Global Adaptation for Energy Efficiency in Multicore Architectures

Alina Lenz*, Tobias Pieper*, Roman Obermaisser*

*University of Siegen, Germany

Email: {alina.lenz|tobias.pieper|roman.obermaisser}@uni-siegen.de

Abstract—Today mixed-criticality systems are used in most industrial domains, because of their integration advantages. They are smaller, weigh less and reduce the idle time of the previously dedicated hardware.

However, these systems can still be improved. Since their hardware is now used more efficiently it automatically suffers more under the aging effects of the heat created by all the simultaneous computations. The heat fastens the aging process of the hardware and increases failure rates. To prevent this the systems need to be cooled down by additional cooling devices like fans. In turn, these devices introduces new failure sources due to their movable parts.

In this paper we propose a chip-wide approach to dynamically manage the system computation and communication to optimize the energy-efficiency. By reducing the energy usage of the system we can reduce the additional hardware as well as the weight of the whole system and prolong the system's lifetime as the available power resource lasts longer. We expand the current usage of tile-based energy management to a system wide scheme by implementing a meta-scheduler. This verifiably monitors the system state and changes the schedule if an optimization can be performed.

I. Introduction

Mixed criticality systems bear many integration advantages for industrial use by sharing the same system on chip (SoC) for applications of mixed criticality which formerly were run on dedicated hardware [1]. The integration takes up less space and reduces the overall energy consumption because the idle time of one critical application can be used efficiently for another application.

In order to ensure their restrictions, many safety-critical applications employ a precomputed temporal behavior defined in a schedule. This schedule defines when which task is allowed to be executed and when messages may be sent. Such a predictable behavior is desirable as the system can be designed to guarantee timeliness.

With the rapid growth of the industrially used MPSoCs the energy consumptions grows in importance. Each energy consumer produces heat which in turn negatively impacts the hardware. The authors of [2] have shown that each $10°C$ the system's failure rate increases by 50%. For this reason fans and other dynamic elements need to be integrated to cool down the system. These dynamic elements are prone to failure and weigh a lot. Reducing these elements is a major concern in today's industry. Energy and reliability optimization are considered to be among of the most critical issues of multiprocessor systems-on-chip (MPSoCs) [3].

One approach to reduce these additional devices is to manage the system's energy consumption more efficiently. Reducing the overall energy consumption will not only reduce the need of cooling devices and hence the negative effects on the hardware aging process. It will also increase the system's lifetime as the power source will be depleted more slowly.

Much research has been done in how to effectively reduce the energy consumption by applying dynamic energy saving mechanisms like dynamic voltage frequency scaling (DVFS). However, most of the approaches do not target a system-wide optimization for mixed-criticality systems.

DVFS can be applied when unpredictable idle times occur in the system. These times called dynamic slack occur quite often since the real runtime of tasks is not exactly known at design-time. Therefore, the schedules are computed based on an assumed Worst Case Execution Time (WCET) which is the maximum time that a resource will be allocated for the corresponding task. To ensure, that high criticality tasks finish on-time with a high probability, their WCETs are assumed more pessimistic than the ones of low criticality tasks. These assumed times lead to an unbalanced usage of the system. In average conditions, the tasks will be executed faster than expected leading to dynamic slack. The slack can be used to run other tasks at lower frequency because the lower frequency leads to longer task execution times.

We propose a system wide energy management that exploits opportunities to save energy. We demonstrate our approach using dynamic slack, although the solution can also be applied to other use scenarios such as fault recovery. The proposed adaptation reacts to context events such as dynamic slack by rescheduling the computations and the communication. These changes are done based on a verifiable schedule wherefore our approach is suitable for safety critical domains like avionics or rails.

The paper is structured into six sections. Section II discusses related work and places our approach within the canon of already existing energy management methodologies. Section III presents our target architecture and in section IV we explain our meta scheduler. In section V the simulation of our approach is described and section VI provides a conclusion with an outlook on our future work.

II. Related Work

Energy efficiency based on scheduling is a highly researched topic ranging from energy management techniques to very

specific energy management applications on MPSoC [4] [5].

Dynamic voltage frequency scaling (DVFS) has been widely researched in embedded systems. This technique exploits the fact that the consumed energy is calculated as: $E = c * V^2 * f$ with c being the system capacitance, V the voltage and f the frequency. Both factors can be reduced to lower the energy consumption. Most approaches aim for a localized tile-based optimization where the slack is distributed to other locally executed tasks or the core frequency is dynamically lowered during a task execution. In these approaches the aim is to optimize the time-slot given by the schedule.

