Schedule-aware Distribution of Parallel Load in a Mixed Criticality Environment

Marc Bommert
RheinMain University of Applied Sciences, Wiesbaden, Germany
Email: marc.bommert@hs-rm.de

***Abstract*—This paper presents an approach to segment divisible load, i.e. parallelized algorithms, in a mixed criticality multi-processor system. Segmentation is based on utilization of processors by higher criticality scheduling layers. The ambition is to establish load distributions with reduced overhead compared to non-clearvoyant distributions. The approach tends to be deterministic in order to guarantee real-time capabilities, i.e. bounded maximum execution time of parallelized segments, by avoiding thread migration completely. We present two methods of load segmentation: A static distribution based on WCET analysis of HRT-tasks and a second method to overcome the issues of the static approach by means of backfilling.**

I. Introduction

A. Contribution

We present a novel approach to segment lower critical divisable load in a partitioned mixed criticality (MC) multi-processor environment, based on information on the temporal behavior of higher critical tasks. It intents to reduce the overhead of workload distribution for the sake of throughput and to facilitate task parallelization, while preserving predictability to a maximum. The main target is to segment parallelized iterations in a way, such that the parallel subtasks achieve equal completion times, even if inhomogenous processing time remains on each processor due to the partitioned execution of higher critical tasks.

Tasks of highest criticality are per definition well-analyzed in terms of their functional and temporal behavior to prove them correct. The result of system level temporal analysis is usually a time budget accounted to a task in order for it to meet its deadline in any possible system state. We show that the proportions between these time budgets can be consulted as an evaluation criterion for proper load distribution between lower criticality subtasks, resulting in less coordination effort, i.e potential earlier completion of the overall task. Since subtask/worker-thread migration is not required with this approach, it basically allows bounded worst case execution times.

B. Organization of this paper

In Section 2, we describe the theoretical background. A suitable MC reference model is defined. The OpenMP standard for shared memory parallel programming is briefly depicted. A description of the identified problem scenario finishes Section 2. In Section 3, we present our approach of defining processor utilization functions and using these functions to control load distribution to tasks. We introduce two methods, a statically weighed distribution and a hybrid, partially dynamic, distribution. A section with simulation results and an outlook on further work complete this paper.

II. Background

A. Conventions

To avoid ambiguity we define the used terms: A *process* is an instance of a computer program. A process comprises one or more *threads*, flows of execution, which can be independently scheduled by the operating system on the system's processors. Threads providing means to execute arbitrary application-specific *tasks* as workload are called *worker threads*. A *real-time task* denotes a part of a real-time (RT) application with sporadic or periodic invocation and a certain *deadline*. We distinguish between a *hard real-time* (HRT) task, which never misses its deadline and a *soft real-time* (SRT) task, which may miss its deadline, but guarantees bounded *tardiness* to still ensure a certain *quality of service*.

B. Mixed Criticality Scheduling

The MC concept has been introduced for the avionic domain in order to reduce size, weight and power (SWaP) of embedded systems. The fundamental idea is to lower the number of hardware platforms in a system by consolidating independent software modules onto the same platform. An important aspect of this reduction is proper fault isolation through *temporal* and *spatial partitioning*. The SWaP paradigm is promising for mobile applications in general, e.g. automotive ones, due to the rising importance of multiprocessor platforms and strong demand for fault isolation [1].

For transfering the MC approach from single processor (SP) systems towards multiprocessor (MP) systems, since RT scheduling on a MP system has been proven to be an NP-hard problem [2], a common approach is to partition the processors against each other for HRT tasksets. This reduces schedule verification efforts on MP systems to methods similar to those used on SP systems, which are well known.

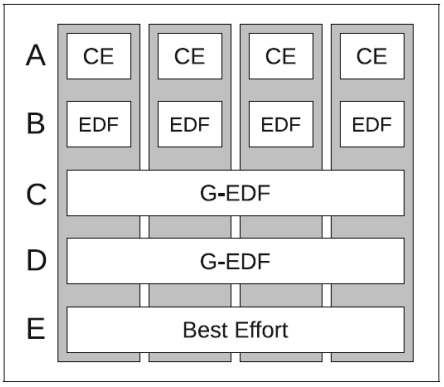

Fig. 1. Container scheduling in a MC environment, Mollison et al. [3].

