Exploiting Intra-Task Slack Time of Load Operations for DVFS in Hard Real-Time Multi-core Systems

Eduardo Quiñones[1], Jaume Abella[1], Francisco J. Cazorla[1,2], Mateo Valero[1,3]
[1]Barcelona Supercomp. Center (BSC) [2]Spanish National Research Council (IIIA-CSIC) [3]Univ. Politecnica de Catalunya (UPC)
{*eduardo.quinones, jaume.abella, francisco.cazorla, mateo.valero*}*@bsc.es*

***Abstract*—Power demand grows much faster than battery capacity in embedded systems. Dynamic voltage and frequency scaling (DVFS) has been shown to be extremely efficient to save energy due to the exponential dependence of power on voltage. However, voltage/frequency cannot be blindly scaled in hard real-time systems because DVFS techniques impact on the execution time, and so potentially on the worst-case execution time (WCET) of tasks.**

This paper presents a new DVFS technique for hard real-time systems that measures dynamically the *intra-task slack* existing between the actual execution time of a task and its WCET estimation, and exploits it to perform DVFS guaranteeing that the WCET is not affected. Concretely, our approach exploits the slack available due to contention in the use of shared resources in a multi-core system.

I. Introduction

Power, energy and temperature are key limiters in embedded microprocessor design. The most effective techniques to reduce energy consumption are those based on scaling the supply voltage (*Vcc*). Those techniques are typically referred to as DVFS techniques. However, decreasing *Vcc* has a direct impact on circuit delay, and hence, on the execution time, so *Vcc* cannot be scaled blindly, especially in hard real-time embedded systems where correct execution lies on the computation of a *Worst-Case Execution Time* (WCET) estimation. Therefore, besides reducing the energy consumption of the system, DVFS techniques for hard real-time systems must consider the impact that *Vcc* scaling has on the execution time (ET) and WCET estimation of tasks.

There are two main sources of slack that can be exploited to apply DVFS: inter-task and intra-task slack. Inter-task slack is the slack available between two or more tasks in the scheduling so that all of them can execute by their respective deadlines. The static inter-task slack, i.e the difference between the estimated WCET for a task and its deadline, has been exploited by several DVFS techniques [2], [12], [18], [19]. Similarly, dynamic inter-task slack, i.e the difference between the actual ET of the task and the estimated WCET, has also been exploited [4], [7], [20]. In general, techniques exploiting inter-task slack assume some freedom in the task scheduling, e.g. the the release time of tasks can be advanced, which may be unfeasible if they are synchronized with external events such as data sensed at fixed intervals.

Instead, techniques exploiting intra-task slack do not make any assumption on the scheduling and simply apply DVFS in such a way that the ET of the task is increased as much as possible to minimize energy consumption without exceeding the WCET of the task. So far intra-task slack has been exploited based on execution path detection [5], [15] or assuming that tasks can be split into subtasks, thus making intra-task slack become inter-subtask slack [3].

Multicore (CMP) processors have been considered for real-time environments due to their good performance-per-watt ratio and because CMPs allow co-hosting several task on the same chip, reducing hardware requirements. Inter-task interferences due to resource sharing have been addressed by means of techniques that bound the maximum delay a task can suffer accessing shared resources due to inter-task interferences [9], [11]. This bound is considered on the computation of the WCET for each task. Hence, the WCET estimations computed for a task in a CMP include some provisioning to account for the worst possible inter-task conflicts when accessing a hardware shared resource. However, tasks do not experience this worst case interference when they run as a part of a workload in the CMP.

In the context of this real-time capable CMP processor, this paper proposes going one step further in energy savings in hard real-time systems by automatically detecting at run-time and by means of hardware mechanisms, the *intra-task slack* available between the ET of a task and its WCET estimation due to provisioning done to deal with conflicts accessing hardware shared resources in CMPs. By dynamically detecting the discrepancy between the real and the worst-case contention assumed in the WCET estimation, we can apply DVFS in the task being run. Note that such an approach is orthogonal to all previous approaches since it can be combined with all of them either at the task or subtask level.