[6] proposed an inter-tile slack propagation, to optimize the effect the slack has on the system. Slack can be used by DVFS to perform executions at a lower frequency.

Their approach to use the slack globally is very similar to our approach, but we we extend it into a temporally predictable and certifiable technique for mixed-criticality systems.

A. Local energy aware scheduling for real-time systems

Other approaches to save energy focus on improving the schedule the system is running upon. They define the timeslots of the tasks based on heuristics to implement less pessimistic timeframes and hence reducing the idle time.

They use these heuristics in on-line scheduling to dynamically adjust to the system state and thus approximate an optimal off-line scheduler [7] which would have known the real execution times beforehand.

Our schedule on the other hand approximates the dynamic behavior with precompiled schedules.

B. Super scheduler

A super scheduler injects new high criticality messages into the system to deal with critical events while maintaining the deadlines. The critical events get downgraded and interrupted by the new events [8] [9].

In contrast to this prior work, our meta-scheduler will not disturb the current high critical applications as the changes will be planned at design time. We will not dynamically reconfigure the system but change into a new schedule which supports the system's deadlines.

Our approach will combine offline pre-computed schedules with the dynamic adaptation to the system state.

III. SYSTEM MODEL

The system model consists of a tile based architecture which is connected via a network on chip. We perform applications of mixed criticality on the tiles with real-time requirements. Therefore our architecture is chosen to enable strict temporal and spatial separation to ensure that no low criticality application can impact or delay a higher critical one or access its dedicated resources. The system must also ensure that all tasks can be performed within their associated deadlines. Real-time systems work with time sensitive data that needs to be evaluated and acted upon directly, e.g. sensor data. Especially in safety-critical systems missing these deadlines can cost human lives, e.g. if the altitude measurement in flight control systems fails.

Figure 1 shows how each tile is managed by a hypervisor running bare-metal on the tile's hardware and allowing for multiple partitions. The applications on these partitions can be of varying criticality though to keep the spatial separation only the same criticality can be executed within a single partition.
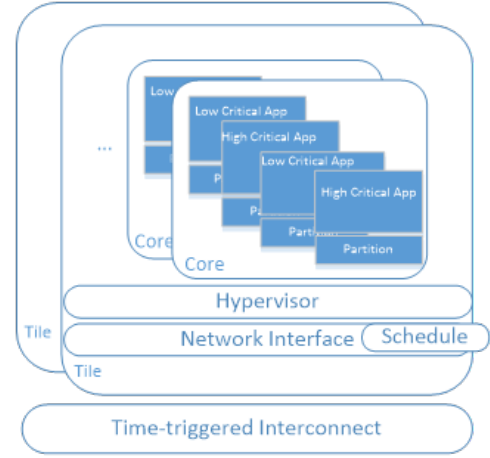

Fig. 1. Architecture

The hypervisor assigns hardware resources of its tile to its managed partitions, ensuring the timing and spatial restrictions for the mixed criticality system. It also manages the partitions by scheduling their execution by the corresponding deadlines to provide the real-time functionality. Since the partitions do not access the hardware on their own, the messages they send are also relayed by the hypervisor. This prevents message flooding to the network on chip (NoC) and the applications of other tiles are protected from faulty behavior.

Each tile is connected via a network interface (NI) to a time-triggered NoC. The time triggered behavior is executed based on an a priori computed communication schedule which commands the message injection times [10]. The schedule is enforced by the NIs which enqueue the messages from the core and inject them into the network based on the precompiled communication schedule. This schedule ensures that no packets collide during the transmission and guarantees timing bounds for the message traversal. For this, the schedule does not only control the message injection but also the path the message is routed on. In addition, the NI contains a scheduler in the packetization and flitization units which compares the current time to the schedule and initiates the message injection at the preplanned points in time.

A. Schedule

The schedule is a function which temporally and spatially allocates a set of executable actions to a physical model and a period w.r.t. their deadlines and dependencies. The executable actions are all actions that need to be performed with a defined period of time by a defined set of hardware. In our case we look at two kinds of schedules: the computational schedule executed by the hypervisor and the communication schedule for the message injection. These schedules depend on each

other as the messages to be sent are computed by the tasks managed by the hypervisor. The computational schedule is defined by the tasks, their start and their end time. As one can see the start and end times are defined by scheduling the tasks based on the assumed WCET. The communicational schedule contains all messages that will be sent on the NoC and their injection times.

