Opportunities and Challenges of Application-Power Control in the Age of Dark Silicon

Dominic Hillenbrand[1,a)] Yuuki Furuyama[1,b)] Akihiro Hayashi[1,c)] Hiroki Mikami[1,d)]
Keiji Kimura[1,e)] Hironori Kasahara[1,f)]

Abstract: In the age of dark silicon on-chip power control is a necessity. Upcoming and state of the art embedded- and cloud computer system-on-chips (SoCs) already provide interfaces for fine grained power control. Sometimes both: core- and interconnect-voltage and frequency can be scaled for example. To further reduce power consumption SoCs often have specialized accelerators. Due to the rising specialization of hard- and software general purpose operating systems require changes to exploit the power saving opportunities provided by the hardware. However, they lack detailed hardware- and application-level-information. Application-level power control in turn is still very uncommon and difficult to realize. Now a days vendors of mobile devices are forced to tweak and patch system-level software to enhance the power efficiency of each individual product. This manual process is time consuming and must be reiterated for each new product. In this paper we explore the opportunities and challenges of automatic application-level power control using compilers.

1. Introduction and Related Work

In the domain of mobile devices the market is dominated by multi-core SoCs such as Texas Instrument's *OMAP*, Qualcomm's *Snapdragon*, NVIDIA's *Tegra* and Samsung's *Exynos*. These SoCs have accelerator- and peripheral-cores for video and audio applications. SoCs for base stations *Freescale QorIQ Qonverge* and car navigation *Renesas SH-Navi3* - for example - are conceptually similar but deploy different domain specific accelerators.

Recent SoCs support various methods for reducing power consumption, such as: DFVS [1] (Dynamic-Frequency-Voltage Scaling), adaptive body bias [2], [3], [4], big-little as well as power- and clock-gating. These power saving mechanisms can often not be independently applied to cores due to resource sharing at the hardware-level. Thus otherwise independent device drivers must be aware of shared clocks and voltage controllers - for example - when they exert power control. Excessive resource sharing may severely reduce the design space of power control.

In the reference [5], the authors projected that in a relatively short time span a significant amount of chip area will remain "dark" - due to power- and parallelism-constrains. New approaches such as near-threshold computing [6] achieve up to 10 times better power efficiency and may help to reduce "dark silicon". Intel [7] has recently designed a prototype processor that is able to operate from 280mV up to 1.2V (3-915MHz) - thus covering the range from sub-, near- up to super-threshold. In the sub-threshold region leakage dominates and in the super-threshold region dynamic power. The lowest energy per instruction is achieved in the near-threshold region.

Should future SoCs provide scaling from sub- to super-threshold then power will vary more than 10x depending on voltages. Thus DFVS-control - for example - has a large window of opportunities for power reductions in such chips.

In this paper we focus on power control in the open-source *Linux*- and *Android* operating system. Both support DFVS through the "*cpufreq*" [8] device driver. The *cpufreq* device driver calls user-selectable "governors" to determine new voltages and frequencies. Afterwards the *cpufreq* device driver invokes low-level device drivers to actually set voltages and frequencies.

Linux has several governors: *user-space, ondemand, conservative, powersave, performance* and *interactive*. For our considerations the *user-space*-governor is most important as it enables DFVS for user-space applications. The *ondemand, conservative* and *interactive*-governor provide automatic DFVS-control based on monitoring application activity. The last two governors *powersave* and *performance* merely configure the lowest- or highest-performance operating points.

Similar to *cpufreq* driver, *Linux* has a *cpuidle* driver [9] which has two governors *ladder* and *menu*. The *cpuidle* device driver calls the active governor to determine sleep modes for idling. The *ladder*-governor selects sleep modes in a step-wise fashion. The *menu*-governor exploits scheduling information which are available when the kernel supports "tickless" mode[10]. In the next

[1] Waseda University, Shinjuku-ku, Tokyo, Japan
[a)] dominic@kasahara.cs.waseda.ac.jp
[b)] furuyama@kasahara.cs.waseda.ac.jp
[c)] ahayashi@kasahara.cs.waseda.ac.jp
[d)] hiroki@kasahara.cs.waseda.ac.jp
[e)] kimura@kasahara.cs.waseda.ac.jp
[f)] kasahara@waseda.jp

