Parallel Computing 81 (2019) 122–130

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Evaluation and modeling of the supercore parallelization pattern in automotive real-time systems

Remko van Wagensveld[a], Tobias Wägemann[b], Ralph Mader[c], Ramin Tavakoli Kolagari[b], Ulrich Margull[a,*]

[a] Technische Hochschule Ingolstadt, Esplanade 10, Ingolstadt, Germany
[b] Technische Hochschule Nürnberg, Keßlerplatz 12, Nuremberg, Germany
[c] Continental Automotive GmbH, Siemensstraße 12, Regensburg, Germany

ARTICLE INFO

Article history:
Received 13 June 2018
Revised 26 October 2018
Accepted 17 December 2018
Available online 18 December 2018

Keywords:
Parallelism
Embedded devices
Real-time systems
Parallel design pattern
Software design
Description languages

ABSTRACT

Today's combustion engines are finely tuned to deliver as much performance as possible out of only little amount of fuel. To achieve such high efficiency a lot of computational power is needed in Engine Management Systems (EMSs), which nowadays is delivered by multicore processors. However, this is a challenge for software developers as most of them are not yet familiar with the specifics of multicore programming. The real-time requirements of an EMSs further complicates software development.

This paper revisits the supercore embedded Parallel Design Pattern, which reduces the fork-join-overhead by ensuring concurrent execution of coupled tasks. To test our pattern we implemented two algorithms using the supercore pattern on a state of the art EMS with an Infineon Aurix TC39x processor. We show that by using the supercore pattern we were able to reduce the response time of the analyzed functions and to achieve a speedup of up to 1.97 on four cores. We also analyzed the effect of the non-uniform memory architecture, which required specific optimization measures and limits the achievable speedup to 2 on four cores.

We also show how the supercore pattern is modeled by previously defined extensions to the **E**lectronics **A**rchitecture and **S**oftware **T**echnology - **A**rchitecture **D**escription **L**anguage (EAST-ADL) and **AUT**omotive **O**pen **S**ystem **AR**chitecture Standard (AUTOSAR).

1. Introduction

The Barr Embedded Systems Survey states that in 2018 68% of all embedded systems have at least two cores. Additionally, over the last 3 years the share of 4 or more cores has risen rapidly whereas the share of 2–3 cores declined [1]. This indicates that a paradigm shift from single to multicore is necessary in the embedded domain.

Continental AG is one of the most successful suppliers in the automotive industry with a focus on premium products, such as Engine Management Systems (EMSs). A typical processor for such a device is the Infineon Tricore Aurix, which has up to six cores and a Non-Uniform Memory Access (NUMA) architecture [2]. Typically, an EMS has real-time requirements

* Corresponding author.
E-mail addresses: remko.vanwagensveld@thi.de (R. van Wagensveld), tobias.waegemann@th-nuernberg.de (T. Wägemann), ralph.mader@continental-corporation.com (R. Mader), ramin.tavakolikolagari@th-nuernberg.de (R. Tavakoli Kolagari), ulrich.margull@thi.de (U. Margull).

https://doi.org/10.1016/j.parco.2018.12.004
0167-8191/

and runs AUTOSAR OS [3]. The number of cores will increase in the future and the current ongoing research about incorporating coprocessors like Graphics Processing Units (GPUs) [4] will only further increase the need for parallelism.

While many programming languages incorporate concepts for parallelism in their core language structure, e.g. RUST or C++17 [5], the dominant language in the embedded domain still is C [1]. Continentals EMSs are also written in C-99, which increases the entry-barrier of utilizing parallel structures in their software. With the popularity of C and with it the C specific knowledge of developers in the embedded domain, another way of describing parallel solutions is needed.

In Section 3 of this paper the embedded Parallel Design Patterns (ePDPs) are described. An ePDP describes the solution to a recurring parallelization problem in a structured manner. The supercore pattern is presented in Section 4 as an example for an ePDP. It is an architectural pattern designed to lower the overhead that is introduced by the forking and joining of parallel tasks. We evaluate this pattern in Section 5 by parallelizing two algorithms, a curve-sketching algorithm and a finite element model algorithm. The way to model this pattern in the Automotive Open System Architecture (AUTOSAR) [3] and in the Architectural Description Language (EAST-ADL) [6] is shortly described in Section 6. We conclude this paper with a summary and an outlook on future work in Section 7.

2. Related work

There has already been conducted a lot of research in the field of parallel design patterns, for current general purpose computers as well as parallel design patterns for embedded multicore processors.

For the research in current general purpose computers, the pattern approach for parallelization of programs started in 1995, when Foster introduced the PCAM-Method [7]. It segments the problem of parallelization into four steps: partition, communicate, agglomerate, and map. In the partition step the problem is split up in the finest-grained parallel version, where everything that can be done in parallel is determined. To find a practical solution, the communication and data dependencies between the parallel processes are defined. The agglomerate step combines highly coupled sections to bigger chunks which can then in the last step be mapped to an available core. While this approach can be very useful for single algorithms, it can be very cumbersome if a complex legacy system needs to be parallelized. We present embedded Parallel Design Patterns (ePDP) to solve reoccurring problems while parallelizing legacy software.