The physical model is defined by the available hardware of the system, including resources like cores, memory, routers and I/O devices.

The schedule assumes the worst case execution time (WCET) when planning the order of task executions, to assure enough time for all applications to finish the task in the assigned time frame. In mixed criticality systems, WCET assumptions can vary based on the criticality level. Low criticality tasks can be assigned an optimistic time slot, as faults are acceptable here. High criticality tasks will always be assigned pessimistic time frames, as the tasks have to be performed at all costs. Reassigning allocations too early will cause fatal system failures. Figure 2 shows that to ensure meeting the deadline the WCET assumption is always an overestimation. In case the task actually finishes at the WCET the overestimation saves the system from deadline misses. However, the figure also shows that the usual execution time is often within the first third.
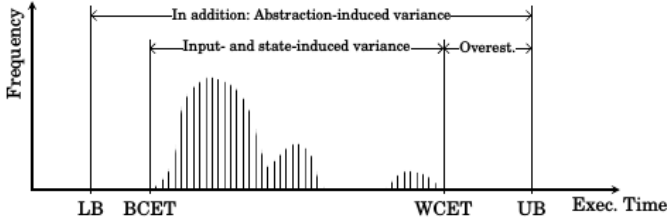

Fig. 2. WCET distribution [11]

The difference between the estimated execution time and the real execution time is called slack and can only be determined at runtime. This makes an optimization impossible in static, prescheduled systems.

The communication schedule depends on the computation schedule: the communication schedule awaits and plans a message at the expected end-time of a computation, which must be approximated a priori with the WCET. In turn, this means that even if a computation is finished before the WCET, the resulting message will be transmitted after the WCET but not instantly. This dependency can be interpreted as a time contract where the communication commits to send messages in provided time slots. Any changes on the system behavior must comply to these predefined "contractual" times. State of the art power management schemes use the inevitable slack resulting from the difference between the WCET and the real execution time to perform DVFS within the tile. These schemes ensure that the prolonged execution due to DVFS ends by the assumed WCET to comply with the communication schedule. Such software based approaches are

usually performed by the hypervisor or the partition software (RTOS).

IV. THE META-SCHEDULER

A. Motivation

The currently applied energy management is software based. Therefore, it is always focused on the local tile due to the inherent scope of the software performing the management. E.g. the hypervisor only has knowledge about his own tile. We want to expand this *local* energy management to a hardware based *global* energy management scheme. By using the local slack in a global optimization, the network can be optimized to a global rather than just the local optimum. This leads to a maximized system lifetime, especially in systems with cross tile task interdependencies. In those, the output of one tile is needed by other tiles to start their computation. For example, a five tile architecture has dependencies where the first tile computes inputs for the other four tiles which themselves compute inputs for the first tile, as shown in figure 3.
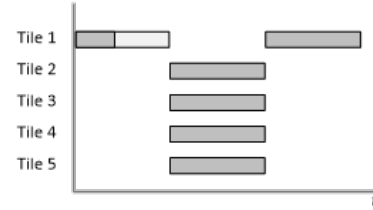

Fig. 3. Example for task dependencies in a multicore architecture

If the first task finishes its execution earlier than the assumed WCET, this slack time can be used for energy optimization. A currently often used technique is to gradually check the execution time. When the monitoring software then recognizes a faster execution than expected, it dynamically reduces the voltage and frequency of the execution. This way, the WCET timeframe is used more efficiently. In our example, local optimization would gradually adapt the frequency on core one. This would extend the core's execution time under lower frequency to reduce the energy consumption which in turn leads to a longer computation time, as shown in figure 4. We cannot just use the slack by sending the message directly to the other tiles because the message injection is bound to the communication schedule. Therefore, in such a local adaptation the goal is to optimize the time slot given by the schedule. This would save energy, but the other cores do not profit from this approach.

