source program into macro-tasks (MTs): basic blocks (BB), loops (RB), and function calls or subroutine calls (SB).

After that, the compiler analyzes the parallelism between MTs and generates a Macro Task Graph (MTG), which expresses the dependencies between MTs. Finally, the MTs are scheduled to the available processors according to this graph.

To acquire accurate estimations of the MT execution costs, OSCAR offers functionality to generate profiling instrumentation measuring the processor cycles consumed by each MT. This data can then be fed back to the compiler to generate a further optimized schedule for the target architecture.

4. Employed Improvement Techniques

Several improvements beyond parallelization were required to enable speed-ups close to the timing predictions by the OSCAR scheduler.

The following subsections introduce selected representative optimizations that turned out to be crucial to achieve viable performance: Inevitably, parallelization introduces overhead due to synchronization code and resource contention.

4.1 Enabling instruction caching

It is possible to run the code with and without cache. In the case of the Infineon board, program cache was not active despite of activation using software instructions.

The Infineon board offers different memory areas regulating accesses to the same memory locations. In the above case, the program flash memory was mapped to segment 10 which is a non-cached segment [10]. Accordingly, the issue was remedied by using a cacheable segment to access the program flash memory.

4.2 Context Save Areas mapping

The AURIX does not have hardware registers to store the stack. It uses context save areas (CSAs) to save context [10]. Context switching occurs when an event or an instruction causes a break in program execution. The CPU then needs to resolve this event before continuing with the program. For example, events and instructions which may cause a break in program execution could be interrupts or service requests, traps, or function calls.

Maximum performance is achieved when CSAs are placed into the data scratch-pad memories of their respective cores.

In the default configuration, CSA regions for all 3 cores were mapped to the scratch area of core 0, which caused a severe congestion in memory, leading to a significant performance loss during the parallel execution of the application.

Mapping the CSA of each core to its local memory reduced cross-core interference. This resulted in a performance increase of 1.84x.

4.3 Reducing Flash Memory interference

The TC277 has 4 Mbyte of PFlash memory divided into two 2 MB banks. By default, program code (.text section) and read-only data (.rodata section) are mapped to the same PF bank. Even with PCache activated, concurrent access to PF memory can cause a noticeable performance loss.

One way to improve performance is to separate code from read-only data, by placing each one into different banks. Since the parallelizing compiler can generate separate code for each core, mapping the generated code separately to different PF banks improves the performance during parallel execution.

4.4 Compile unit optimization

OSCAR compiler generates new code for the parallelized task, including new definitions to all Runnables used in the tasks subjected to parallelization.

By having all the code in one source file, it is likely that the target compiler is enabled to apply more aggressive interprocedural optimizations, resulting in further optimized machine code. On the other hand, the compilation time increases because the generated files are large.

5. Data locality

In most of embedded system architectures, individual cores are attached to local memories. Local memories can be accessed by the CPU with low latency. Thus, correct data placement in memory is crucial to improve performance of the application.

In the sequential setup of the application, all global data (.bss section) are mapped to the local RAM of core 1 (DSPR1). Mapping all data to the same DSPR will lead to memory contention during the parallel execution, since both cores will access the same memory. Additionally, latency overhead due to remote accesses of the second core degrades the performance of the parallel execution. To counter this problem, we use an automated mapping of data to adequate DSPR based on data usage statistics, which is described in the following subsections.

5.1 Data analysis

To be able to map data correctly to DSPR, data access statistics are required. The most straight-forward way is to dynamically trace data accesses to memory during the execution of the application.

The Infineon Multi-Core Debug Solution (MCDS) trace viewer tool [12] is employed for this purpose. The software allows acquiring various traces including data accesses. Although this dynamic approach gives accurate results, it does require an initial execution of the application and taking several traces to get data accesses for all tasks. To simplify this process, a static approach based on OSCAR built-in memory access statistics can be used as alternative.

OSCAR analysis can provide data usage estimates by each core during parallel execution. The generated statistics include information about the variable name, size and read/write count for each core.