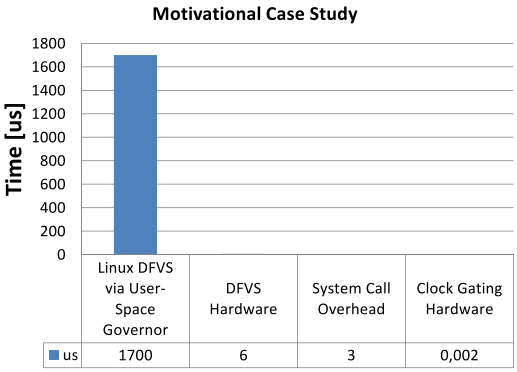


Fig. 1 Motivational Case Study
The Figure illustrates that under *Linux* 2.6.27 user-space DFVS is inefficient on the Renesas RPX-SoC . The *user-space* governor interface needs 1700μs on average but writing to the hardware voltage- and frequency registers takes just 6μs. In this particular micro-benchmark an user-space test-application toggled DFVS between 81- and 648 MHz. In Section 5.3 we will show more efficient interfaces for DFVS for user-space applications. Clock gating on RPX takes one cycle. However, clock gating can only be utilized from kernel-space since a privileged instruction must be executed. If the kernel would provide a clock gating system call it would take 1500 times longer at 648 MHz than actually conducting the clock gating operation.

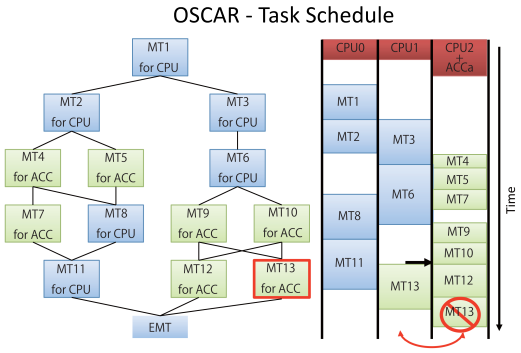


Fig. 3 OSCAR - Task Schedule
The figures illustrate a static schedule generated by our OSCAR compiler for a heterogeneous SoC with processors and accelerators. The left figure visualizes task dependencies. The boxes represents macro-tasks (MT) which are coarse grained tasks with loops, function calls and basic blocks. The blue coloured boxes can be mapped to processors, the green coloured boxes can additionally be mapped to accelerators (ACC). The figure on the right shows the schedule for three processors CPU0, CPU1, CPU2 and an accelerator (ACCa). CPU2 offloads tasks to the accelerator and performs necessary data transfers. First, our compiler assigns the ready macro-task MT1 to CPU0. Then MT2 and MT3 are mapped to CPU0 and CPU1. After MT1 finishes, MT2 and MT3 become ready and so forth until all tasks have been executed. Occasionally, ”green” accelerator tasks are mapped to processors if the accelerator is unavailable.

section we present a motivational case study to highlight the challenges of user-mode pwoer control in the *Linux* kernel.

2. Motivational Case Study

The Renesas RPX processor - see Figures 1 and 2 - provides low latency DFVS and clock-gating. Changing the voltage- and frequency registers takes a few microseconds and clock gating merely nano-seconds. In Figure 1 we can see that the *Linux user-space*-governor interface is not able to exploit the hardware capabilities. Where do these overheads occur?

For user-space DFVS it is necessary to understand how the *user-space*-governor functions:

First, applications open a pseudo-file ”*scaling_setspeed*” in the *sysfs*-file system. Secondly, they write a text string with the new frequency into the pseudo-file. Thirdly, they close the file.

Thus three system calls are required. However, this still does not explain the manifold overhead. Our analysis indicate that the *sysfs*-kernel layer which passes pseudo-file operations to the *cpufreq*-device driver is to blame. In Section 5.1 we will present improved interfaces for user-space DFVS control.

Our novel contributions are:

- Case Study: Analysis of three methods for user-space DFVS
- Efficient clock- and power-gating for user-space applications via ”autoidle”-threads and new system calls
- A power-adaptive in-kernel barrier for user-space applications
- Discussion of opportunities and challenges of user-space power control

Our contributions and insights apply to embedded systems as well as large data centers since both are power constrained and frequently utilize *Linux*. In the following section we introduce our experimental hardware setup.