In systems with interdependent applications, the later tasks can also profit from slack. Our approach is to use the slack where the energy savings are best: in this case, the slack would be passed to the other four tiles, as figure 5 shows. These 4 tasks could be started right after the first one finishes, therefore giving them more time to execute. The local tasks then can optimize their newly assigned timeslots using the local DVFS. This way we can multiply the saved energy by four. Not only the energy savings are great for the energy
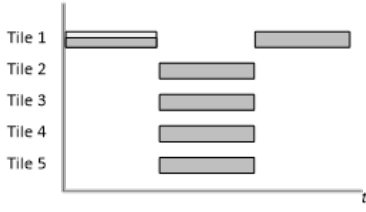
Fig. 4. Local energy management

usage aware system. Considering that especially in mixed criticality systems highly critical tasks are planned with a very pessimistic WCET assumption, these slack times unnecessarily block resources for other applications. Reconfiguring these access times on-line is another advantage of our approach.
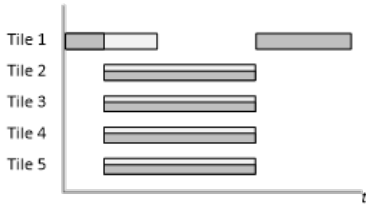

Fig. 5. Global energy management

B. Global adaptation

The changes described above can only be realized when the communication schedule is adaptable. The message of tile 1 needs to be transmitted at the end of the execution to enable the other cores executing their tasks. To do so, the NI's schedule must be changed. Considering that all NIs control the message injection locally, it is crucial that all NIs change their schedule at the same time. Otherwise it is possible that two NIs stay in incompatible schedules which may cause unnecessary re-transmissions due to missed messages and collisions, when a tile is gated while another core wants to send some messages. These inconsistent behavior raises the systems energy usage and can cause deadline misses, leading to system failure.

To avoid such conditions our approach is to define a meta-schedule according to which all changes in the NIs can be done. This meta-schedule is a state machine where each node is a verified schedule that by itself would already define all executions of the system. Additionally to this baseline schedule we compute variations of the schedule that optimizes the task execution and message injection according to a context. For example if we are in the base schedule state and a slack event is monitored this will be interpreted as a system context and the state machine will change into a schedule that is adapted to this slack.

Looking at figure 3 and figure 4 the first schedule can be assumed to be the baseline schedule. As soon as the event of the detected slack happened the meta scheduler changes the system schedule to the second schedule. This approach realizes a local adaptation which is based on a global knowledge. For

the design of the adaptation we assume that such a global knowledge is available. By using a local adaptation we avoid implementing a dedicated global energy manager which would cause energy overhead for the system.

The context is a set of environmental and task specific information that can be exploited to improve the system energy consumption or indicate that a safe state with minimum energy consumption must be entered. Examples for this information are the slack of tasks, the temperature and the energy levels. If for example the energy is low a low-energy state can give a photovoltaik adapter the chance to re-charge to a minimum threshold energy level before re-entering the normal execution schedule. The context information can be designed system specific and therefore reduce or expand the scheduling problem.

C. Modeling of global adaptation

To perform the changes the meta schedule need access to the NI's schedule and must be able to change it directly. Figure 6 shows the new network interface. The meta scheduler needs to store the meta schedule with the triggering global context information. It needs to save all schedules that its local NI can assume to be able to place them in the NI. To reduce the memory space needed for this NI block we reduce each NIs schedule knowledge to the subset of scheduling decisions it is involved in. This way each NI only stores a fraction of the whole schedule. Only the global context information must be stored in whole to ensure that each local decision is unanimous. The decision will be made based on the global information, to assure the global optimum and to avoid being trapped in a local peak. The schedule memory space will further be optimized by saving a baseline schedule which is stored as a whole. All other schedules will be stored as a Δ to the previous schedule. This way not only the memory space for the schedules will be reduced but also the schedule change operation can be streamlined by *binary adding* the new schedule on top of the current one. These Δ schedules are dependent on their parent schedule, therefore they will be saved in a treelike structure that will be traversed in parallel to the traversal of the meta schedule.

Additionally the meta schedule has a core interface via which it can send commands to the hypervisor informing him about the new deadlines. This adaptation can be understood as an update of the time contract defined by the time interfaces between the computational and communication schedule. As these schedules are interdependent any change on the communication side will have direct effects on the computation either prolonging the available execution time or shortening it.

