Paving the Way for Multi-cores in Industrial Hard Real-time Control Applications

Ralf Jahr, Martin Frieb, Mike Gerdes, Theo Ungerer  
University of Augsburg, Augsburg, Germany  
Email: {jahr,frieb,gerdes,ungerer}@informatik.uni-augsburg.de

Andreas Hugl, Hans Regler  
BAUER Maschinen GmbH, Schrobenhausen, Germany  
Email: {andreas.hugl,hans.regler}@bauer.de

Abstract—The rise of multicore processors for industrial embedded control applications forces companies to face the challenge of replacing legacy single-core applications by multi-threaded programs. We present a systematic and tool-supported approach starting with existing single-core code and transforming it into multi-threaded code such that timing analysis is preserved and eased. The approach is based on (a) scheduling periodic tasks onto multiple dedicated cores as well as (b) executing other code parts after a model-based parallelization, which introduces structured parallelism only, on the remaining cores. The main advantage of our approach compared to a re-implementation is a strongly reduced effort for implementation and testing because of the reuse of existing code. The approach is demonstrated and evaluated for the control code of a foundation crane; slack time is introduced as measurement for the effectiveness.

I. INTRODUCTION

Multi- and many-core processors are more and more available for embedded industrial applications. Recent examples are the Freescale QoriQ P4080 with eight cores, the Infineon Aurix with up to three cores, and TI’s KeyStone II Multicore with up to four ARM cores and eight DSP cores. Gaining higher performance from these additional cores requires parallel software [1]. However, the transition from a sequential program to a parallel one is known to be complex [2].

The focus of this paper is on how to introduce parallelism into an existing real-world industrial control code of a hard real-time embedded system. Our prototypical example is the control code of a foundation crane, which is a large construction machine by BAUER Maschinen GmbH. The main drivers for execution on a multi-core are (a) the need for additional computation power as well as (b) the faster execution of existing algorithms and still (c) high-precision results with low jitter in the update intervals.

As for the single-core source code, a static timing analysis to determine the worst-case execution time (WCET) [3] is planned for the parallel application, too. One limitation introduced by the real-time perspective is the static mapping of threads to cores: There is no dynamic thread scheduling, hence it has to be known a priori which core executes which thread. Moreover, only one thread per core is allowed. Also, the hardware platform must support timing analysis. Hence, a future predictable many-core processor as proposed in the parMERASA project [4] for embedded hard real-time systems is assumed as target platform. Communication between cores is done via shared variables in virtual shared memory [5] and a network on chip (NoC). Barriers are available for coordination and locks for synchronization.

II. CONTROL APPLICATION OF THE CRAWLER CRANE

The BAUER foundation crane MC 128 (engine power 709 kW, max. boom length 54.4 m, weight of the base unit 170 t, max. capacity 200 t) can be used for driven drilling, milling, and soil compaction techniques. It features an embedded real-time-capable electronic control unit (ECU), ESX-3XL [7] by Sensor-Technik Wiedemann (STW) with a single-core Infineon TriCore processor. Functions provided by the software running on this ECU are, e.g., movement and positioning of the machine and its tools as well as assistance for the different operation modes.

The software on the ECU is written in the programming language C and comprises three layers (BIOS with scheduler, middleware, and application program). APIs are defined between the different layers and data is exchanged by shared memory. The main control loop (short: main loop) in the application program is executed concurrently to the scheduler in the BIOS which invokes periodic tasks defined in the middleware. Hence, the scheduler interrupts the main loop from time to time.

Generally speaking, the code is control intensive (in comparison to data intensive); there are no large loops but many conditions on parameters and sensor values.

Gerdes et al. [8] modified the control application of a large drilling machine manufactured by BAUER Maschinen GmbH for a timing-analyzable processor with two or four cores. However, the parallelization was based on knowledge of the application (i.e., domain knowledge), whereas the approach followed in this paper is rather systematic and model-based.

III. DEVELOPING MULTI-CORE SOFTWARE BASED ON SINGLE-CORE CODE

A. Decision Space for the Transition to Multi-core Software

There are several options for porting software onto a multi-core system. All have the same goals, which are (a) running...
software providing the same functionality as the single-core version, and (b) offering further capacity to run more complex programs to provide functionality beyond the current level. Each option has different advantages and disadvantages concerning several objectives, e.g., the effort for implementation and testing as well as timing analyzability, but also maintainability and tool availability, to mention only a few.

Running the existing unmodified single-core source code on one core of the multi-core processor is possible but not of much use: Synchronization is necessary to allow communication with other cores; this requires changes of the existing code. This adds additional complexity which could lead to over-utilization of the single core of the new system by the modified existing single-core software. This can result in violation of deadlines. In addition, the use of shared resources like global memory can slow down the application, too.