3. Experimental Hardware Setup

For our experiments we have used the Renesas RPX-SoC[11], [12]. This 45nm research SoC - see Figure 2(a) - has eight SH4A processors, reconfigurable ALU arrays, two MX-2 matrix processors, a video processing unit, and various peripheral cores for DDR, SATA, PCIe, DMA, GPIOs and UART. The chip consumes ca. 3 watts at 648 MHz and 1.15V. In our board configuration the voltage can be scaled in three steps from 1.1 - 1.3V. The frequency is adjustable in four steps: 81 MHz, 162 MHz, 324 MHz and 648 Mhz.

The RPX-SoC is supported by two operating systems: *Linux* 2.6.27 and *LWOS*. *LWOS* is a light-weight operating system written by Renesas for internal usage. *Linux* can only utilize the first processor cluster (4 cores) since cache coherency is not maintained between clusters. *LWOS* and its applications do not have this limitation and can utilize all 8 cores.

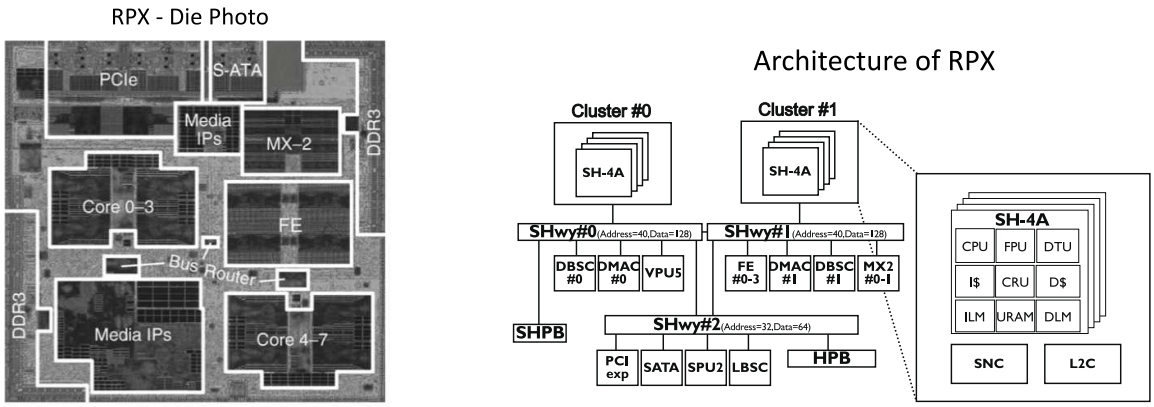
The following section we propose compiler assisted power control for some user-space applications and introduce our OSCAR compiler tool-chain.

4. Compiler generated power control to the rescue

The computing world is moving away from standard computers to more specialized devices. Tablet PCs, smart phones and server processors utilize highly specialized SoCs. Accordingly, power management becomes more specialized as well.

Operating systems usually have DFVS- and idle-device drivers for new SoCs - but they are not able to schedule applications and power control efficiently together - simply because the scheduler is unaware of higher-level behaviours.

To escape the dilemma partially one possibility - we propose



(a) RPX- Die Photo
The die photo above shows the 45nm Renesas-RPX experimental processor[11], [12] which we have used for our experiments. On the top we can see that the SATA- and PCIe-core. Latter takes a significant amount of chip area. The two processor clusters: cores 0-3 and cors 4-7 are connected by a bridge. The DDR memory controllers are located on the lower left and upper right of the die. The FEPGA accelerators which are reconfigurable and the matrix processors (MX-2)for image processing have not been used in this paper. Image source: [12]

(b) Architecture of RPX
The diagram shows the architecture of the Renesas RPX prototype chip. On the top we can see two processor clusters - each with four SH-4A cores. The SH-4A core has an floating-point unit (FPU), data transfer unit (DTU), instruction and data caches, instruction- and data local memory (ILM/DLM), distributed memory (URAM), cache ram control unit (CRU) and snoop controller (SNC). Each cluster is internally connected by a bus. Both clusters have DDR memory controllers (DBSC) and direct memory access controllers (DMAC). Additionally, the first cluster has a VPU5 codec engine and the second, four FEGA accelerators which are reconfigurable and a matrix processor (MX2) for image processing. Both clusters are connected by a bridge. Cache coherency is maintained within clusters but not between them. A third bus provides access to peripherals such a PCI, SATA, sound processing unit (SPU) and local bus state controller (LBSC) [11], [12].

