Consolidate-to-Idle
The second dimension is almost for free.

Marcus Völp, Johannes Steinmetz, Marcus Hähnel
Institute for Systems Architecture, Operating Systems Group
Technische Universität Dresden
Dresden, Germany
{voelp, mhaehnel}@os.inf.tu-dresden.de

***Abstract*—Following time, energy is the second most important resource developers of real-time and embedded systems have to consider. Many solutions have been proposed for exploiting slack along the time axis, e.g., to improve the responsiveness of aperiodic jobs or to reduce the processor frequency and supply voltage. In this work, we focus on the spatial dimension. Our goal is to consolidate a given real-time workload to as few cores as possible to keep the majority of system resources powered off until their unavailability risks missing a deadline. Our proposed scheme is completely independent of the actual multiprocessor scheduling policy and also independent of the actual method for computing slack as long as they reveal the original processor allocation and the slack that is available on each CPU. We started implementing a partitioned EDF based scheduler for Linux using our consolidation scheme. This scheduler will enable us to demonstrate expected energy savings and can be used to evaluate different, consolidation-based scheduling policies.**

Keywords*—*energy; slack time; scheduling; many-core; real-time

I. Introduction

Energy is a vital resource of real-time and embedded systems, outweighed only by the demand to meet the deadlines of admitted tasks. Many solutions have been proposed to dynamically adjust the speed of processing resources and, as a consequence, the supply voltage while preserving the systems' real-time properties (see e.g., [1]–[3]). However, when we continue to scale to smaller technology nodes, leakage will dominate the energy consumption and the returns from dynamic voltage and frequency scaling will diminish [4]. Clock-gating and the implied disabling of hardware resources will become the dominant mitigation strategy. Some computer architects even predict that 50 % of a fixed-size chip cannot be kept powered up for long periods of time [5] once we have reached 8 nm and below.

This paper presents our early results on a slack-based energy-aware scheduler. Unlike previous approaches, our scheduler does not primarily seek to optimize along the time dimension but instead focuses on the spatial dimension. Our goal is to consolidate a given real-time workload on as few cores as possible, powering up additional resources only if we would otherwise risk missing a deadline. In accordance to race-to-idle [6], we call this approach *consolidate-to-idle*.

To our surprise, consolidate-to-idle needs to know only the current assignment of tasks to CPU cores as it is produced by an arbitrary multiprocessor scheduler, which we call the *underlying scheduler*, and the slack time that is available on each of these cores. In all other respects, consolidate-to-idle is completely independent of this multiprocessor scheduler and of the actual method used for computing this slack. In this respect, once this original scheduler has produced a schedule and computed the slack, consolidate-to-idle comes almost for free. The only two costs that we have to consider are the time required to power up additional CPUs and the overheads that occur when migrating tasks more often than the underlying scheduler.

To enable an in depth evaluation of our approach, we are currently implementing a partitioned EDF (pEDF) scheduler for Linux with consolidation functionality. While the scheduler was not finished in time for this publication we hope to soon be able to present first results of *consolidate-to-idle*.

II. Foundations

To explain consolidate-to-idle, let us start with a rather abstract definition of schedulers. A scheduler decides when and where to run the tasks τ of the task set T. We characterize the behavior of such an arbitrary scheduler by the mapping S, which selects for each point in time t and for each CPU m the task that runs on this CPU. We use the special symbol $\diamond$ to denote that no task is selected and that the CPU is idle at time t. Tasks τ can typically react to previous scheduling decisions (e.g., by yielding until the release of their next job if they did run long enough to complete the current job). We characterize this behavior by a not further specified function ϕ, which given the previous scheduling decisions returns how each task would react (i.e., run, block or yield until the next release), and evaluate ϕ and S in an interleaved fashion. That is, ϕ takes the schedule up to time $t-1$ to produce a reaction at time t and S picks a runnable task or returns $\diamond$ to indicate an idle CPU in the schedule for time t. If S produces such a mapping for task τ at time t (i.e., $S_\phi(t,m) = \tau, \tau \neq \diamond$), we call the CPU m to which S has assigned τ the *home CPU* of this task at time t and write $m = home_{S,\phi}(\tau, t)$.