D. Meta schedule requirements

The potential schedule space of such an approach is infinite. If one were to mimic a dynamic energy management each fine grained context change would lead to a dedicated new schedule. Such a system would collide with our initial plan to save energy, as the management would consume more energy
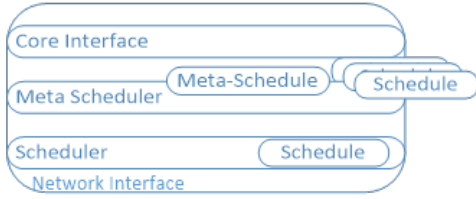
Fig. 6. New adaptive Communication Interface

than the actual computation. Therefore we design two kinds of state boundaries:

- discrete context intervals
- forced convergence of the schedules

E. Discrete context intervals

The first boundary is a minimization of the context that the system can react to. If certain executions times are passed the slack is too small to adapt him globally, these intervals are ignored by our system and the local adaptation can use it. Within the viable slacks we define discrete intervals, e.g. if the monitored slack is within 50%-30% we will jump into a new optimized schedule. Figure 7 shows how a baseline schedule can lead into 3 possible slack optimization states.

The boundary for the context can be set according to the designed system. If many other context variables are already observed, the slack options can be reduced. If the context will be defined solely by slack, more specific slack options and reactions can be modeled. We keep our system intentionally as generic as possible to be applicable to as many applications as possible.

F. Schedule convergence

The second way to reduce the potential schedules is to force a convergence of the schedules by bounding the amount of tasks that can profit from. To do that we must look at the different ways the meta-scheduler can evolve:

1) the context implies that the whole system mode has to change to assume a low energy mode, e.g. battery status
2) the context implies that the a advantage can be exploited by changing the way the following tasks are executed, e.g. DVFS, gating specific areas for some time

The additional states generated by the first context type is just one, because when a low battery status will be reached the whole system will go into a low power mode defined with a low power schedule. There will be no further changes in the state to save as much energy as possible and prolong the systems lifetime until maintenance can recharge the battery. If a rechargeable battery is used the low power mode can be left and a normal schedule can be applied when a satisfying threshold of battery capacity is restored. In the low power state itself the adaptive services will be deactivated to save as much energy as possible.

The second context type is generated by suddenly appearing *windows* that can be used to apply low power techniques. Theoretically such additional time can be used by all following

tasks, leading to exponential growth of the meta-schedule states. Bounding the states by restricting the amount of tasks to a value k that can benefit from such a context change, we can force the schedules to converge after k hops. This way may also bound the impact the changes can have. Additionally, it may use e.g. slack times as soon as possible, rather than where they could be used best. But, with the volatile system context we cannot assume that the best task to benefit from slack times at t_0 will also be the best choice at t_2.

As an example we look at a schedule which can be changed based on three slack options. If each job can pass three possibilities of slack s to the next ones, then one can see that the resulting tree grows exponentially according to the number of states and jobs (s_j). The problem is not scalable in this way, therefore the passing on is bounded to the next two jobs ($c = 2$). Then, the tree grows in the same way as before until job three has been reached. At job four, the convergence occurs. Regarding each subtree of the nodes of level J_1, the following paths are equal for each of the three nodes. If J_1 passes possibility $S"_1$ to J_2 and J_2 passes $S"_2$ to J_3, the path in the tree is

$$S_0 \rightarrow S_0 S"_1 \rightarrow S"_1 S"_2 \tag{1}$$

which is the left path in the subtree. The paths for the other subtrees are similar:

$$S_0" \rightarrow S"_0 S"_1 \rightarrow S"_1 S"_2 \wedge S_0' \rightarrow S_0' S"_1 \rightarrow S"_1 S"_2 \tag{2}$$

all ending in the same state $S_1 S"_2$. Regarding the figure, this is also true for the other states in layer J_3 printed in red. Each of those states has a related state in the subtree of S_0.

We can formulate the state expansion with a formula:

$$N = \sum_{c-1}^{i=0} s^i + s^c(j - c) \tag{3}$$

Until level $c - 1$ is reached with c being the convergence constant, the total number of states can be calculated as the sum over s_i. The variable s represents the number of supported context changes. As soon as level c has been reached, the number of possible states on each level is equal to sc. Then, the number of jobs can be reduced by the convergence constant as they are already regarded in the exponential part. The resulting number of residual jobs can be multiplied with the number of states per level.

At runtime the current schedule is known to the network interface where according to the scheduled message injection times the serialization units injects the time-triggered messages onto the NoC. Our NI receives the context information, decides on changes and manipulates this particular schedule into the new one.