Automatic parallelization as suggested by Cordes et al. [9], for example, leads to unstructured parallelism making later timing analysis much harder; structured parallelism as provided by the parallelization presented in the remainder is very beneficial [10], [11].

A re-implementation from scratch probably leads to the best solution in terms of extendibility and is the best basis for future development. Its negative aspects are the high required efforts for development and testing (code tests and machine tests). Companies might not be willing to pay this high price (which might be shortsighted).1

We suggest a combination of model-based structured parallelism and existing source code to yield the potential of multi-core processors: This leads to a reduced effort for implementation (because of code reuse and tool support) and testing (because of code reuse). In addition, timing analysis is supported.

B. Parallelization Concept and its Application

Our parallelization concept is to (a) execute the periodic tasks on dedicated cores (see Section III-B1) and (b) parallelize the main loop and run it on the remaining cores of the target platform2 (see Section III-B2).

Communication is still done via shared memory; locks are employed in all parts of the software to secure critical sections. As discussed in [13], this leads to a timing-analyzable parallel implementation of the control application.

1) Scheduling of Periodic Tasks: In the single-core version of the control code, periodic tasks are executed by a scheduler provided by the ECU supplier. The scheduler interrupts the execution of the main loop and can also interrupt periodic tasks of lower priority. This leads to jitter in the execution times of periodic tasks and the execution time necessary for an iteration of the main loop.

To overcome this and to leverage timing analysis, dedicated cores are reserved for the execution of periodic tasks in a simple static cyclic schedule [14]. With a more sophisticated scheduling algorithm (see, e.g., [15]), periodic tasks could also comprise multi-threaded parallel code.

The analysis of the main control loop of the construction machine showed (see [16]) that several functions are called periodically during its execution. These functions are also treated as periodic tasks and be started by the scheduler. Overall, 23 periodic tasks were found in the software: 8 were previously defined for the scheduler and 15 are carved out from the main loop. Every 3,000 ms, the highest accumulated maximum observed execution time of 0.978 ms is to be completed in a 1 ms interval. With multiple cores, additional synchronization effort is necessary; hence, two dedicated cores are reserved for the periodic tasks.

2) Model-based Parallelization of the Main Control Loop: Parallelism in the main loop of the control program is extracted and optimized according to our pattern-supported parallelization approach. It was introduced in [17] and its applicability for hard real-time embedded systems is discussed in [13]. The approach describes a systematic process starting with a sequential and resulting in a parallel program. Parallel Design Patterns (PDPs) are the only allowed means for introducing parallelism, hence the resulting program features structured parallelism. PDPs are abstract textual concepts describing best-practice solutions for recurring situations of parallelism [18].

The applied parallelization approach is model-based and parts require guidance by an engineer, especially to isolate and describe situations suitable for parallelization. They must match PDPs as described in a Pattern Catalogue; we selected the parMERASA Catalogue [19] with time-predictable design patterns. The Activity and Pattern Diagram (APD) [17] is used for modeling. It is a slightly extended UML2 Activity Diagram: The fork/join operators must not be used; instead, a new node type similar to an activity is introduced representing an instance of a PDP.

The parallelization of a control intensive application like the main control loop is supposed to be hard compared to data-intensive programs. Hence, a high speedup should not be expected. However, a preliminary speedup up to 5.7 for 17 cores was found for the main control loop [16]. This number is based on an approximation only and ignores some complications like synchronization effort for shared global variables; the speedup in reality will be smaller.

IV. SUPPORT FOR THE PARALLELIZATION PROCESS OF THE MAIN CONTROL LOOP

Figure 1. Overview of the model-based parallelization process

For the periodic tasks, the work to do is comparably simple: They can be identified by analyzing the configuration of the existing scheduler (or source code, which is clearly manual.
work). Commercial and open source tools are available for schedulability analysis and to calculate a static cyclic schedule; heuristic algorithms can facilitate this, too.

Tool support is very important for the parallelization of the main control loop: Figure 1 shows the parallelization process according to [13], [17]. The main work effort is caused by code analysis and adaption of the source code according to the optimized model of parallelism:

**Building the Model for Optimization, Adding Parallelism:**
The source code has to be checked to identify situations for PDPs. Even if available, automatic tools should be used carefully: The main difference between automatic parallelization and our approach is that also situations fitting only roughly (and maybe requiring manual adaption of the source code) can be identified by the developer. This analysis step is of little effort once the engineer is familiar with the PDPs provided by the selected pattern catalogue and the structure of the source code. The final model can be represented graphically with a standard UML tool or more structured and textual as XML file.

**Analysis of Accesses to Shared Global Variables:** For each potentially parallel situation, it has to be decided if the code fragments—in the model they map to activities—can actually be executed in parallel without difficulties. This is mainly dependent on the data and sometimes control dependencies. They can be identified in the source code with static analysis: For example, CScope\(^3\) can provide assistance by listing the accessing functions for a global variable. However, a list of variables accessed by a function is not available. A tool for the analysis of such dependencies is also under development by Rapita Systems\(^4\), which will be commercialized in near future. In the end, the found dependencies should be documented in a readable and parsable format (“dependency list”).