Parallel design patterns have been presented in 1999 [8] and became well-known in the research community with the release of “Patterns for parallel programming” [9]. Additionally, they presented the “Pattern Language for Parallel Programming” (PLPP) [9], a way to structure the definition of a parallelization pattern. Keutzer, and Mattson formalized the PLPP language and presented “Our Pattern Language” (OPL) [10]. While being very thoroughly designed and thought out, both PLPP and OPL are not suitable for describing real-time constraints. Our research extends these patterns to embedded devices with real-time requirements.

In the embedded domain exists the parMERASA project, that created the OPL-based parMERASA Pattern Language [11]. It focuses on tool-based Worst Case Execution Time (WCET) analyzability, especially with the Open Tool for Adaptive WCET Analysis (OTAWA)[1]. They introduced an annotation format for OTAWA to describe their patterns [12]. The parMERASA project used a custom multicore architecture that was created in the proceeding project “MERASA” [13]. While their results are very promising, they make use of their specialized MERASA architecture which is currently not available as real hardware and thus is not in use. We focus our research on heterogeneous processing architectures that are in use in the automotive industry.

Delange and Feiler [14] present an approach to model partitioned multicore architectures with AADL. The AADL language [15] is a modeling language standardized by the Society of Automotive Engineers (SAE) however its main current application domains are avionics and aerospace. Delange and Feiler conducted a case study where two multicore patterns are presented which are consistent with the standard and compatible with existing single-core patterns. The case study shows that the AADL language is in principle applicable in a multicore environment, but modeling heterogeneous multi- and manycore systems is not systematically investigated.

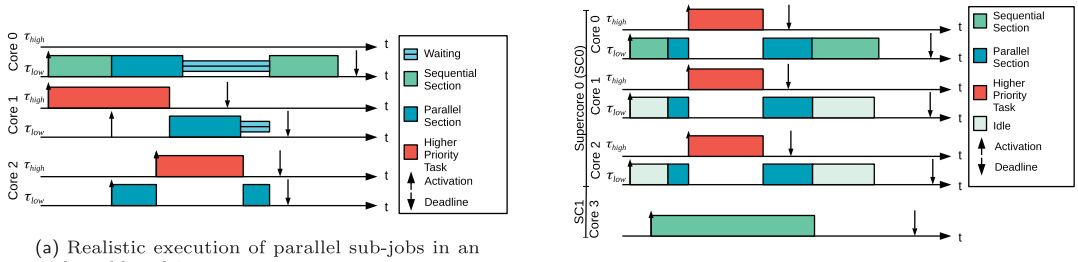
Rodrigues et al. [16] present an approach which describes the application of the MARTE standard to model a GPU architecture. MARTE [17] is a specification of a UML profile with additional capabilities for model-driven development of real-time and embedded systems. The approach presented by Rodrigues et al. shows a model of a GPU without investigating specific manycore aspects and without application of timing modelling aspects.

The AMALTHEA project [18] was focused on the development process of embedded real-time systems with multicore architectures. It defined a meta-model that is used as the basis of an Eclipse-based open-source tool environment.

3. Embedded parallel design pattern

In the domain of software architecture, patterns are widely used [19,20]. The principle of a pattern is to give a solution to a recurring problem when developing software. Our embedded Parallel Design Patterns (ePDPs) are patterns that give solutions to common problems encountered when trying to parallelize software for embedded real-time devices. These patterns need to be identified when parallelizing legacy code for embedded devices by finding similarities between these

[1] www.otawa.fr.



(a) Realistic execution of parallel sub-jobs in an real-world realtime system.

(b) Execution of parallel sub-jobs with the supercore pattern.

Fig. 1. Comparison of executing parallel sub-jobs without and with the supercore pattern.

parallel solutions. They can also be derived from existing patterns from the High Performance Computing (HPC) Domain, e.g. the patterns from Mattson et al. [8,9].

Our meta pattern is based on the OPL [10], but re-introduces the *Implementation Hints* and *Consequences* from [19], to be suitable for the domain of real-time systems. The Implementation Hints include information relevant to the developers implementing the pattern, e.g. which synchronization mechanisms to use or how to manage the task creation. So, an ePDP is a description of a problem, its classification and a detailed description of the problem solution. The implementation hints and the consequences section contain important information for successfully applying the pattern in an embedded context [21].

4. Supercore pattern

First, the terminology will be defined and after the fork-join pattern the supercore pattern will be summarized.

4.1. Terminology

Task A schedulable entity of the operating system; contains one or more sub-tasks
Sub-Task A sub-task is a small part of a task that can be allocated to another core
Response Time The time from activation of a task instance until its completion
Execution Time Net execution time of a job without preemptions.