V. EXPERIMENTAL SETUP

This section focuses on the simulation we used to test our assumptions and its results.
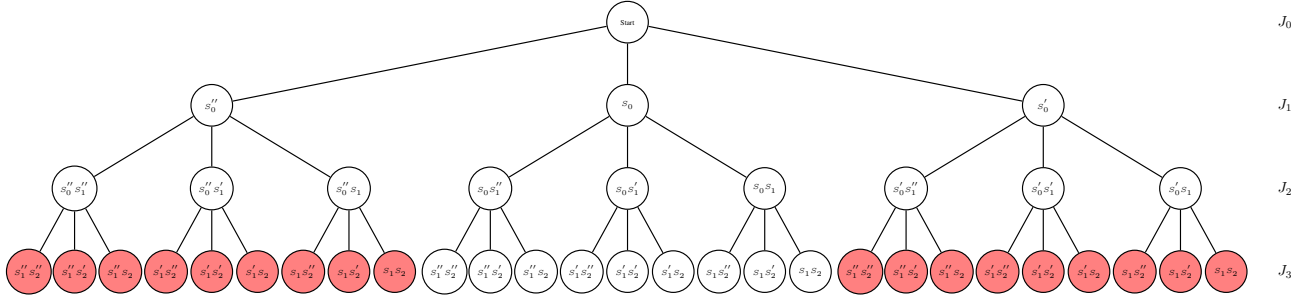
Fig. 7. Bounded expansion of possible states

A. Gem5 Simulation

We used Gem5 [13] to verify our assumption. We set up a simulation based on the use case that we described in figure 8. We used a 2x2 Mesh Noc with 4 tiles connected to it. The nodes A and B are located in the northwest and southeast of the the NoC and we used x-y routing to determine the message paths. Figure 8 shows that we have two applications, the tasks t_0, t_1, t_2 and t_3 are high critical tasks while t_4 and t_5 are low critical. We also modeled inner-tile and inter-tile dependencies to proof our slack propagation assumption: t_2 depends on t_0 and t_3 depends on t_2 and t_3. In the figure, the numbers in the tiles and those associated to the messages are their assumed WCET time for computation and message transmission.

Because Gem5 does not support DVFS, a simpler model must be applied to measure the energy consumption in the NoC. We modeled our power model based on the known behavior of CMOS-circuits. We used them to approximate the consumed power in the NoC. As shown there, the consumption depends on the frequency among others. However, the frequency the messages are sent with shall be set by the adaptive communication at defined instants. It is assumed, that the input and the output frequencies in equation 4 are similar such that the dynamic power consumption can be calculated by equation 5

$$P_D = P_T + P_L = C_{pd} \cdot V_{CC}^2 \cdot f_I \cdot N_{SW} + C_L \cdot V_{CC}^2 \cdot f_O \cdot N_{SW} \tag{4}$$

$$P_D = \left(C_{pd} \cdot V_{CC}^2 \cdot N_{SW} + C_L \cdot V_{CC}^2 \cdot N_{SW} \right) \cdot f \tag{5}$$

In the simulation, the factor in braces is not exactly determined but approximated by a constant. The value is set to 1 to simplify the model. At each tick, the constant is scaled by the current frequency in the NoC and added up.

We modeled a power model including idle and running states of the tiles for the normal and the low power frequency. To compute the overall energy consumption, the associated energy value is added to a global energy counter each tick. At the end of the simulation, the energy counter is shown. The traces that were used on the tile are created by randomly scaling the WCET to simulate the unpredictable tile behavior. For the results, we have created traces with low slack and high

slack. We compare our results to a baseline configuration. In this configuration, we let the tile run in idle mode until the WCET is finished.

Our adaptation block has a text file representing the schedule-changes which it traverses based on its current state. For the adaptation, the adaptive communication adds the Δ value in the row of the current schedule-change in the Gem5 config-file.
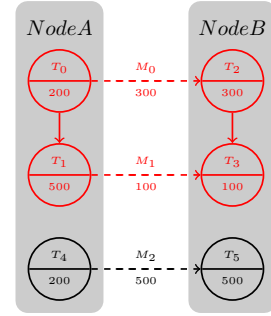

Fig. 8. Use case

B. Simulation results

The simulations of the adaptive communication in this section show the correct behavior and the possibility of saving energy. For this, the system is configured according to the use case shown in figure 8 as shown in the previous section. There are four different simulations presented with an approximation of the resulting energy consumption. The tile's message injection is defined such that in the first case there is much dynamic slack generated and few slack in the second one. Switching off the adaptive communication in the other cases enables a comparison with the baseline NI.