Fig. 2 RPX - Heterogeneous Mulitcore SoC - Photo and Architecture
The left Figure shows the die of the Renesas RPX-prototype chip, the right Figure shows the schematic architecture diagram. We used the RPX chip for our experiments in this paper.

in this paper - is to use *auto-parallelizing* compilers such as our *OSCAR*-compiler[13], [14], [15] - for suitable applications. Our *OSCAR*-compiler generates static task- and data-transfer-schedules - see Figure 3 - as well as power control code with nano-second resolution. *OSCAR* requires a SoC-specification file to compute the schedules. Thus porting the software stack involves creating a new SoC-specification and re-compiling the code.

OSCAR is implemented as a source-to-source -compiler for C/FORTRAN. From input sources - *OSCAR* generates sources for each processor which is compiled by standard C/FORTRAN-compilers such as *gcc* for example. In the following section we discuss different methods for user-space power control.

5. Case Study: User-space DVFS-control

The motivational case study in Section 2 revealed that user-space power control can be inefficient. In this section we will introduce two alternative methods of user-space DFVS control.

5.1 New system call for DFVS

To avoid the pseudo-file system overheads of the *Linux user-space* governor, we implemented a new system call that directly invokes the *cpufreq*-device driver. Our initial version resembled this code fragment:

```
asmlinkage long sys_freq(int core, int freq) {

struct cpufreq_policy policy;

cpufreq_get_policy( &policy, core);

policy.cpu = core;
policy.governor->store_setspeed((&policy, freq);

}
```

The above code first fills the *cpufreq_policy* data structure with the core number and calls the governor's *store_setspeed* function. Our new system call avoids textual parameter parsing, the pseudo-file system layer and reduces the number of systems calls - 1 instead of 3.

5.2 User-space device driver

After reducing the overhead of the kernel system call we were asking ourselves how we could further minimize overheads. On our hardware platform frequency- and voltage-registers are memory-mapped registers. Via remapping memory-pages it is possible to access these registers from user-space.

On *Linux* memory mapping can be performed by custom device drivers or more generically by using the */dev/mem* device driver and the *mmap*-system call. The following code fragment illustrates the procedures:

```
fd = open("/dev/mem", O_RDWR|O_SYNC);
...
mapped_addr = (unsigned int) mmap(NULL,
   num_of_map, (PROT_READ | PROT_WRITE),
   MAP_SHARED, fd, CnIFC_ADRS(0));
...
```

CnIFC_ADRS(0) stands for the frequency control register address of core 0 on RPX. The frequency registers of the remain-

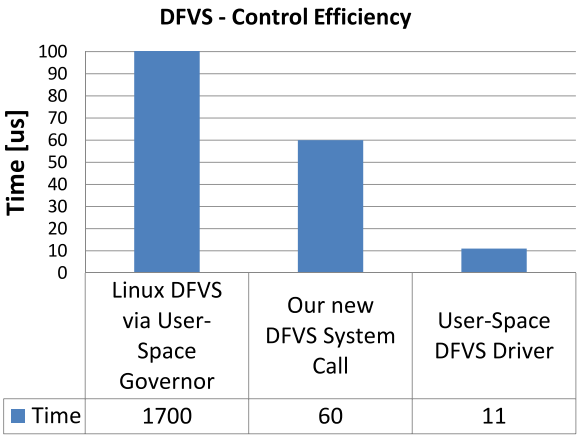


Fig. 4 DFVS - Control Efficiency
In Section 2 we revealed that user-space DFVS on *Linux* takes 1700 μs but the hardware accounts just for 6μs. In Section 5 we introduced two new methods for user-space DFVS, a new DFVS-system call and a user-space DFVS device driver. This Figure shows that our new system call is about 30 times faster than the standard *Linux* DFVS *user-space* governor interface. Our user-space device driver is even 150x faster. It takes 11μs, 5μs (between 300-3000 cycles) more than writing to the registers. The additional time is spend executing driver code instructions and to perform memory accesses. Optimized device drivers could reduce the number of instructions executed and place device driver state information in on-chip memories.

ing cores follow on the subsequent memory pages. Care must be taken that these mappings are not cached. On *Linux* the */dev/mem* device must be opened with the O_SYNC flag set. In our case even this did not work till we patched the kernel */dev/mem*-device driver.

After mapping the necessary registers changing frequencies becomes a memory store operation:

```
*(unsigned volatile int*) freq_ctrl_addr = ifc;
```

The *ifc* value is a platform specific and is used to configure the on-chip frequency divider. Changing the voltage is done in a similar fashion.