To illustrate our approach, this paper exploits the intra-task slack generated due to inter-task interferences of load operations accessing the shared bus and a memory controller in a CMP processor. Other techniques attacking other sources of inter-task interferences are left for future work. Our approach monitors the latency of the requests of those shared resources to track how much slack is regained with respect to its worst-case behavior, and uses such slack to perform DVFS without impacting the WCET.

II. Background

In this section we present the impact of DVFS techniques on power dissipation, as well as the multi-core architecture considered in this paper.

Impact of DVFS Techniques on Power Reduction: DVFS techniques rely on the fact that *dynamic power dissipation* (P_{dyn}) decreases nearly-cubically with *Vcc*, $P_{dyn} = p_t \cdot C_L \cdot V_{cc}^2 \cdot f$, where p_t stands for switching probability, C_L for load capacitance and f for operating frequency. In general, frequency scales near-linearly

with *Vcc* [14]. Thus, when computing P_{dyn} we obtain that decreasing *Vcc* produces near-cubic power savings (and quadratic energy savings). Similarly, *leakage power* (P_{leak}) decreases linearly with *Vcc*.

CMP Architecture. We consider an analyzable CMP in-order four-core processor in which each core has a private data and instruction L1 cache connected to a partitioned L2 cache through a shared bus [9], [11]. The L2 cache is loaded from a JEDEC-compliant DDR2 SDRAM memory system.

By design, this architecture guarantees that the maximum delay a request to a shared resource can suffer due to any other task is bounded by a pre-computed *Upper Bound Delay* (UBD). The UBD for any particular resource (e.g., bus and memory) is computed as follows, $UBD = (No_{cores} - 1) \cdot t_{busy}$, where No_{cores} is the number of cores and t_{busy} the amount of time that the resource is busy due to another core using it. Such bound relies on the assumption that shared resources are accessed in a round-robin fashion by the different cores and that the number of tasks does not exceed the number of cores [9], [11]. When computing a WCET estimation for a given task in the CMP, the UBD is considered as an additional delay on every access to any shared resource. By doing so, WCET estimations are safe and tight. Moreover, in order to avoid cache interferences cache is partitioned (*bankization*) assigning to each core a private subset of the total number of banks that no other core can use [10].

III. Intra-Task DVFS Technique for Hard Real-Time Multi-core Systems

Next we present our DVFS technique to exploit the intra-task slack of a hard real-time tasks running in a CMP. We present the approach focusing on the slack time provided by loads accessing the shared bus and memory controller, although the approach could be used for other types of events whose worst-case behavior is known and used when estimating the WCET.

A. Rationale Behind our Approach

Applying intra-task DVFS techniques to hard real-time tasks requires being aware of the impact that the reduction of the voltage/frequency has on the ET so that the WCET is not exceeded. To do so, our technique detects those events that behaved better than their worst case and quantifies how much slack has been regained.

By constructon of [9], [11] the WCET estimation for a task considers the maximum delay that inter-task interferences may introduce when accessing to hardware shared resources in a CMP. Authors proposes a CMP in which the maximum delay a request to the bus or the memory controller can suffer due to inter-task interferences is bounded by UBD_{bus} and UBD_{MC} cycles respectively (see Section II) [9], [11]. Such values are used when computing a WCET estimation for accesses to those resources so that the resultant WCET estimation is independent of the workload in which hard real-time tasks run, as the worst possible delay is considered in every bus or memory controller request.

However, not all accesses to the bus or the memory controller experience such worst-case delay. In particular, *load operations*, experience a delay due to inter-task interferences between 0 and UBD cycles. Our technique identifies the slack between the actual delay suffered due to inter-task interferences when accessing the bus or the memory controller and their UBD, allowing us quantifying how far the ET is from the WCET. By doing so, we can *guarantee* that, despite the lower execution speed due to *Vcc*/frequency reduction, the task finishes its execution before its WCET if we manage to limit the slowdown to consume only the accumulated slack.

B. Mechanism to Regain Slack

In order to identify and quantify the slack regained due to load operations that behave better that their worst case, we propose using a counter, named $CurrentSlack$, per shared resource and on-going event, and a counter named $TotalSlack$, global for each core.