Each table printed in the following shows the message statistics for the cases. The first two columns show the message ID and the total communication delay from dequeuing from the source Core and enqueuing in the destination Core. Columns three and eight show the de- and enqueuing instants of the Cores. Time-triggered messages are enqueued in the priority queues unit (PQU) by the NI according the schedule. While this instant is printed in column five, the next column shows when the message is dequeued. As the guarding window is enabled, the message can directly be enqueued into the NoC.

This instant is presented in the sixth column. On the other hand, the seventh column contains the instants when the last flit of the message leaves the NoC. In each case, at first all messages are sent in the first application mode with all tasks enabled. This mode is represented by the first lines with the IDs 0, 200 and 100. Afterwards, the application mode changes at the end of the period and the low-battery mode is executed. Then, only messages 0 and 100 are sent.

Msg ID	Total Delay	Deq Core	En PQU	Deq PQU	En NoC	Deq NoC	En Core
0	211	8	200	201	201	210	219
0	68	151	200	201	201	210	219
200	29	390	400	401	401	410	419
100	29	890	900	901	901	910	919
0	101	1618	1700	1701	1701	1710	1719
100	101	2118	2200	2201	2201	2210	2219

TABLE I
MESSAGE STATISTICS FOR BOTH CASES WITH ADAPTATION DISABLED

Table I contains the message statistics when the adaptive communication is disabled. Because the cases of much and few slack only differ in the total communication delay and the instant of dequeuing the Core, both simulations are shown in the same table. According the schedule, in the first mode the messages are sent at instants 200, 400 and 900. The total communication delays for message zero show, that due to dynamic slack the delays are much greater than those of messages 200 and 100 which nearly require their WCET. This time can be used to apply power saving techniques. Changing the application mode at the end of the period occurs at instant 1500. Therefore, the phases of the safety critical messages 0 and 100 are 200 and 700.

Msg ID	Total Delay	Deq Core	En PQU	Deq PQU	En NoC	Deq NoC	En Core
0	22	8	11	12	12	21	30
200	32	390	403	404	404	413	422
100	29	890	900	901	901	910	919
0	101	1618	1700	1701	1701	1710	1719
100	101	2118	2200	2201	2201	2210	2219

TABLE II
MESSAGE STATISTICS FOR MUCH SLACK WITH ADAPTATION ENABLED

In the following, the adaptive communication is enabled. At first, table II presents the case with much dynamic slack. Message 0 is available at the Core at tick 8 resulting in slack of 192 ticks. The matching interval end is tick 10 which enables the sending at tick 11 instead of tick 200. Since the gem5 version used for the implementation does not support DVFS, the communication delay is not elongated and the message arrives at the destination at tick 30. However, DVFS would result in a longer communication delays such that message 200 had to be sent 3 ticks later at tick 403. At message 100, the convergence leads to a transmission at the specified instant. In the low-battery mode, all messages are sent with the lowest supported frequency and no further changes due to dynamic slack are specified.
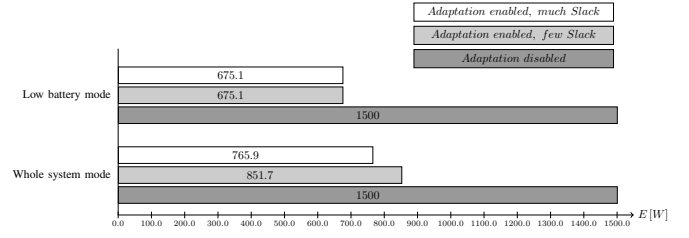

Fig. 9. Energy consumption in simulation

Table III on the other hand shows the case with few dynamic slack. Message 0 is dequeued from the Core at tick 151. Then, the matching schedule change in this case triggers the transmission at tick 166 and delays the next message by 33 ticks.