4.2. Fork-join pattern

The fork-join pattern [22] is an implementation strategy pattern and can be used if a calculation can be split into multiple different computational paths. To achieve this, the sequential control flow is *forked* into multiple parallel running control flows, executing the calculation in parallel. After the calculation is finished, the parallel control flows converge by *joining* them into the main sequential control flow. A join acts as a barrier for the sub-tasks. If the forked control flows are executed on different cores they can run in parallel.

4.3. Supercore pattern

To be able to map the fork-join pattern efficiently, we developed the supercore pattern [21], where a set of cores is combined into one *CoreCluster* with a homogeneous scheduling strategy for all cores. This allows to run a set of distributed tasks on the CoreCluster, similar to Gang Scheduling [23], and thus reduce response time jitter and sporadic delays.

In Fig. 1(a) the execution of a task with parallel sub-jobs in a real-world system without a supercore implementation is shown. Because tasks with a higher priority preempt the parallel sub-jobs there will be relative high waiting times when the tasks are joining for a sequential section.

In Fig. 1 the execution with a supercore implementation is shown. In this implementation cores 0, 1, and 2 are bundled into a virtual supercore SC0. When a task is currently running on a supercore and a higher priority task is scheduled, it will preempt the whole supercore, therefore all real cores of that supercore. After the higher priority task is finished all cores of the supercore resume with their previous task and because the tasks are running at the same time, the synchronization overhead can be minimized. When a task is allocated to a supercore and is currently executing a sequential section, there are two strategies possible to not waste the core-time on the now idle cores. First, one can busy-wait for the parallel section to start and thus have a very low forking overhead, which is shown in Fig. 1(b). Alternatively, one can execute non-critical background tasks in idle time.

5. Case-study

The behavior of the supercore pattern can be evaluated as presented in [21]. There, we simulated different supercore implementations, and performed first evaluations. In this section, we review our previous findings by executing a typical work-load on a real-world EMS.

5.1. Emulating the supercore

Currently, the automotive industry uses AUTOSAR OS which is based on the OSEK scheduler. The OSEK standard defines four types of scheduling policies

i Fully preemptive scheduling,
ii Non-preemptive scheduling,
iii Mixed preemptive scheduling, and
iv Groups of tasks scheduling [24].

While it is not intended to add custom policies, it is possible to implement a user-space scheduler, e.g. an EDF scheduler [25].

For a realistic scenario, the behavior of a supercore must be mapped to the available OS and scheduling policies. This is achieved with one *master task* and several *slave tasks*, where the master task activates the slave tasks at the beginning of its execution. Each task is mapped to different cores, and all together execute one functionality. With a high enough priority, the running tasks will preempt immediately the currently running task and then run nearly synchronously on all cores.

5.2. Setup

The measurements were conducted at Continental Automotive GmbH on a state of the art EMS equipped with an Infineon Aurix TriCore TC39x processor. This processor consists of six cores of the TriCore architecture, but only four cores were active, with the remaining cores being switched off. All cores are clocked at 300 MHz. The TC39x has a total SRAM memory of 6MB, organized in a NUMA architecture: the memory is split and each part is attributed to one core as local memory. While a core still can access all memory, the access from a core to its local memory is faster than access to non-local memory [26]. As we will see later, this effect has a significant influence on the supercore pattern and requires careful partitioning of the data that is processed in parallel fashion.

The EMS is connected to a measurement computer running GLIWA Timing Suite T1[2], a tool to trace the execution of tasks and collect different timing statistics of those tasks. For example it is possible to measure the response time, net execution time, number of preemptions, etc. with GLIWA T1.

5.3. Methodology

To further verify the supercore behavior from [21], two real-world algorithms were parallelized using the supercore pattern, which are a curve-sketching (CS) and a finite element model (FEM) algorithm, see 5.3.2 and 5.3.1.

The algorithms are executed in parallel on one to four cores with one core being the master core. As found in the previous work, priorities on a supercore running on a fixed-priority operating system must be high, otherwise synchronicity between cores is destroyed. Therefore, the CS algorithms' tasks have the highest priority on each core (except interrupt routines). The FEM has a larger relative deadline, and its tasks have the third highest priority, only preemptable by a 1ms-system tick task and necessary interrupts.

Initially, both algorithms had all data located in the memory of core A0. Because of the NUMA architecture, core A0 accesses this data much faster than the other cores. Therefore, on two or more cores, we have two scenarios: (a) evenly splitting the data to cores, such that each core calculates the same amount of data, or (b) an uneven partitioning of the data, such that all cores require the same amount of time for their calculation. Scenario (b) has then a shorter overall response time than (a). It would be possible to distribute the data on each core of the NUMA node, however, what needs to be processed in reality is sensor data, which is read by one core and saved into its local memory. A distribution of the data to the different cores results again in a lot of remote accesses which can obliterate the optimizations.