The $CurrentSlack$ counter quantifies the slack regained by the load operation currently processed in the corresponding shared resource. In the multi-core processor considered in this paper, we require eight $CurrentSlack$ counters: four-cores and two shared resources, the bus and the memory controller.

The $TotalSlack$ counter quantifies the total slack regained by the task running in a given core and it is used to decide when to scale frequency and *Vcc* of this core. The counter is incremented with $CurrentSlack$ every time a bus or memory controller access is performed, i.e. once the load operation has accessed the corresponding shared resource and its actual latency is known.

Since frequency changes dynamically, we cannot count cycles. Instead, we use a time unit being the *greatest common divisor* of any feasible cycle time so that we can measure time accurately independently of the current operating frequency. We refer to such time unit as $MinTime$. On every access to the bus and the memory controller, $CurrentSlack$ can be incremented by upto $UBD_{bus} \cdot \frac{CycleTime}{MinTime}$ and $UBD_{MC} \cdot \frac{CycleTime}{MinTime}$ time units respectively, if the accessing task suffers no interaction. $CycleTime$ stands for the cycle time under the current frenquency expressed in the same time unit as $MinTime$. In other words, on an access to a given resources $CurrentSlack$ is initialized to UBD. Whenever a request arrives to the shared resource $CurrentSlack$ is decreased by the amount of time spent until the request is served (by $\frac{CycleTime}{MinTime}$ every cycle). Hence, whenever the request is served $CurrentSlack$ value matches the amount of time regained by such request.

We assume a linear degradation of performance and hence slack. That is, if we change processor to a frequency that is $(\frac{1}{2})$ of the nominal frequency, every cycle in the new frequency we loose one cycle of slack in the nominal frequency. This linear relation is a safe bound as in reality this is the maximum performance degradation a task can suffer when the frequency is reduced.

1. **If** ($TotalSlack > OvhDecOneStep + MinPeriod +$
$+OvhIncOneStep \cdot (NumStepsBelowMax + 1) +$
$+CheckPeriod \cdot (NumStepsBelowMax + 1)$) **then**
2. Decrease frequency by one step
3. **Else if** ($TotalSlack \leqslant$
$OvhIncOneStep \cdot NumStepsBelowMax +$
$+CheckPeriod \cdot NumStepsBelowMax$) **then**
4. Increase frequency by one step
5. **Else**
6. Keep current frequency
7. **Endif**

Figure 1. Algorithm to perform our intra-task DVFS approach

C. Using Regained Slack for DVFS

We propose an aggressive DVFS policy that decreases frequency/*Vcc* as soon as we have enough slack, run some significant amount of time at low frequency and raise frequency again if the amount of slack available matches the overhead required to raise frequency back to the original frequency.

The detailed algorithm is depicted in Figure 1. $OvhDecOneStep$ ($OvhIncOneStep$) stands for the amount of time required to decrease (increase) frequency by one step, $MinPeriod$ is a threshold indicating how long we can spend in a given frequency level to obtain significant energy savings [8] (determined empirically), $NumStepsBelowMax$ indicates how many frequency steps there are between the current frequency and the nominal one at which this task was intended to execute, and *CheckPeriod* is the amount of time elapsed between two consecutive checks of the algorithm inputs (it can be 0 if they are checked continuously).

As shown, frequency is decreased only when there is enough time to decrease the frequency, obtain significant energy savings and return to the maximum frequency in time to prevent any timing violation (line 1). Similarly, once we detect that the frequency increase cannot be delayed without putting the WCET at risk, frequency is increased (line 3). Otherwise, the core remains at the current operating frequency (line 5).

Note that all values in the algorithm but $TotalSlack$ and $NumStepsBelowMax$ are constants and hence, can be hardwired. Given that the number of frequency steps is limited in general (typically below 10), multiplications are very simple. Overall, the hardware required to track the slack regained and to decide whether to increase/decrease frequency is very small. Moreover, if our approach is extended to regain slack from other sources, only $TotalSlack$ will be affected and the algorithm will remain exactly the same without requiring further logic to control frequency/*Vcc*.