Our reference model of a MC *stack* is inspired by MC^2 [3]. Figure 1 shows the MC^2 stack, with each of the nested layers implementing a specific scheduling algorithm:

Layer A uses a *cyclic executive* (CE) for applications of highest criticality with HRT constraints. The second-highest layer B uses *partitioned EDF scheduling* (P-EDF) for applications of highest criticality with HRT constraints. A third tier, comprising layer C and layer D uses *global EDF scheduling* (G-EDF) for applications with SRT constraints. Finally, the lowest layer E assigns remaining processing time with, for example, *global fixed-priority scheduling* to processes that are executed in a *best effort* manner.

In such an MC stack, tasks on each layer are preempted if tasks on a higher scheduling layer become eligible for execution. Preemption delays are accounted when RT behavior is analyzed. In practice, in contrast to the referenced MC stack, on layer B, rate monotonic scheduling (RMS) is more commonly used than P-EDF due to its simplicity and higher determinism in case of deadline misses. RMS, however, is not optimal for tasks whose periods are not harmonic. Thus, in most real-world scenarios, a certain processor capacity will remain unused by scheduling layer B. For non-periodic tasks in RMS, it is common practice to be mapped to periodically planned task-containers [4], with a period according to the highest possible rate of readiness. This additionally will lead to more processing time being accounted than being used in average when RMS applies at layer B.

C. Shared Memory Parallel Programming with OpenMP

Whilst using a MP platform allows parallel processing to execute independent single-threaded applications in parallel-this can be legacy applications that are consolidated in a MC approach-, it also allows algorithms to exploit real parallelism, e.g. performing segments of computations on multiple processors, collecting the results. Computational problems face a permanent rise in complexity and amount of to-be-processed data. Due to the challenge for modern SP systems to further increase circuit density and throughput to satisfy Moore's Law, this is one important method to cope with increasingly strong temporal requirements.

Parallel programming usually involves explicit definition of thread behavior in terms of synchronized execution and data access. Until today, it is a topic in research to simplify this process by abstracting to a parallel programming model which hides these issues from the more functional considerations of development. Such a standardized and widely used programming model is OpenMP [5]. It allows shared memory parallel programming within a single process, starting parallel worker-threads on demand and hiding their specific synchronization and data access. OpenMP imposes a thread *fork-join* structure to underlying software layers, i.e. the operating system (OS) scheduler.

Due to its size, its high level of abstraction and the methods of work allocation to be found in current implementations, e.g. distributing subtasks to processors from a global queue, OpenMP itself has not been payed much attention in development of RT systems. The fork-join model of OpenMP imposes tasks with *zero laxity* to the scheduler, which cause worse schedulability. Transforming fork-join task models to improve their schedulability is a recent topic in research [6].

An example of an OpenMP-parallelized loop is shown in Listing 1. A large amount of iterations is implicitly parallelized into independent subtasks of equal weight by the preceding preprocessor statement. Threads are forked at loop entry and have to join at loop exit. There is no explicit concurrent data access. The shown example may implement any single-instruction, multiple-data algorithm with independent iterations, e.g. vector operations, tree or cube traversal, searching or even sorting.

Listing 1. C Language Example Code: OpenMP-parallelized for-loop

```
#pragma omp parallel for schedule (static)
for (int i = 0; i < 1000000; i++) {
  result[i] = processData(&data[i]);
}
```

D. Problem scenario & Incentive

In a multi-processor MC system, highest scheduling layers work in a partitioned or clustered manner. Furthermore, application software of high criticality is verified with corresponding high detail regarding its temporal behavior. Worst case execution time (WCET) is determined, accounted to tasks, and forms deadlines. For layers of lower criticality with SRT requirements or without temporal requirements, the partitioning is relieved and tasks are allocated to processors from a global ready queue.