The amount of time by which the schedule may be deferred before tasks risk missing their deadlines is called the *slack* or slack time σ. Obviously, the slack varies over time as jobs of tasks complete or as we defer the execution of the tasks selected by the underlying scheduler. We write $\sigma(t,m)$ to denote the slack available at time t on CPU m. In other words, if at time t, we have slack $\sigma(t,m)$ on CPU m then we may defer from $S_\phi(t,m)$ until time $t+\sigma(t,m)$ to run other tasks. Notice that $S_\phi(t,m)$ may differ from $S_\phi(t+\sigma(t,m),m)$, for example, because the job of the former task may already be

completed. However, as long as $\sigma(t,m)$ correctly represents the slack, continuing with $S_\phi(t+\sigma(t,m),m)$ will not result in deadline misses.

Our approach is to run τ either during the slack-time of some consolidating CPU or on its home CPU according to $S_\phi(t,m)$. This way, feasibility of consolidate-to-idle follows immediately from the feasibility test for the underlying scheduler respectively from the correctness of the slack computation. In this way, consolidate-to-idle is independent of this underlying scheduling and slack computation method.

For example, a partitioned EDF (pEDF) scheduler assigns each task a fixed CPU m and returns for m the task τ whose current job has the earliest deadline. If we assume periodic tasks τ_i with implicit relative deadlines $D_i := P_i$, characterized as usual by tuples $\tau_i := (P_i, C_i)$ with period P_i and worst-case execution time C_i, $S(t,m)$ will return τ_i if $t \leq r_i + D_i$ (where r_i is the point in time when τ_i's current job was released), if there is no task τ_j with $r_j + D_j < r_i + D_i$, if τ_i is runnable (i.e., $\phi(S^{t-1}, \tau_i, t) = run$), and if τ_i's remaining execution time $e_i \leq C_i$ is positive.

Tia's [7] static slack computation algorithm is one method for determining slack in pEDF. Without loss of generality let jobs indices be sorted by their absolute deadlines (i.e., for two jobs J_j, J_k, $r_j + D_j < r_k + D_k \Rightarrow j < k$). Starting from a precomputed slack table for each CPU, Tia partitions the periodic jobs that are in the current hyperperiod and that have deadlines after t into subsets of jobs with deadlines between $r_{c_i} + D_{c_i}$ and $r_{ci+1} + D_{c_{i+1}}$, where c_i is the current job of task τ_i. The cells $w(j,k)$ of the precomputed slack table denote the minimum of the initial slack $\sigma_l(0,m)$ of all periodic jobs τ_l with deadlines in the range $[r_j + D_j, r_k + D_k]$. That is,

$$\sigma_j(0,m) = r_j + D_j + \sum_{\{k | r_k + D_k \leq r_j + D_j\}} C_k \qquad (1)$$

and

$$w(j,k) = \min_{r_j + D_j \leq r_l + D_l \leq r_k + D_k} \sigma_l(0,m). \qquad (2)$$

The static slack $\sigma(t,m)$ at time t by which the schedule of CPU m may be deferred is given by

$$\sigma(t,m) = \min_{1 \leq i \leq n} w_i(t) \qquad (3)$$

where

$$\begin{aligned} w_i(t) &= w(c_i, c_{i+1} - 1) - I - ST - \sum_{k=i+1}^{n} \xi_{c_k}, \\ w_n(t) &= w(c_n, N) - I - ST. \end{aligned}$$

In this equation, $n = |T_{par,m}|$ is the number of tasks allocated to CPU m, N is the number of the jobs of these tasks in the hyperperiod, I is the total idle time on CPU m relative to the beginning of the current hyperperiod, ST stands for the time stolen since the beginning of this hyperperiod and ξ_k is the completed portion of J_k.

III. Consolidate-to-Idle

The idea behind consolidate-to-idle is rather simple: exploit slack time on a few active cores to run the real-time tasks allocated to other cores, powering up additional cores only to prevent tasks from missing their deadlines. For the remaining discussion, we consider CPUs to be in one of two possible states: *consolidating* and *passive*. Consolidating CPUs run tasks of one or more passive CPUs as long as the slack on these CPUs permits. Once the slack is used up, they run the tasks that the underlying scheduler has assigned to them. Passive CPUs run no task but idle or enter a deep power saving state. In this work-in-progress report, we consider only identical CPUs although we see a huge potential for consolidate-to-idle in a heterogeneous setup. For example, extending the general idea of big-little architectures (as proposed by Kumar et al. [8]) to predictability, we may schedule the real-time workload on the predictable cores and then consolidate them on more efficient but unpredictable cores [1].