Msg ID	Total Delay	Deq Core	En PQU	Deq PQU	En NoC	Deq NoC	En Core
0	34	151	166	167	167	176	185
200	62	390	433	434	434	443	452
100	29	890	900	901	901	910	919
0	101	1618	1700	1701	1701	1710	1719
100	101	2118	2200	2201	2201	2210	2219

TABLE III
MESSAGE STATISTICS FOR FEW SLACK WITH ADAPTATION ENABLED

Figure 9 shows the resulting power consumption. The x-axis shows the consumed energy in watt while on the y-axis, the different application modes are presented. In each mode, the case with the adaptive communication disabled is printed in dark gray. On the other hand, the case with few slack is colored in light gray and the one with much slack in white.

Since all activities are executed with a frequency of 100% when the adaptive communication is disabled, the consumed energy amounts to $1500W$. In the whole system mode, the frequency the messages are sent with is adapted when the message transmissions start. As shown in the chart, the adaptation enables power savings of $648.3W$ for few respectively $734.1W$ for much slack which equals 43.22% respectively 48.94% of the consumed power when the adaptation is disabled. On the other hand, in the low battery mode the frequency is set to 45% at phase zero and not changed for the complete period. This results in an energy consumption of $675.1W$ independent from the available dynamic slack which corresponds to 45% of the consumed energy without adaptation.

The results show, that adapting the communication at run-time to apply low-power techniques has the potential to save energy of more than 50% if there is sufficient slack available. However, the simulation does not consider the reduction of the supply voltage. Because there is a linear relationship between frequency and supply voltage and a cubic relation between frequency and power, adjusting the voltage can lead to higher energy saving in future optimizations.

VI. Conclusion

We presented our approach to realize a dynamic system adaptation for pre-planned systems. With our approach we can guarantee timing boundaries for real-time systems and support mixed-criticality-applications while still apply dynamic energy oriented reconfiguration to prolong the system lifetime. For a first step we have implemented the context as a slack monitor, but we plan to expand the context to further energy saving possibilities. These adaptation techniques can also be used to change the system behavior in cause of faults and dynamically reconfigure the on-chip communication to cope with node or link faults without needing dynamic routing algorithms. In future work, we will define a protocol to consolidate the global knowledge about the context which we assumed to be available in this paper.

VII. Acknowledgement

References

[1] A. Burns and R. Davids. *Mixed Criticality Systems A Review*, 2016.

[2] Hong, Inki, et al. *"Power optimization of variable-voltage core-based systems."* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18.12 (1999): 1702-1714.

[3] Das, Anup, et al. *"Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs."* 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2014. IEEE, 2011.

[4] J.J. Chen and C. Kuo, *"Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms".* in Proc. International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2007, pp. 28-38.

[5] C. Li and P. Ampadu, *Energy-efficient NoC with variable channel width* in 2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS), 2015, pp. 14.

[6] P. Zaykov, G. Kuzmanov, A. Molnos, K. Goossens, *"Run-time Slack distribution for real-time data-flow applications on embedded MPSoC".* in Proc. of DSD, 2009, pp. 409-418.

[7] Paterna, Francesco, Andrea Acquaviva, and Luca Benini. *"Aging-aware energy-efficient workload allocation for mobile multimedia platforms."* IEEE Transactions on Parallel and Distributed Systems 24.8 (2013): 1489-1499.

[8] S. Ogrenci Memik, E. Bozorgzadeh, R. Kastner and M. Sarrafzadeh, *A super-scheduler for embedded reconfigurable systems* in IEEE/ACM International Conference on Computer Aided Design 2001. ICCAD 2001, pp. 391-394.

[9] Persya, A. Christy, and TR Gopalakrishnan Nair. *"Model based design of super schedulers managing catastrophic scenario in hard real time systems."* Information Communication and Embedded Systems (ICICES), 2013 International Conference on. IEEE, 2013.

[10] H. Ahmadian and R. Obermaisser, *Time-Triggered Extension Layer for On-Chip Network Interfaces in Mixed-Criticality Systems* in Euromicro Conference on Digital System Design (DSD), 2015, pp. 693-699.

[11] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, et al., *"Building timing predictable embedded systems".* in ACM Transactions on Embedded Computing Systems (TECS), 13(4):82, 2014.

[12] A. Molnos and K. Goossens, *"Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors".* in Proc. of DSD, 2009, pp. 409-418.

[13] Binkert, Nathan, et al. *"The gem5 simulator."* ACM SIGARCH Computer Architecture News 39.2 (2011): 1-7.