Thus, software (with or without RT constraints) demanding parallel execution of algorithms, such as the fork-join model applied by OpenMP, is not directly considered by the MC approach. Due to inhomogenous remaining processing capacities on scheduling layers below partitioned HRT scheduling, throughput of parallel algorithms is not efficient, if the distribution of parallel workload to processors does not consider RT task parameters from higher criticality layers. Due to excessive verification effort required at the highest layers, those tasks parameters are usually to be determined anyway. They could easily be used to optimize workload distribution, although it is important not to expose them to isolated tasks of lower criticality. A trusted layer for workload distribution, i.e. an implementation of the OpenMP runtime libraries at lower levels of the MC stack, is suitable to overcome this issue in order to potentially improve efficiency while preserving fault isolation and preventing covert information channels from higher towards lower criticality processes. Furthermore, if a fixed mapping between worker threads to the system processors is used, RT guarantees can basically be given by means of bounded execution times of parallelized task segments.

III. A SCHEDULING LAYER FOR DIVISIBLE LOAD IN A MIXED CRITICALITY ENVIRONMENT

A. Overview

We define an additional scheduling layer in the presented MC model. This new scheduling layer is located below the two topmost partitioned scheduling layers A and B. Threads of that new layer are preempted when HRT-tasks of a higher layer become ready. In turn, activity in lower scheduling layers is preempted when either the new layer or HRT layers signal readyness. In the new layer, work is distributed globally for all processors of the system or a specified cluster of at least two processor nodes.

For simplification, we assume the new scheduling layer to contain a single process for now. This process interfaces a RT capable subset of the OpenMP programming standard. It realizes a single parallelized loop as introduced in listing 1. When the loop is executed, a number of worker threads, one

for each processor, become eligible for execution. Each of the threads is assigned a subtask to process a disjoint share of the parallelized loop. The share, i.e. the loop iteration count to be assigned to a specific subtask, is determined by a *distribution function* which itself decides about workload distribution based on a *utilization function*.

B. Utilization Functions

If a parallelized task is executed on a lower MC layer, below the partitioned HRT-tasks, a processor may be claimed by higher criticality tasks. Since the superior task schedule is static and a complete WCET analysis is available, information on worst case processor utilization by HRT tasks can be statically provided as a utilization function $u(i, t_0, t_1)$. It delivers the worst case utilization of processor i in $[t_0, t_1)$.

In a periodically planned partitioned MC system, a system *global period* T_p can always be derived as the lowest common multiple of all task periods of all partitioned tasksets. For a realtime system which measures and controls a real-world process with high frequency, task periods are usually short. Thus, the system global period is also potentially short. More precisely, it is required that the overall runtime of the parallel sequence is a large multiple of T_p. This allows to consolidate utilization functions for whole multiples of the system global period T_p with reasonable error, despite the exact execution time of the parallelized task segment.

T_p is the period of the utilization function. The functions result is a normalized value: A result of 0 indicates an idle processor i and a result of 1 indicates that the processor will not be available at all in the given interval. The utilization function is defined at system integration time, based on static HRT task parameters, e.g. their period and their WCET.

For the following definition of workload distribution methods, we distinguish between worst-case utilization of each processor, based on WCET analysis of RT tasks, and its best-case utilization. Therefore, we distinguish between worst-case and a best-case utilization functions, $u_{WCET}()$ and $u_{BCET}()$.

C. Weighed Static Distribution of Parallel Load

From $u_{WCET}()$, certainly remaining processing time of processor i can be determined as $1 - u_{WCET}(i, t_0, t_1)$. The proportions between worst case processor utilization define a proportional distribution function $z(i, N, m, t)$ which is consulted at each fork operation for each of the involved processors $[1..m]$:

$$z(i, N, m) = \frac{1-u_{WCET}(i,0,T_p)}{\sum_{k=1}^{m}(1-u_{WCET}(k,0,T_p))} * N = N_i$$

Where N is the overall loop iteration count as illustrated in listing 1, and N_i is the loop iteration count to be distributed to processor i.