5.3.1. Finite element model algorithm

The Finite Element Model (FEM) algorithm executes an finite element simulation of an 1-dimensional gas-flow system consisting of different connected pipes. Every pipe can be calculated in parallel and the calculation of the connections can also be calculated in parallel. The FEM algorithm is a longer running algorithm with a higher mix of sequential and parallel sections in the code. The sequential sections are executed on the master core A0, while the other cores wait on the synchronization point indicating the parallel section.

In Fig. 2 the timing diagram of the FEM algorithm is shown including its synchronizations. In total 6 synchronizations are necessary to run the algorithm in parallel. Again, a scenario without (a) and with optimized data-locality (b) is presented.

[2] http://www.gliwa.com/.

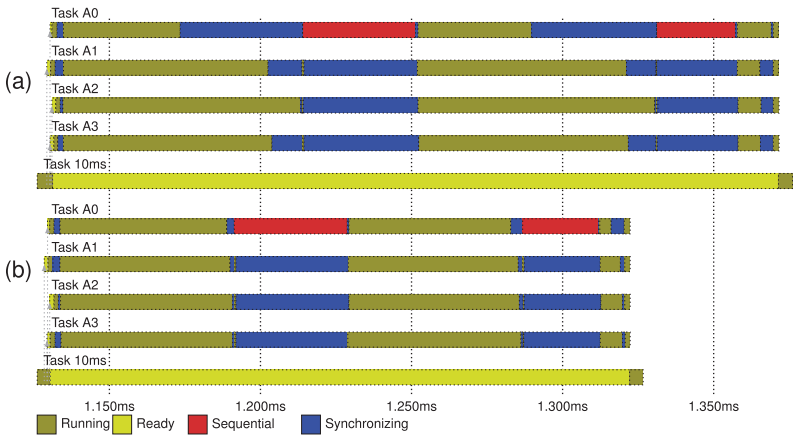
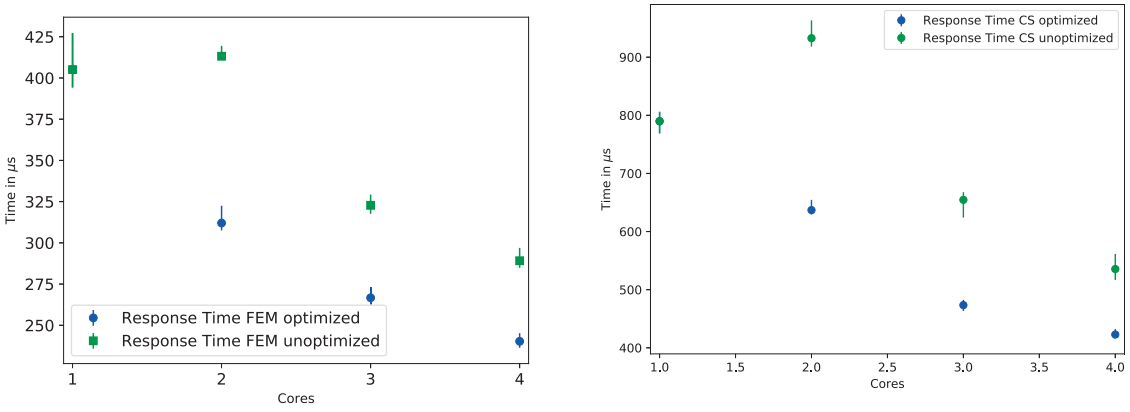


Fig. 2. Traces of the FEM algorithm without (a) and with optimizations (b).



(a) Response Times of the FEM algorithm.

(b) Response Times of the CS algorithm (7500 data points).

Fig. 3. Results of the measurements with one to four cores. The error bars are the minimum and maximum of the measured values. For one core only one data point exists.

5.3.2. Curve-Sketching algorithm

The Curve-Sketching (CS) algorithm calculates the numerical second derivative of a curve and then integrates between its roots to find the largest area under the curve which marks a characteristic turning point. Both the derivative and the determination of the roots can be calculated in parallel. The CS algorithm requires two synchronizations, with a short sequential part between them and a final sequential calculation afterwards. The runtime in the original configuration with 75 data points for the curve is rather short (about 21 µs), but can be expanded up to 7500 data points (about 800 µs).