A. Passive CPUs

Consolidating CPUs may keep other CPUs passive as long as they finish the tasks of these passive CPUs quicker than their slack time runs out. Let SU_m be the time required to start up CPU m after it has entered a power saving state. Let further $M_m^{passive}(t)$ be the costs for migrating tasks at time t to the passive CPU m. Then clearly, we have to start powering up a passive CPU latest at

$$t_{migrate\_home} := t + \sigma(t,m) - SU_m - M_m^{passive}(t). \qquad (4)$$

Because real-time tasks tend to complete well before their worst-case execution times, it is likely that all tasks will complete on the consolidating CPUs and that start up and migration to passive home CPUs will not be necessary most of the time.

The term $M_m(t)$ in Eq. 4 is to prepare for cache aware worst-case execution time analyses. Typically, migration costs are characterized on a per task basis as direct and indirect costs. Direct costs is the time required to stop and to restart it on its destination CPU. Indirect costs is the time required to transfer state that the task requires but that is not immediately transferred with the task. Most notably these costs are the cache transfers of the task's working set. Consolidation typically affects more than just the migrated task. On the consolidating core, the task may have disturbed the cache working set of local tasks (i.e., tasks whose home CPU is this consolidating core). Let $M_{m_c}^{active}(t)$ capture these costs. On the formerly passive CPU, consolidated tasks had no chance to fetch cachelines required by subsequent tasks. At the same time, they had no chance to interfere with these tasks. Whether $M_m^{passive}(t)$ is positive or negative therefore depends on the consolidated tasks and on the tasks that follow after time t in the schedule of CPU m. Of course, we will have to look at reasonable approximations of $M_{m_c}^{active}$ and $M_m^{passive}$.

As it is our goal to keep passive CPUs off most of the time, computing the slack times and triggering the startup of passive CPUs has to be part of the responsibility of the consolidating CPUs. In this case, we say such a consolidating CPU m_c *observes* the slack of a passive CPU m and collect all CPUs m that m_c observes in the set $O(m_c)$.

B. Consolidating CPUs

In general, the task set T may include real-time tasks whose home CPU is a consolidating CPU m_c. In this case, we must

[1] We assume here that it is possible to limit the interference between these cores in the on-chip network and memory blocks.

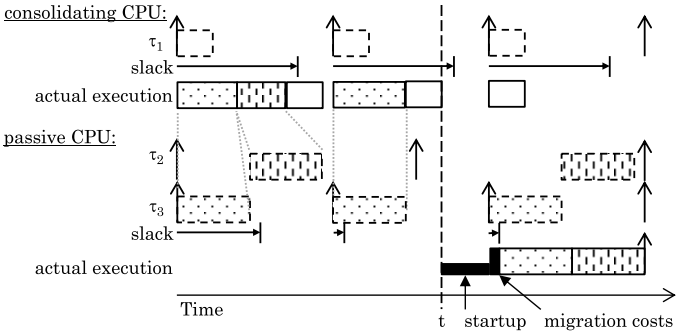

Fig. 1. Tasks of the passive CPU are consolidated to the consolidating CPU. At time t, the slack time on the passive CPU m falls below $SU_m + M_m^{passive}(t)$. We need to power up this CPU.

also consider the slack $\sigma(t, m_c)$ of the consolidating CPU. That is, to not risk missing a deadline, the consolidating CPU must start executing local tasks latest at $t + \sigma(t, m_c) - M_{m_c}^{active}(t)$.

To reduce synchronization and management overhead, we maintain the invariant that a consolidating CPU only runs tasks from passive CPUs in its slack time and only local tasks if no more slack is available. When activating a passive CPU, we already consider the migration overhead of all local tasks that may have been relocated to other consolidating CPUs. Also, we do not have to consider the migration overhead of tasks that we consolidate on such a consolidating CPU because we directly deduce feasibility from the local and remote slack and from the admission for the underlying scheduler. Instead, we execute tasks that we consolidate as fast as we can to cause them to make as much progress as possible.

In general, it would be possible to further defer the activation of passive CPUs if the slack on consolidating CPUs suffices to complete the scheduled passive CPU before its deadline. In our preliminary analysis, we did not consider this possibility. In non-identical settings, exploiting this option requires reevaluating the feasibility of the passive CPU's schedule, an overhead we seek to avoid.