IV. Experimental Setup

We use the CMP processor presented in Section II. Further details can be found in [9]–[11]. Independent per-core voltage domains are used [2], [18], [19]. Each core can operate at different frequency/voltage levels [6] as shown in Table IV. *Idle* stands for the data retention state reached when a task finishes. Voltage/frequency can be changed every 10K cycles [8].

Table I
Frequency, voltage and power levels

Frequency (MHz)	Voltage (V)	Power (mW)
1064	1.95	1800
532	1.47	770
266	1.21	340
133	1.00	160
idle	0.85	44

We used an in-house cycle-accurate, execution-driven simulator compatible with Tricore ISA and derived from CarCore [16]. Our simulator models a CMP architecture composed of 4-cores with the characteristics described above. We model the DRAM memory system with DRAMsim [17], which we integrated inside our simulation framework. Moreover, we also integrated the RapiTime comercial tool [1] inside our simulation framework in order to estimate the WCET of hard real-time tasks.

For our experiments, we use EEMBC Autobench [13], a well-known benchmark suite that reflects the current real world demands of embedded systems. We define four different application workloads, each composed of four different benchmarks, which aim to balance system load. To do so, we sort the benchmarks from highest to lowest energy savings (a proxy of their shared resources requirements) achieved by our DVFS technique when run in isolation in the CMP. Then, we create the workloads by grouping the 16 benchmarks as follows: 1-8-9-16 (*cacheb01, iirflt01, bitmnp01, tblook1*), 2-7-10-15 (*aifirf01, canrdr01, idctrn01, puwmod01*), 3-6-11-14 (*aifftr01, ttsprk01, matrix01, basefp01*) and 4-5-12-13 (*aiifft01, pntrch01, rspeed01, a2time01*).

V. Evaluation

This section evaluates our approach in terms of energy savings and execution time impact, considering two different scenarios: (1) when the target task runs without inter-task interferences from the other tasks (labeled as *DVFS no interferences*). This represents an upper bound for energy savings, since none of the assumed conflicts in the WCET estimation happen in reality; and (2) a realistic scenario where the task runs simultaneously with three other tasks (labeled as *DVFS full workload*) competing for the shared resources, as described in Section IV. In this latter case, all tasks start simultaneously and, if any of the other tasks competing for the shared resources finishes early, it is re-run again to keep always those three tasks competing.

Side effects of running the tasks in a real scheduling have not been considered. In such an environment each task will observe changes in shared resource demand coming from other tasks since all of them will apply DVFS simultaneously and independently. So far we consider an scenario where the period matches the WCET and the task release time is fixed, so the only source of slack that our approach can exploit is the intra-task slack due to shared resource usage.

Energy Savings: Figure 2 depicts the energy savings achieved by our approach for each task on the two scenarios explained above and using as a baseline the energy consumption of tasks when running in isolation

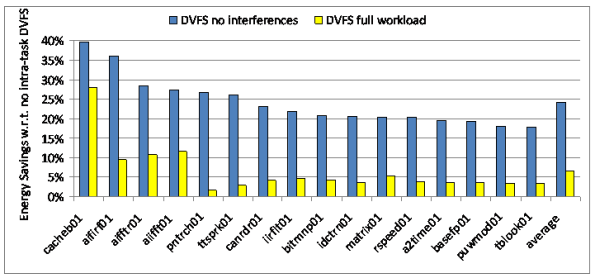

Figure 2. Relative energy savings of our intra-task DVFS approach.

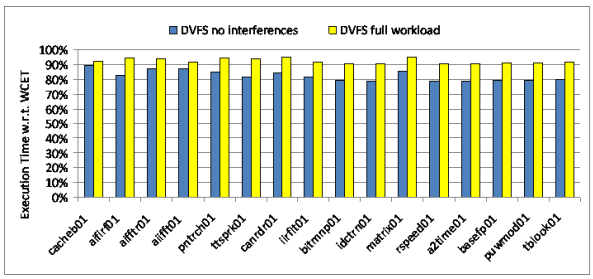

Figure 3. Relative ET of our intra-task DVFS approach w.r.t. the WCET.

without applying any DVFS technique.

We observe that energy savings are very high in the *DVFS no interferences* scenario. In particular, energy savings are in the range 18% - 40%, with average savings of 24%. Note that those energy savings are very large, especially if we consider that our approach exploits only the slack available for load operations.