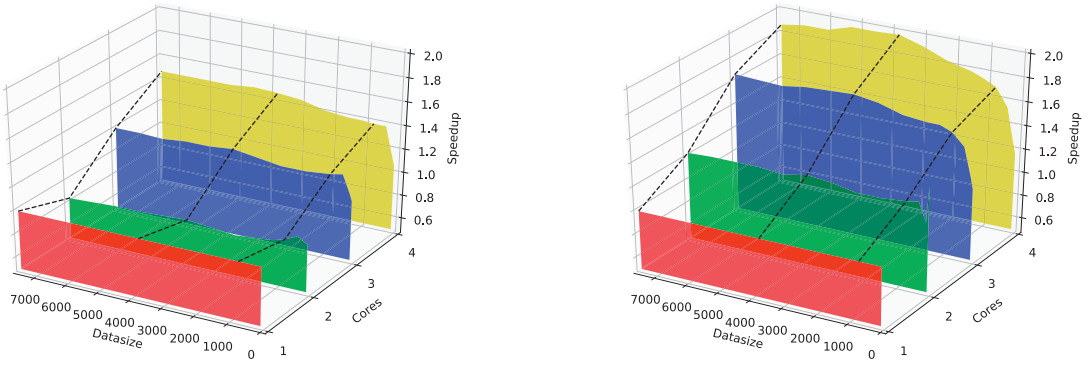
In our measurements core A0 runs the master task and A1, A2 and A3 the slave tasks. In order to compensate the slower memory access from the slave tasks, we additionally implemented an optimized version where the master task calculates a bigger share of the data than the slave tasks. All tasks have the highest priority as required by the supercore pattern.

5.4. Results

Fig. 3 shows the response times of different algorithms for execution on 1, 2, 3, and 4 cores, both with even and optimized split.

For both algorithms, an even data split (unoptimized, i.e. each core calculates the same amount of data) leads to a longer overall response time when using 2 cores, and only a small improvement for 3 or 4 cores. This is a consequence of the non-uniform memory architecture. Since all data is stored local on core A0, the other cores have a much slower memory access, which prevents a significant speedup. This can also be seen in Fig. 2(a), where task A0 spends a significant amount of time waiting for the slave tasks.

The situation is much better when the data split is optimized, such that each core needs about the same processing time. This is shown in Fig. 2(b), where task A0 calculates about 57 % more as each of the slave tasks, thus compensating for the slower access time of the latter. In this case, a speedup of 1.7 (FEM) respective 1.97 (CS) can be achieved when using



(a) Speedup for the CS algorithm with even data distribution.

(b) Speedup for the CS algorithm with optimal data distribution.

Fig. 4. Comparison of the speedup per core and datasize for the CS algorithm.

4 cores. Note that the speedup is limited both by the sequential part of the calculation as well as the slowed memory access of the slave tasks.

In addition, we analyzed how the possible speedup depends on the data size. We increased the data size from the 75 data points (original size) up to 7500 data points. The speedup is calculated as the quotient of the overall response time t_{par} divided by the serial response time t_{ser}, i.e. the response time when running on one core only. The measurement results are shown in Fig. 4 with Fig. 4(a) depicting the even data distribution and Fig. 4(b) showing the optimal data distribution.

For our example setup, we can derive a kind of parallelization threshold for the runtime. Running the CS algorithm with 750 data points and optimized data partitioning yields a the speedup of 1.23, 1.55, and 1.69 for 2, 3, and 4 cores respectively. In the single core case, the CS calculation with 750 data points takes around 100 µs, and with 3750 data points around 400 µs. In this case, a maximum speedup is achieved of around 1.97 on four cores. Note that due to the uneven data split, the master calculates about 49 % of all data, and each slave about 17 %, which gives a theoretical maximum speedup of ≈ 2.

So, on the investigated hardware platform, parallelization of algorithms is only helpful if the runtime of the calculation is at least 100 µs and is optimal for runtimes of 400 µs or more. The FEM algorithm executes in a little bit over 400 µs with one core, therefore the algorithm is well suited for parallelization on an TriCore TC39x. Depending on the data consumption pattern of the algorithm, a speedup of 2 can be reached at most, when the data is located local on one core.

These findings confirm our simulation results shown in [21]. The speedup is less than expected, but since the algorithms are chosen from real-world examples they have more barriers as well as additional sequential sections that haven't been modeled in the simulation.

Improvement of the response time with parallel execution is only possible if a high priority of the supercore tasks has been chosen. When using a supercore with many tasks it is necessary to develop a separate supercore implementation, which takes care of task creation and triggering of the tasks.

6. Modeling the supercore pattern

This section presents an automotive-specific modeling approach of the supercore pattern based on the automotive standards EAST-ADL[3] and AUTOSAR[4]. In the FORMUS^{3}IC project, extensions and advancements have been proposed for both languages in order to address problems when modeling heterogeneous multi- and manycore systems. We give an introduction to the proposed metamodel extensions to address the modeling of the supercore pattern in particular and showcase the application of the concept at the modeling level.

6.1. The supercore pattern in EAST-ADL

The EAST-ADL (Electronics Architecture and Software Technology - Architecture Description Language) in its current version offers very limited support for modeling heterogeneous multi- and manycore systems. During our investigation of the language as part of our work in the FORMUS^{3}IC research project, we demonstrated that support for supercore architectures is part of the overarching issue of general multicore function allocation. The language specification [27] does currently not

[3] www.east-adl.info.
[4] www.autosar.org.

offer a useful way for allocating (software) functionality to multiple allocation targets, as the EAST-ADL element *FunctionAllocation* references both the allocatable element and the allocation target with a multiplicity of 1. Using multiple FunctionAllocations for representing a multicore allocation would therefore describe a redundant allocation of one runnable on each allocation target (as opposed to a parallel allocation of the runnable on all allocation targets). Our solution is the introduction of three new language elements: *ExecutionDomain, CoreCluster* and *FunctionConcurrency* [21].