5.2 Data mapping

To generate a data mapping, data access statistics by each core are required. The mapping approach is straightforward: Data needs to be placed close to the core accessing it the most. For each variable, a score based on remote data access latencies for each core is calculated. Finally, data is mapped to the core that minimizes the remote access' overhead.

5.3 Implementation

The mapping of data to the appropriate DSPR is done via adding section attribute directives directly to source code. This process is scriptable, and the mapping can be applied automatically. Furthermore, synchronization variables generated by the compiler for the parallel execution are mapped to the core waiting for variable changes.

5.4 Data mapping results

To test the effectiveness of the mapping and confirm the correctness of the static analysis, mapping results for three scenarios are presented (2 cores parallel execution): mapping all data to DSPR1, mapping data based on OSCAR static analysis and mapping data based on accurate execution-time trace data.

The graph below illustrates the results of the mapping of mutable global variables (.bss) in task 16ms:

	Local mem. access	Remote mem. access	Local mem. hit-rate	Mapping correctness	Vars in DSPR1|vars. In DSPR2
DSPR1	1122 [434/688]	2197 [723/1474]	33%	62%	757|0
OSCAR (static)	2974 [1066/1908]	345 [91/254]	89%	97%	464|293
MCDS (dynamic)	3081 [1086/1995]	238 [71/167]	92%	100%	473|284

Table 1: Task 16ms – mutable data mapping (2 cores)

The table summarizes the data access details and the effectiveness of the resulting mapping. Local memory access column represents the number of data accesses [write/read] by cores to their own local DSPR. The hit-rate is the ratio of local accesses by remote accesses. The mapping of data based on execution trace is considered as the reference to compute the mapping correctness (% of data placed to the closest DSPR).

Putting all data in the same location causes many remote accesses degrading the performance of the parallel code. E.g., for the task with a period of 16ms, (wrongly) mapping all accessed mutable data (non-constant global variables) into DSPR1 produces many remote data accesses, leading to more than 35% performance loss during parallel execution.

Based on run-time data access traces (MCDS), the resulting mapping is able to reduce the remote accesses rate to 8%, and the predicted speed-up of 1.81 times is almost achieved (1.82 times is predicted). The mapping based on the OSCAR static memory access analysis produces a mapping leading to performance similar to the mapping based on real execution traces. Even with two independent data sources, still, virtually the same results are produced.

This mapping approach can be applied to read-only data as well. Due to the difference of access latencies between ROM and RAM, mapping read-only data to DSPR improves the performance.

The resulting mapping of data for individual tasks is usually effective since OSCAR tends to generate a schedule that reduces remote accesses to memory. An overall mapping for all tasks is needed and could be generated by combining the accesses data of all tasks together, reducing the remote accesses for the whole application. Since most tasks are periodic and have different periods, coefficients could be introduced to prioritize the reduction of remote accesses for tasks.

6. Inline expansion

Due to the high number of data dependencies, the extracted coarse grain parallelism may lead to moderate parallelism for tasks. Therefore, inline expansion of Runnables might be useful. The goal is to extract more fine-grained parallelism by splitting Runnables into smaller parts. Inline expansion is an improvement technique that replaces a function call with the body of the function called.

OSCAR compiler can apply inline expansion to desired function calls, allowing it to exploit near-fine-grain-parallelism among statements in the body of the subroutine. As AUTOSAR tasks are a set of function (*Runnable*) calls, exploiting the parallelism inside these functions can increase the overall parallelism of the task.

Using directives, the parallelizing compiler can inline function calls, replacing calls with function bodies. It is also possible to recursively inline all or a specified level of function calls inside a subroutine. Considering analysis time and code size, selecting target subroutines for inline expansion is essential; A Selective Inline Expansion approach [13] might be beneficial to simplify the process. Typically, candidate functions for inline expansion are functions that cause bottleneck for parallelization (idle time). The Gantt-chart can be useful to spot such functions.

7. Evaluation results

This section presents the result summary of the parallel execution of the EMS on the Infineon TC277.