Fig. 1 illustrates our approach on the example of three tasks $\tau_1 \ldots \tau_3$. Shown is the EDF schedule (dashed boxes) and slack (horizontal arrows) as determined by the underlying scheduler. Also shown is the actual execution that happens on the respective CPUs. The CPU slack suffices to consolidate the jobs $\tau_{2,1}$, $\tau_{3,1}$ and $\tau_{3,2}$ together with τ_1 on the consolidating CPU. At time t, we have to start powering up the passive CPU to be ready to execute $\tau_{2,2}$ and $\tau_{3,3}$ on their home CPU. Their execution is prolonged by the migration overhead $M_m^{passive}(t)$. If during τ_1's first period, $\tau_{2,1}$ would have executed longer, $\tau_{1,1}$ would have preempted it at the point in time denoted by the end of the first horizontal arrow when the local slack on the consolidating CPU would be depleted.

C. Algorithm

Before we proceed by sketching the algorithm for consolidate-to-idle, it is helpful for a deeper understanding to cast the above operations into the terminology of Tia's slack computation method (Eq. 3). At consolidating CPUs, we steal the slack of local tasks to run tasks of passive CPUs. Hence, we increase the stolen time ST at the consolidating CPU. On the passive CPUs, we increase the idle time I by keeping them passive. However, at the same time we also complete jobs J_k, whose home CPU is passive, by consolidating them.

```
1  // invoked at the usual scheduling events,
2  // in case σ(t, m_c) = 0 and when starting up
3  // a CPU.
4  schedule:
5    // update statistics for current job:
6    adjust progress of the current job J_k;
7    if (home(J_k) ∈ O(m_c)) then
8      recompute the slack time of home(J_k);
9      setup a timer to
10       σ(t, home(J_k)) − SU_home(J_k) − M^passive_home(J_k)(t);
11   endif
12   recompute the local slack σ(t, m_c);

14   // check whether we need to run local jobs:
15   if (σ(t, m_c) = M^active_m_c(t)) then
16     setup the usual timers (e.g., deadline);
17     switch to S_φ(t, m_c);
18   endif

20   // check whether to deactivate:
21   if (become_passive()) then
22     distribute observed CPUs to the sets
23       O(m_c) of other consolitating CPUs;
24     set state to passive;
25     enter sleep state();
26     goto schedule:
27   endif

29   // keep track of local slack:
30   setup timer to σ(t, m_c);

32   // consolidate task of passive CPU:
33   pick τ = S_φ(t, m') from some passive CPU m';
34   switch to τ;

36 // invoked in case σ(t, m) = SU_m + M_m(t):
37 startup_passive:
38   set state of m to active;
39   adjust observation of passive CPUs;
40   startup CPU m;
41   let J_k be the current job on m_c;
42   if (home(J_k) = m) then
43     schedule();
44   endif
```

Fig. 2. Pseudo code of the central functions of consolidate-to-idle. The respective entry points are the call backs of the timers setup to react to slack depletion. As before t is the current time and m_c denotes the CPU executing this code.

Fig. 2 contains pseudo code describing the basic operation of consolidate-to-idle. Lines 5–11 adjust the slack of the consolidating CPU. In addition, if the consolidating CPU has executed a job of a passive CPU, we must also adjust the slack of this CPU. Timeouts are set to the time when the slack of an observed CPU m reaches $SU_m + M_m^{passive}(t)$ because latest at this time, the observing CPU has to start activating m. Lines 14–18 execute local jobs if no more slack remains on the consolidating CPU by switching to the current job selected by the underlying scheduler. The function `become_passive()` determines whether a CPU should become passive or remain

active as a consolidator. In our example, this function always returns false for CPU 0 and true for all others. In general, more sophisticated strategies are imaginable, which consider the number and distance to passive CPUs and the heat that has been building up. In line 33, we pick a task τ to consolidate on m_c if such a task exists. The strategy for selecting tasks from passive CPUs is a further dimension in the design space of consolidate-to-idle that is worth exploring in the future. Sec. III-D1 gives some hints on the implied synchronization overhead. However, a thorough discussion of this point is out of the scope of this work-in-progress report. For our example implementation, we plan to use a Round-Robin like approach, which cycles through passive CPUs picking the current task (if present) and runs it to completion or until some slack is depleted.