The element *ExecutionDomain* inherits from the EAST-ADL *AllocationTarget* and serves as a virtual container for hardware components. It is therefore possible to encapsulate arbitrary (and in particular arbitrarily distributed) execution nodes—e.g., CPU Cores—in an *ExecutionDomain*, which can then serve as a common allocation target element for a function allocation. This works even for cores that are distributed over multiple ECUs or for subsets of the set of all cores of a multicore ECU. The new element *CoreCluster* inherits from the EAST-ADL *Node* (which is an allocation target itself) and was introduced to support representations of manycore architectures, which usually are a prerequisite for supercore architectures. A supercore represents a cluster of cores in a manycore system that can be addressed and used like a single core. It usually consists of cores of one manycore CPU. Since supercores are intended to process allocated functions highly parallel by as much of the contained cores as possible, they are suitable in particular as a platform for highly concurrent functionality. The EAST-ADL supports hierarchical hardware modeling, so in principle a manycore system could already be represented with the current language specification. However, this would include the manual creation of potentially hundreds or even thousands of hierarchical cores by hand, which makes the use of a dedicated language element obvious. *CoreCluster* fills this role, by allowing a representation of a manycore CPU and its parallelization capabilities by means of attributes instead of a (ultimately not needed) hierarchical core model.

In conjunction with the new element *FunctionConcurrency*, which is used to annotate potential concurrencies for software functions, the scheduler can now make sensible allocation decisions at runtime: when allocating a function to an *ExecutionDomain*, the function is mapped to either one (without concurrency information) or a maximum of suitable execution nodes (with concurrency information) in the virtual container, which corresponds to the minimum of attribute *maxNumberOfThreads* and the actual number of suitable nodes. When allocating a function to a *CoreCluster* the concurrent functionality is mapped to all cores of the cluster. A supercore usually consists of cores of a single (real) manycore CPU, while an *ExecutionDomain* is a virtual container for arbitrary (heterogeneous) execution nodes that can be distributed in the system.

6.2. The supercore pattern in AUTOSAR

In the AUTOSAR (AUTomotive Open System ARchitecture) application layer the system functionality is realized by means of a functional decomposition using connectors and application software components [28]. These software components contain Runnables, which represent system routines and can ultimately be allocated to CPU cores using the OS services of the AUTOSAR Basic Software (BSW). For realizing limited multicore support, AUTOSAR introduced entity containers called OSApplications, which can be related to ECU partitions as defined by ISO26262. Software Components are mapped to OSApplications, which are then allocated to CPU cores.

When mapping the supercore concept to the existing AUTOSAR infrastructure [21], it becomes apparent that the general AUTOSAR constraint that an *OSApplication* must be allocated to a single core can no longer hold; instead, the OSApplication with all contained *Runnables* is to be allocated onto a cluster of cores. As a consequence, in order to support modeling of heterogeneous multi- and manycore systems in AUTOSAR the allocation relationship between *OSApplication* and cores effectively becomes n:m. Several problems connected to a manycore allocation can be solved (implicitly) by this measure, e.g., the allocation of a set of *Runnables* on a set of given cores could then be handled the same way as the parallel processing of concurrent functionality within *Runnables*. The necessary scheduling behavior must defined as an annotated policy for the *OSApplication*.

On the AUTOSAR metamodel level (M2) a supercore cluster can be defined as a specialization of an *ECUInstance*, thus an allocation of functionality is done similar to the single core case. On the AUTOSAR type level (M1)[5] an *OSApplication* has a policy for handling the concurrency behavior of the comprised *Runnables*.

6.3. Application of the modeling pattern

In Fig. 5 we demonstrate the application of the metamodel extensions by means of a mockup example on M1, i.e., the modeling level. The example includes two cases: firstly, an allocation of concurrent functionality *A* on diverse (and potentially distributed) execution nodes by means of an execution domain; and secondly, the allocation of concurrent functionality *B* on potentially all cores of one manycore CPU, which is represented by a core cluster.

While the latter case is more straightforward and requires less effort on part of the architect, it will not be suitable for all scenarios in practice. Making use of the execution domain concept not only makes it possible to allocate functionality on arbitrary and distributed hardware, but also allows for selecting only a subset of all cores of a multi- or manycore CPU (as opposed to the set of all cores of the CPU) as an allocation target. This is shown in the example, where the execution

[5] M1 elements in the AUTOSAR meta model are to be understood as attributes, which are defined in the context of the configuration of the BSW.