This proportional distribution function z is consulted during execution of the fork operation of the parallel task segment. It is called exactly once per processor per parallelized loop to decide about the proportions in which N is shared between subtasks. Load segmentation according to z would be optimal, if the effective execution time of higher criticality tasks would always equal their WCET and the runtime of the parallel task segment would be a whole multiple of T_p. Depending on the execution time variance of higher criticality tasks, and the parallel segment's execution time, the distribution function z is thus presumably always affected by a certain error. This error has to be tolerated to give RT guarantees to the parallel task segment. Dividing the parallel segment based on WCET analysis of higher criticality tasks will thus barely ever achieve an optimal result. Nevertheless, this *static weighed* approach is able to reach better results than a static distribution not considering processor utilization by higher criticality tasks, which we will further refer to as *static naive*.

With use of the static weighed method, *starvation* of parallel subtasks is prevented by design. It is technically easy to limit the maximum processor usage and to inherit definitive processing time to lower MC layers.

D. Hybrid Distribution of Parallel Load

In order to overcome an issue of a static weighed distribution based solely on the WCET of higher criticality tasks, a *hybrid distribution* concept seems promising. Instead of entire static segmentation of load, a hybrid algorithm would predivide load into a fraction N_s which is assigned statically and a second fraction N_d, which is assigned dynamically. This allows threads finishing their statically assigned share early, to actively claim new work. Such *backfilling* is potentially capable to reduce or eliminate inserted idle time caused by reduced utilization of a specific processor at higher criticality layers. Its cost is an increased number of potential synchronization operations between workers when dynamically requesting new work, i.e. a certain fixed number of loop iterations.

To pre-segment the overall number of loop iterations, the band between WCET and BCET in each processor's RT-taskset may be considered by means of the utilization functions u_{WCET} and u_{BCET}. The dynamically allocated part has to be sufficiently large to compensate the expected imbalance in processor utilization due to execution time variances of higher criticality tasks. Therefore, processing capacity definetly not used by a processor by HRT tasks $c_{def} = 1 - u_{WCET}$ is distinguished from utilization that may additionally not be required $c_{pot} = u_{WCET} - u_{BCET}$. N_s and N_d can be determined to:

$$N_s = N - N_d = N * \frac{\sum_{i=1}^{m} c_{def}}{\sum_{i=1}^{m}(c_{def} + c_{pot})}$$

By choosing the number of operations which are dynamically allocated on worker thread request, speaking in OpenMP terminology, the *chunk size* of dynamic work allocation, this approach trades between worker synchronization overhead and the finally remaining maximum inserted idle time. The divisibility factor, i.e. the WCET of a single loop iteration, provides an ultimate limit.

IV. Simulation Results

Figure 2 shows simulation results comparing three types of workload distribution methods. The simulation platform was a dual-core processor running a vanilla Linux kernel. One of the two processors was constantly stressed to a known share ($\frac{1}{3}$) by a highest priority thread simulating a cyclic executed RT task with a period of 3ms. The given plot compares a static naive workload distribution, a fully dynamic workload distribution, and our clearvoyant static weighed distribution which allocated shares according to remaining processing capacities. A single work cycle iteratively computed a short Fibonacci sequence. The workload consisted of 100 million cycles. Dynamically assigned chunks consisted of 1000 single cycles. Each simulation was repeated 500 times.

It can easily be seen, that the static weighed distribution outperforms the naive distributions. Its main advantage

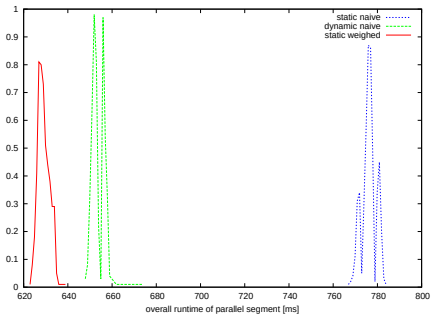

Fig. 2. Static weighed distribution with constant stress from higher priority task.

compared to the static naive distribution is the drastically reduced inserted idle time at the unstressed processor node. The advance to the fully dynamic (naive) distribution is due to reduced worker synchronization.