7.1 Coarse-grain Parallelism

Figure 3 summarizes the parallelization results of all periodic tasks. The horizontal axis defines periodic tasks, and the vertical axis describes the speed-up values. The speed-up values are computed as follows: Seq_τ / Par_τ, where Seq_τ is the execution time average of task τ executing Runnables sequentially, and Par_τ is the parallel execution time average. The expected speed-up is predicted by the complier assuming no memory interference overhead. The achieved speed-up represents the real performance gain due to parallelization. The impact of individual task parallelization depends on its period and cost. Hence, the measured weighted average speed-up achieved for the whole application is 1.46x.

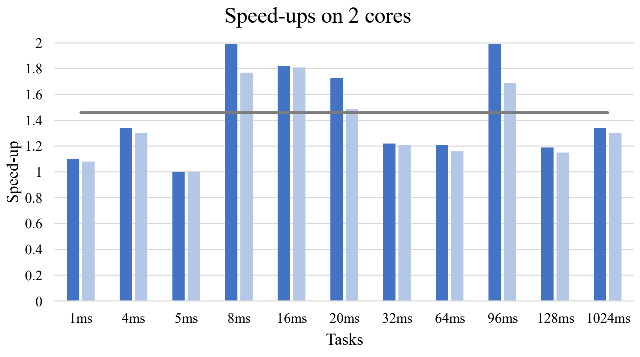

Figure 3: Coarse grain parallelization results on 2 cores

The results reveal that some functions do not have enough inherent coarse-grain parallelism due to data dependency or code size (task 5ms is not parallelized since it contains one Runnable). Reducing memory interference made it possible to achieve expected speed-ups for most of the tasks (overhead impact is large over small tasks).

To illustrate the effect of the improvements described in section 4 and 5, the figure below shows execution costs of the task with 16ms periodicity with different optimization configurations.

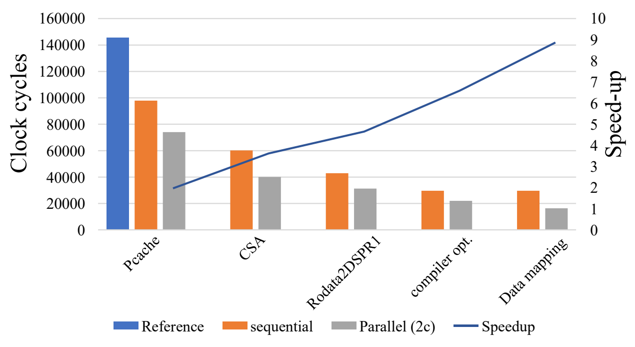

Figure 4: Task 16ms - optimization results

By using the cached PFlash segment and placing CSA to core-local DSPR, the sequential execution become 2.42 times shorter. Placing read-only data to DSPR reduces the read latency overhead improves performance as well. Finally, by mapping all data to adequate DSPR, the parallel execution is shorter and the predicted 1.81 times speed-up is achieved (1.32 at first).

7.2 Inline expansion

Inline expansion results are presented for task 32ms that has scarce parallelism (1.22 times). The poor parallelism of this task is due to high data dependency caused by aggressive usage of the diagnostic subroutine. The recursive inline expansion of all subroutines in the task increased the parallelism of the task to 1.72 times.

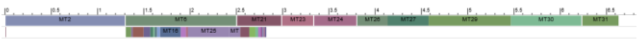

Figure 5: Task 32ms - Gantt-chart (coarse grain)

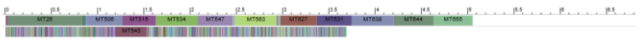

Figure 6: Task 32ms - Gantt-chart (inline expansion)

Figure 5 illustrates to predicted execution Gantt-chart of the two cores coarse grain parallel execution of task 32ms. Due to data dependencies, the second core is idle for most of the time. The new schedule of Task 32ms after the inline expansion is shown in Figure 6. The fine grain MTs are distributed effectively on both cores.

To note, the fine-grain nature of the inlined program allows the compiler to generate a scheduling that reduces remote accesses of data leading to a near perfect mapping of data.