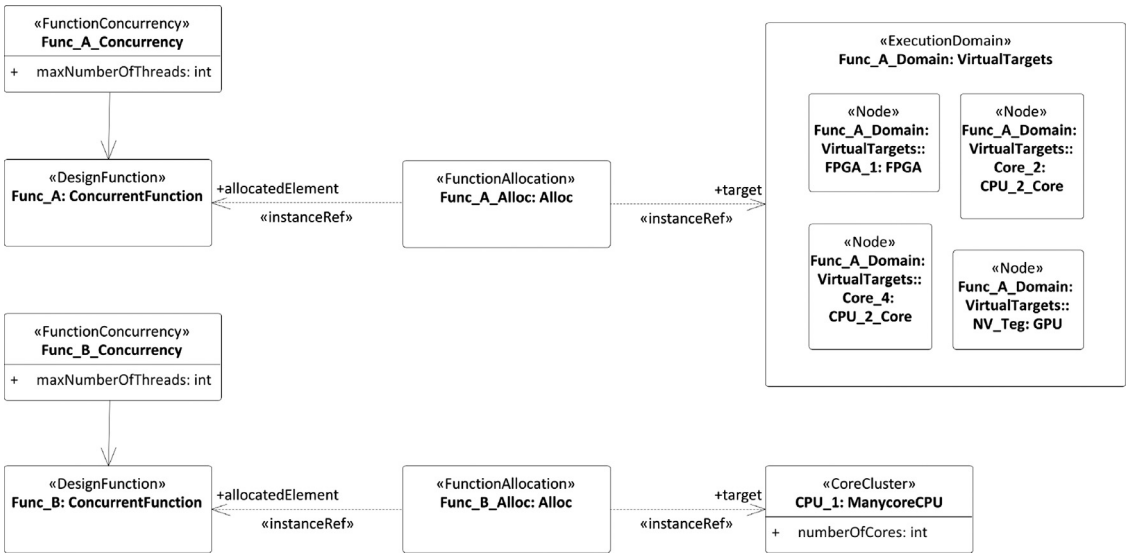


Fig. 5. Application example of Supercore language extensions on M1-level.

domain not only includes a GPU and an FPGA, but also a subset of all cores of the multicore CPU_2, i.e., cores 2 and 4 and no others.

The execution domain concept furthermore facilitates the use of affinities, both functional and in regard to the execution nodes, i.e., the hardware. Affinities are a related concept that were proposed for the language specifications in the context of the FORMUS^{3}IC project. The use of affinity annotation enables the scheduler to make intelligent decisions about the dynamic allocation of concurrent functionality onto heterogeneous hardware components (that are part of the execution domain) at runtime.

7. Conclusion

In this paper we presented further evaluation of the supercore pattern, an extension to the fork-join pattern. By grouping multiple cores into a supercore with coupled sub-tasks and coherent scheduling, it is possible to reduce the response time of parallel calculations. We implemented the pattern on a real-world EMS from Continental by parallelizing two algorithms with the help of the pattern.

The supercore pattern enabled us to reduce the response times of the algorithms by a factor of 1.97 resp. 1.7 when using 4 cores. The speedup is limited not only by the sequential parts in the algorithm, but also by the non-uniform memory architecture, which leads to slow data access for some of the cores. In order to optimize the speedup, the data must be partitioned carefully, such that calculations on each core have the same runtime. In our example setup, a significant speedup is only possible when the calculation runtime is at least 100 µs, and an optimal speedup is achieved for runtimes of 400 µs or more. For shorter runtimes it is advised to use the remaining cores for other calculations rather then using them for parallelization.

The findings confirm the results of our previous research and show that our pattern is functional and can be used in the real-world, as long as the given constraints are met. With concrete language extensions we were able to show how the supercore pattern can be modeled in EAST-ADL and AUTOSAR.

In future work, the supercore pattern should be implemented on a real-world automotive processor using a Gang Scheduler with Earliest Deadline First Policy, as proposed by Kato [23] or Osinski et al. [29]. To embed this on top of the current OSEK Stack a similar method as [25] could be used. More algorithms should be ported and response time analysis, as well as worst case execution time analysis should be performed. The effect of data-to-core mapping should be also analyzed more in depth; here, with suitable compiler-based configuration tools an optimized data access pattern for the cores can be imagined (e.g. as described in [30]).

Acknowledgment

The authors would like to thank Continental Automotive GmbH for providing the measurement facilities, as well as the personal support for the real-world measurements.

Funding: This work was supported by the Bayerische Forschungsstiftung [grant number AZ-1165-15].