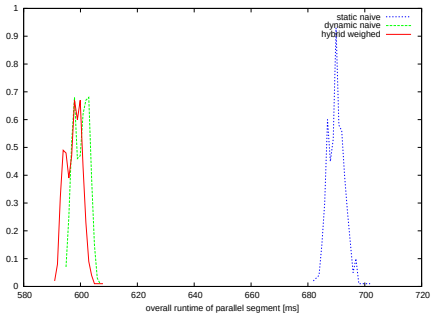

Fig. 3. Hybrid distribution with variable stress from higher priority task.

Figure 3 shows simulation results comparing three types of workload distribution methods. One of the two processors was dynamically stressed within known bounds ($\frac{1}{6}$ to $\frac{1}{3}$) by a highest priority thread simulating a cyclic executed RT task with a period of 3ms. The given plot compares a static naive workload distribution, a fully dynamic workload distribution, and our clearvoyant hybrid weighed distribution. Simulation parameters were as mentioned before, but dynamically assigned chunks consisted of 1 million single cycles.

Again, we observe a definite advantage of our distribution over the static naive distribution. More interesting, the hybrid method also performs slightly better than the dynamic naive distribution, although the latter is subject to only 100 work-stealing cycles, i.e. woker-thread synchronizations, during each run.

V. Conclusion

A. Discussion

In this paper, we described a novel approach for scheduling divisible load in an MC system at a layer which is subordinate to partitioned HRT scheduling. We made clear how parameters of RT tasks may be considered in order to determine an adequate segmentation of large numbers of independent calculations. We focused on calculations with equal demands of processing time, e.g. OpenMP-parallelized for-loops.

We concluded that the RT system could provide utilization functions to predict definetly and potentially unused processing time for each processor. Data provided by such functions, based on static information from the system integration domain, could be used to segment divisible load regarding the optimality principle, i.e. reaching equal completion time of subtasks. The resulting optimization towards the optimality principle would improve efficiency of parallel task execution.

Since subtask/worker-thread migration is not required with this approach, it basically allows bounded worst case execution times and, consequently, it provides a HRT-capable interface for parallelized algorithms.

B. Future Work

We currently develop a prototype implementation of the presented approach by gradually establishing a subset of an OpenMP-compatible environment on basis of a RTOS. This environment will have to interface the integration domain to gather highest criticality task parameters, i.e. WCETs and BCETs. Load distribution within this environment will then consider these parameters. Prior attention lies on parallelized loops, that is, dynamic (runtime) assignment of loop iterations to worker threads. So far, we can say that the scheduling methods that can be interfaced through the OpenMP API, namely *static, dynamic and guided*, suit the presented approach. Such a middleware layer aims towards a general purpose solution. Static pinning of worker threads to processors, disallowing the (non-clearvoyant) OS scheduler to migrate them between processors, will presumably allow execution time guarantees to be given in order to fulfill HRT requirements. Verification has to consider specific, yet to be defined, use cases reproducing real-life scenarios such as processing multimedia data with and without RT constraints.

In our future work, the method can then eventually be extended towards multiple parallel task segments, nested parallel task segments and parallel task segments that explicitly access shared data and thus are subject to locking. The approach may also be adapted to parallel segments with inhomogenous runtime which itself are weighed against each other by means of annotations. Limits in memory- and bus-bandwith have not been considered yet, but are crucial in a real-world implementation and thus need further attention. Methods towards the system integration domain have to be designed in order to automatically define utilization functions based on given or gathered HRT task parameters.

References

[1] ISO 26262, *Road Vehicles Functional Safety*, 2011.

[2] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 01, no. 2, pp. 184–194, 1990.

[3] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, ser. CIT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1864–1871.

[4] L. Sha, M. Klein, and J. Goodenough, "Rate monotonic analysis for real-time systems," Carnegie Mellon University, Software Engineering Institute, Tech. Rep., 1991.

[5] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008.

[6] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st.* IEEE, 2010, pp. 259–268.