8. Conclusions

This paper describes the automatic parallelization of an automotive Engine Management System using OSCAR Compiler on the Infineon AURIX Tri-Core board. The periodic tasks in the application are parallelized to run on two cores. The data dependency between Runnables limited the extracted parallelism in tasks and the achieved speed-ups ranged from 1.10 times to 1.81 times shorter execution time. Executing the parallel tasks on hardware showed that the overhead due to memory interferences needs to be considered. To avoid memory contention during parallel execution, automatic data mapping of data to local memories is introduced, placing data closer to the core using it the most. Reducing contention in the program flash memory is important as well: The contention is reduced by the separation (duplication) of code used by each of the two cores to different physical PF banks. More improvements related to the Infineon board configuration are explained, including: the usage of cached memory segments of program flash memory and the mapping of Context Save Areas (CSAs) of each core to its own DSPR.
The usage of OSCAR generated code and applying these optimizations resulted in a significant performance gain. In total, the parallel execution of tasks together with memory optimizations (compared to the execution of the original sequential application without optimization) yielded a speed-up up to 8.7 times. These are the results of the memory usage optimization and coarse grain parallelization of tasks (without splitting the Runnables). By the memory optimization, the sequential execution speed was shortened up to 4.9 time and the coarse grain task parallelization gave us up to 1.81 times speed-up using two cores. More parallelism can be extracted by the inline expansion of Runnables. For example, initial experiments show that the inherent parallelism of task 32ms can be improved from 1.22 times to 1.72 times by full inline expansion of Runnables. Further parallelism could be achieved by manual restructuring of the diagnostic routines causing most of the data dependencies within the application.

References

[1] G. Macher, A. Höller, E. Armengaud and C. Kreiner, "Automotive embedded software: Migration challenges to multi-core computing platforms," 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), Cambridge, 2015, pp. 1386-1393.
[2] P. Gai and M. Violante, "Automotive embedded software architecture in the multi-core age," 2016 21th IEEE European Test Symposium (ETS), Amsterdam, 2016, pp. 1-8.
[3] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara, "Oscar api for real-time low-power multicores and its performance on multicores and smp servers," 2010.
[4] AUTOSAR GbR. AUTomotive Open System Architecture (AUTOSAR) Operating System. http://www.autosar.org, February 2013.
[5] Dan Umeda, Yohei Kanehagi, Hiroki Mikami, Akihiro Hayashi, Keiji Kimura, Hironori Kasahara,"Automatic Parallelization of Hand Written Automotive Engine Control Codes Using OSCAR Compiler", CPC2013
[6] D. Cordes, P. Marwedel and A. Mallik, "Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming," 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Scottsdale, AZ, 2010, pp. 267-276.
[7] M. Panić, S. Kehr, E. Quiñones, B. Boddecker, J. Abella and F. J. Cazorla, "RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores," 2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), New Delhi, 2014, pp. 1-10.
[8] Mader, R., Graf, A., and Winkler, G., "AUTOSAR Based Multicore Software Implementation for Powertrain Applications,"

SAE Int. J. Passeng. Cars – Electron. Electr. Syst. 8(2):264-269, 2015, https://doi.org/10.4271/2015-01-0179.

[9] S. Kehr et al., "Supertask: Maximizing runnable-level parallelism in AUTOSAR applications," 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2016, pp. 25-30.

[10] Infineon. AURIX - TC27x D-Step, 32-Bit Single-Chip Microcontroller, User's Manual V2.2 2014-12, https://www.infineon.com/dgdl/Infineon-tc27xD_um_v2.2-UM-v02_02-EN.pdf?fileId=5546d46259d9a4bf015a846b363874d1

[11] M. Mase, Y. Onozaki, K. Kimura and H. Kasahara, Parallelizable C and its performance on low power high performance multicore processors In Proc. of 15th Workshop on Compilers for Parallel Computing, 2010

[12] Multi-Core Debug Solution (MCDS) Trace Viewer. https:// https://www.infineon.com/DAS, (accessed 2018-06-20).

[13] Shirako, J., Nagasawa, K., Ishizaka, K., Obata, M., & Kasahara, H. (2004). Selective inline expansion for improvement of multi grain parallelism. Parallel and Distributed Computing and Networks.

[14] "System and method for selectively enabling load-on-write of dynamic ROM data to RAM", 1999.