References

[1] Barr Group, Embedded Systems Safety & Security Survey (2018).
[2] Infineon, AURIX 2nd Generation & TC3xx, 2017, https://www.infineon.com.
[3] AUTOSAR, Classic Platform Release Overview 4.3.0, Technical Report, 2016.
[4] C. Hartmann, R. Mader, L. Michel, C. Ebert, U. Margull, Massive parallelization of real-world automotive real-time software by GPGPU, in: ARCS 2017; 30th International Conference on Architecture of Computing Systems, 2017, pp. 1–8.
[5] R. van Wagensveld, U. Margull, Experiences with HPX on embedded real-time systems, in: 2017 International Conference on Applied Electronics (AE), 2017, pp. 1–6, doi:10.23919/AE.2017.8053626.
[6] H. Blom, H. Lönn, F. Hagl, Y. Papadopoulos, M.-O. Reiser, C.-J. Sjostedt, D.-J. Chen, R. Tavakoli Kolagari, White Paper Version 2.1.12: EAST-ADL - An Architecture Description Language for Automotive Software-Intensive Systems (2013).
[7] I. Foster, Designing and Building Parallel Programs, 78, Addison Wesley Publishing Company, Boston, 1995.
[8] B.L. Massingill, T.G. Mattson, B.A. Sanders, Patterns for parallel application programs, in: Proc. of the 6th Pattern Lang. of Prog.(PLoP'99), 1999.
[9] T.G. Mattson, B.A. Sanders, B. Massingill, Patterns for Parallel Programming, Addison-Wesley, Boston, 2005.
[10] K. Keutzer, T. Mattson, Our pattern language (OPL): a design pattern language for engineering (parallel) software, ParaPLoP Workshop on Parallel Programming Patterns, 14, 2009.
[11] M. Gerdes, R. Jahr, T. Ungerer, parMERASA Pattern Catalogue: Timing Predictable Parallel Design Patterns, Technical Report, Informatik, 2013.
[12] R. Jahr, M. Gerdes, T. Ungerer, On efficient and effective model-based parallelization of hard real-time applications., in: MBEES, 2013, pp. 50–59.
[13] M. Paolieri, J. Mische, S. Metzlaff, M. Gerdes, E. Quiñones, S. Uhrig, T. Ungerer, F.J. Cazorla, A hard real-time capable multi-core SMT processor, ACM Trans. Embed. Comput. Syst. 12 (3) (2013) 79:1–79:26, doi:10.1145/2442116.2442129.
[14] J. Delange, P. Feiler, Design and analysis of multi-core architectures for cyber-physical systems, in: Proceedings of the Embedded Real-Time Software and Systems Conference (ERTSS2014), IEEE, 2014.
[15] S. International, Architecture Analysis and Design Language (AADL), Technical Report, Society of Automobile Engineers International, 2012. AS5506
[16] A.W.D.O. Rodrigues, F. Guyomarc'h, J. Dekeyser, A modeling approach based on UML/MARTE for GPU architecture, CoRR abs/1105.4424 (2011). http://arxiv.org/abs/1105.4424.
[17] OMG, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1, Technical Report, Object Management Group, 2011. formal/2011-06-02
[18] AMALTHEA Partners, Amalthea homepage http://amalthea-project.org/.
[19] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture Volume 1: A System of Patterns, volume 1 edition, Wiley, Chichester; New York, 1996.
[21] R. van Wagensveld, T. Wägemann, N. Hehenkamp, R.T. Kolagari, U. Margull, R. Mader, Intra-task parallelism in automotive real-time systems, in: Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores, in: PMAM'18, ACM, New York, NY, USA, 2018, pp. 61–70, doi:10.1145/3178442.3178449.
[22] Our Pattern Language, Fork-Join, 2017. https://patterns.eecs.berkeley.edu/?page_id=252.
[23] S. Kato, Y. Ishikawa, Gang EDF scheduling of parallel task systems, IEEE, 2009, pp. 459–468, doi:10.1109/RTSS.2009.42.
[24] OSEK group, OSEK/VDX Operating System Specification 2.2.3, 2005.
[25] C. Diederichs, U. Margull, F. Slomka, G. Wirrer, An application-based EDF scheduler for OSEK/VDX, 2008, pp. 1045–1050, doi:10.1109/DATE.2008.4484819.
[26] N. Hehenkamp, R. van Wagensveld, D. Schoenwetter, C. Facchi, U. Margull, D. Fey, R. Mader, How to speed up embedded multi-core systems using locality conscious array distribution for loop parallelization, in: ARCS 2016; 29th International Conference on Architecture of Computing Systems, 2016, pp. 1–5.
[27] E.-A. Association, EAST-ADL Domain Model Specification (2013). Version V2.1.12. http://east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf.
[28] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, T. Scharnhorst, Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures, Convergence (2004) 325–332.
[29] L. Osinski, T. Langer, R. Mader, J. Mottok, Challenges and opportunities with embedded multicore platforms – a spotlight on real-time and dependability concepts, ERTS 2018, 2018.
[30] J. Kienberger, C. Saad, S. Kuntz, B. Bauer, Efficient parallelization of complex automotive systems, in: Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, ACM, 2016, pp. 40–49.