In the *DVFS full workload* scenario energy savings are lower, but still significant. They range between 2% and 28%, with average energy savings of 6.5%.

Execution Time (ET) Impact: In all cases the ET of each task never violates its WCET estimation in CMP. Figure 3 shows the ET with respect to the WCET estimation when our DVFS technique is applied. (1) The average ET is 92% of the WCET for the *DVFS full workload* scenario. If DVFS is not applied, then the ET is 79%. (2) Similarly, average ET is 82% of the WCET for the *DVFS no interferences* scenario. If DVFS is not applied, then the ET is 65%.

The difference between the actual ET and the WCET for those two scenarios (8% and 18% respectively) is due to other events not exploited in this paper (store operations, etc.), which will be studied in our future work.

VI. Conclusions and Future Work

In this paper we describe a DVFS approach for hard real-time systems that exploits the multicore-generated intra-task slack automatically at runtime by means of hardware mechanisms, by identifying and quantifying those events that behave better than their worst-case latency, thus guaranteeing that no hard real-time task exceeds its WCET estimation in a CMP. Our evaluation shows that by simply exploiting the intra-task slack provided by inter-task interferences of load operations when accessing to the shared bus and memory controller in a multi-core processor, average energy savings per task between 6% and 24% are achieved.

Our approach neither requires to perform any static analysis of the task at design time, nor has dependencies on the actual schedule. Hence, our approach can be applied regardless of the scheduling technique used. Moreover, our approach can be combined with current DVFS techniques for hard real-time systems without further changes.

Future work includes a complete evaluation of our technique when a real task scheduling is performed instead of studying each task in isolation in a fully controlled environment. Similarly, we will extend our technique to also deal with store instructions accessing the bus and to other hardware shared resources.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Education and Science under grant TIN2007-60625 and by the MERASA FP7 STREP European Project under grant 216415. Eduardo Quiñones and Jaume Abella have also been partially funded by the Spanish Ministry of Science and Innovation under the grant Juan de la Cierva JCI2009-05455 and by the Generalitat de Catalunya under grant Beatriu Pinós 2009 BP-B 00260 respectively. Authors would also like to thank Marco Paolieri for their helpful comments.

References

[1] *RapiTime: WCET analysis. www.rapitasystems.com.*
[2] T.A. AlEnawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In *RTAS '05*, 2005.
[3] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems. In *ISCA '03*, pages 350–361, 2003.
[4] A. Andrei, M.T. Schmitz, P. Eles, Z. Peng, and B.M. Al Hashimi. Quasi-static voltage scaling for energy minimization with time constraints. In *DATE*, 2005.
[5] S.V. Gheorghita, T. Basten, and H. Corporaal. Intra-task scenario-aware voltage scheduling. In *CASES*, 2005.
[6] Intel Corporation. *Intel PXA270 Processor, Data sheet.*
[7] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED*, 1998.
[8] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *ICCAD '02*, 2002.
[9] M. Paolieri, E. Quinones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA '09*, 2009.
[10] M. Paolieri, E. Quinones, F.J. Cazorla, Robert I. Davis, and M. Valero. Ia^3: An interference aware allocation algorithm for multicore hard real-time systems. In *RTAS*, 2011.
[11] M. Paolieri, E. Quinones, F.J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. In *Embedded System Letter*, 2009.
[12] P. Pillai and K.G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01*, 2001.
[13] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[14] T. Sakurai and A.R. Newton. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *IEEE Journal of Solid-State Circuits (JSSC)*, 25(2), 1990.
[15] D. Shin and J. Kim. Optimizing intra-task voltage scheduling using data flow analysis. In *ASP-DAC*, 2005.
[16] S. Uhrig, S. Maier, and T. Ungerer. Toward a processor core for real-time capable autonomic systems. In *ISSPIT '05*, 2005.
[17] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 2005.
[18] C. Xian, Y.-H. Lu, and Z. Li. Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In *DAC '07*, 2007.
[19] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *RTSS '01*, 2001.
[20] Y. Zhu and F. Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *RTAS*, 2004.