**Optimization of the Model:** A multi-objective optimization minimizing both the approximated execution time and the number of necessary threads can be performed with a heuristic algorithm. Our soon to be released optimization tool for this task reads the APD from an XML file and the dependencies from the dependency list, see above.

**Implementation Effort:** The implementation effort, i.e., the amount of work for transition from the optimized model to source code, is mainly defined by the need for (a) implementing the parallel design patterns (PDPs) and (b) securing accesses to global shared variables, data, and devices (resources, in general).

The PDPs can be implemented by algorithmic skeletons [20] or a framework like MTAPI [21]. This simplifies implementation because already tested code is employed. Also, software maintenance is simplified. We are working on a special collection of timing-analyzable algorithmic skeletons (TAS) with special support for timing analysis with OTAWA [22], [11].

To ease the adaption of the source code with respect to synchronizing accesses to shared variables, the use of mutator methods (like `get/set`) is preferred wherever possible. The mutator methods hide the necessary lock and unlock calls. Reading accesses throughout the code base are replaced by the `get` method and writing accesses by the `set` method, respectively. These changes have to be made both in the parallelized main control loop as well as in the periodic tasks. This process can be assisted with a custom code generator as well as state-of-the-art refactoring tools.

**WCET Analysis:** The WCET analysis can be performed after completing the parallel source code. Currently the number of suitable tools capable of analyzing parallel programs is very small. We are aware of progress being made in this direction for OTAWA [11] and RapiTime [23].

V. Slack Time as Performance Indicator and Preliminary Results

In difference to other parallelization projects, speedup is not the suitable performance number for the parallelization of embedded hard real-time systems: The system works well as long as defined deadlines are not missed, hence also with a single core. Instead, we focus on slack time describing the time in which the processor is idle (see hatched area in Figure 2). It is a good measure for the potential to run more complex algorithms or to provide further functionalities; this is the major goal of the transition towards a multi-core.

![Graphical representation for utilization and slack time in a 7 ms interval for one, three, and six cores running the construction machine control code](Figure 2)

In the single-core implementation of the application, about 0.8 ms of a 1 ms interval are needed for periodic tasks and 0.2 ms remain for the main control loop. It takes about 6 ms to complete an iteration of the main loop, which is below the deadline of 7 ms. The slack time is hence 0.2 ms per 7 ms.

With more cores, both the periodic tasks and the main control loop get slowed down by the synchronization and parallelization overheads (represented by `Sync` in Figure 2; a constant factor of 0.25 is assumed). With three cores in total, the slack time is 7 ms - 6 * 0.2 ms * 1.25 = 7 ms - 1.5 ms = 5.5 ms per 7 ms interval.

For example, with six cores, the main control loop can be parallelized and executed on multiple cores with a speedup; let it be 2 as conservative assumption for 4 cores. This leads to a higher slack time either (a) for a sequential code part (7 ms - 1.5 ms / 2 = 6.25 per 7 ms interval), (b) for a parallel code part in the same style as the existing control loop with a speedup.

---

\(^3\)Homepage: http://cscope.sourceforge.net/

\(^4\)Homepage: http://www.rapitasystems.com/
of 2 (6.25 ms * 2 = 12.5 ms per 7 ms interval), or (c) a highly parallel code part efficiently using all four cores available for the main loop (6.25 ms * 4 = 25 ms per 7 ms interval).

VI. CONCLUSION AND OUTLOOK

One possibility to make additional computation power available for embedded hard real-time systems is to migrate existing control code to multi-core processors. To reduce the necessary effort, this paper presents an approach starting with existing legacy code. Periodic tasks are isolated and run on dedicated cores. The remaining sequential code parts can be parallelized with a model-based parallelization approach [17], [13] and be executed on the remaining cores. The approach is applied on a real-world code, the control code of a complex construction machine by BAUER Maschinen GmbH.

With the proposed parallelization techniques, all goals of the migration (see Section I) can be achieved: (a) Additional computation power for more complex algorithms or further functionality is made available. Slack time for the main control loop is highly increased. (b) Existing algorithms can be executed faster if parallelism is added, see [16], (c) Jitter is reduced because periodic tasks can be called with precise timer interrupts and are not interrupted by other tasks.

With only timing-analyzable parallel code structures and synchronization primitives, a static WCET analysis with OTAWA [22], [11] is possible and will be performed as future work. Necessary preparations were already made as mentioned in the paper. Also, the parallelization approach was evaluated for timing analysis already [13]. The timing-analyzable algorithmic skeletons (TAS) will reduce the annotation effort in source code and help to provide the necessary flow facts for a WCET analysis.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA).

REFERENCES