D. Practical matters

We would like to raise attention to two practical matters: the interrelationship between the consolidation strategy and run-queue synchronization and the possibility to integrate other slack-based approaches.

1) Run-queue synchronization: Clearly, if multiple consolidating CPUs pick from the same passive CPUs, global synchronization on the run queue of this passive CPU is needed to prevent different consolidating CPUs from picking the same task. Notice that because we deactivate a CPU if it is passive, it will not modify its local run queue. The queue may therefore be accessed by a single consolidating CPU without further precaution or overheads (assuming passive CPUs stay offline most of the time)[2]. At the same time, strategies picking multiple tasks from the same passive CPU are faster in creating more slack on these CPUs by increasing the completed portion of the jobs they execute on their stolen time.

2) Integration of other slack-based approaches: Consolidation decisions are only based on the home CPUs of tasks and on the slack $\sigma(t, m)$ that the used slack-computation method reports to the consolidation algorithm. Reducing this slack by reporting only the time that is left after considering other uses (e.g., executing aperiodic tasks) is an easy way to integrate other slack-based approaches. Ideally, these approaches are aware of the consolidation strategy and report reduced slack only on consolidating (i.e., active) CPUs or they influence the scheduler S in that it reports also the use of this slack, for example, in terms of servers on the corresponding CPUs whose budgets are used to execute aperiodic tasks.

In these integrated scenarios, where slack needs to be determined anyway, the overheads of consolidate-to-idle are limited to the power up times of passive CPUs and to the overheads for migrating tasks from consolidating to recently activated CPUs. In this sense, consolidate-to-idle comes almost for free.

IV. Conclusions and Future Work

This paper has shown early results of consolidate-to-idle, a slack-time based approach to save energy in manycore systems by disabling inactive CPUs, aggregating their real-time tasks to active consolidating cores. We have seen that consolidate-to-idle is completely independent of the underlying scheduler and slack computation method provided they reveal the home CPU of all current jobs and the slack that remains before these jobs must be run. Of course, many directions in the design space of consolidate-to-idle remain open. To avoid contention on the run queue, we have chosen an ad-hoc $n : 1$ association of passive CPUs to consolidating CPUs. Other strategies, such as balancing the slack of passive CPUs by picking possibly several tasks from the one with the smallest slack to consolidate them on different CPUs or picking a task with hot cache working set on the consolidating CPU, might further defer when passive CPUs need to be powered up. Other dimensions include decisions on when consolidating CPUs should become passive and heterogeneity between the resources. Despite these open questions, we confidently believe that there is a case for the spatial exploitation of slack time and that consolidate-to-idle is a first step in this direction.

[2]Notice that local synchronization may still be necessary to avoid races with code that re-enters unblocking tasks to the run queues.

Acknowledgments

This work is in part funded by the DFG through the collaborative research center "Highly Adaptive Energy Efficient Systems" (HAEC) and through the cluster of excellence "Center for Advancing Electronics Dresden" and by the EU and the state Saxony through the ESF young researcher group "IMData". Thanks to Michael Roitzsch for the insightful discussions.

References

[1] J.-J. Che, C.-Y. Yang, and T.-W. Kuo, "Slack reclamation for real-time task scheduling over dynamic voltage scaling multiprocessors," in *Sensor Networks, Ubiquitous, and Trustworthy Computing*, vol. 1, June 2006, p. 8 pp.

[2] R. Jejurikar and R. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *Proceedings of the 42nd annual Design Automation Conference DAC '05*, New York, NY, USA, pp. 111–116.

[3] D. Zhu, R. Melhem, and B. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 7, pp. 686–700, 2003.

[4] E. L. Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *2010 Workshop on Power Aware Computing and Systems (Hot Power'10)*. Vancouver, Canada: Usenix.

[5] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376.

[6] S. Albers and A. Antoniadis, "Race to idle: new algorithms for speed scaling with a sleep state," in *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '12, pp. 1266–1285.

[7] T. S. Tia, "Utilizing slack time for aperiodic and sporadic requests scheduling in real-time systems," Ph.D. thesis (Tech. report No. UIUCDCS-R-95-1906), Dept. of Computer Science, University of Illinois at Urbana-Champaign, April 1995.

[8] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st annual international symposium on Computer archit ecture ISCA '04*. Washington, DC, USA: IEEE Computer Society.