Once we could remap frequency- and voltage-registers successfully into user-space, we ported the kernel device driver to user-space and tested it successfully. In the following section we compare the performance of our two new power control interfaces and original one.

5.3 User-Space DFVS-Performance

Figure 4 shows that both our DFVS methods have a much lower latency than the original Linux DFVS interface via the *user-space* governor. Our user-space device driver performs best. Our new system call takes longer than can be explained by system call overhead which accounts only for 3μs. The additional 46μs are spend in kernel for ”extra” activities.

Closer investigation within the *Linux* kernel reveals that the *cpufreq* driver calls a *cpufreq_notify_transition* function that is invoked before and after every frequency change. The function notifies kernel sub-systems about processor frequency changes. The call chain must be synchronized across processors and may therefore be costly. The *adjust_jiffies* function - for example - is called before and after frequency changes to adjust time keeping in the *Linux* kernel. Besides this function there are no other sub modules that need notification on RPX.

However, for more complex SoCs such as those mentioned in the introduction the situation is often more complex. Frequency- and voltage changes may affect multiple on-chip components and kernel drivers. Through the notification call chain otherwise independent *Linux* device drivers can act upon changes in shared infrastructure - such as clocks or voltage regulators. The cost of this flexibility is however, that changes must be synchronized across multiple processors. Thus potentially diminishing the power saving capabilities of modern chips for ”short” time periods. In the following section we try to make clock- and power gating accessible to user-space applications.

6. Case Study: Clock- and power-gating

In addition to DFVS we wanted to make clock- and power-gating accessible to our OSCAR-compiled applications. On our prototype platform RPX clock- and power-gating can be initiated by issuing the privileged *sleep* instruction with different flags. Unfortunately, the instruction is only accessible if the processor is in privileged mode. This is especially annoying since clock gating requires just a few nanoseconds but system calls at least 3μs. Executing applications in privileged mode would allow instructions such as *sleep* to be accessible by applications.

The *Linux*-kernel supports clock- and power gating indirectly through the *idle* threads. *Idle* threads are invoked whenever (per-processor) scheduler’s run-queues are empty. Eventually, *idle* threads will cause processors to transition to certain *sleep*-modes which deploy clock- or power-gating.

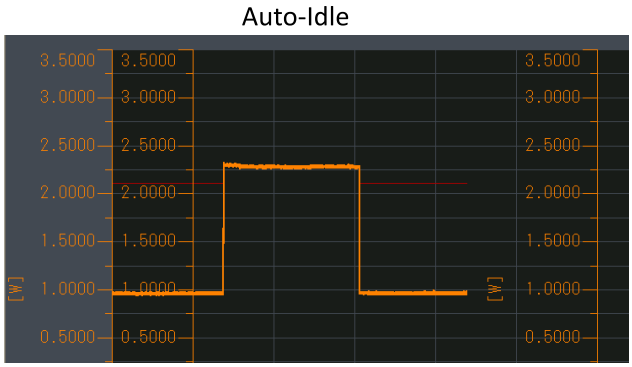
For user-space applications we have implemented a new pair of system calls which (1) invoke the kernel *idle* functionality directly, or (2) wake idling threads up. The following code fragment was taken from the *Linux idle*-function and integrated into our new *idle* system call:

```
if (cmd == SYSFREQ_IDLE) {
..
tick_nohz_stop_sched_tick(1);
while (!need_resched())
      idle();
tick_nohz_restart_sched_tick();
...
}
```

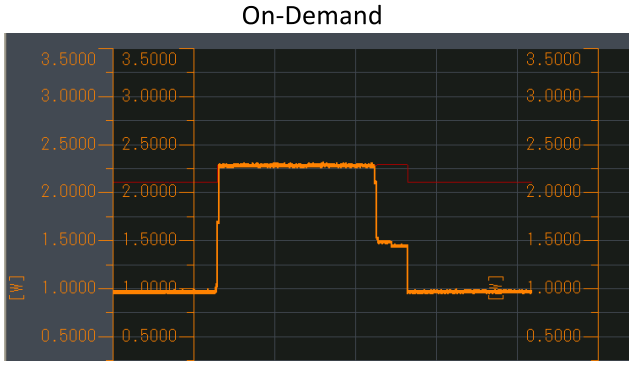
The code disables the periodic scheduler tick to avoid unnecessary wake-ups. As long as the scheduler does not require rescheduling the *idle* task wfunctionill invoke the platform specific *idle* function. The *idle* function calls low-level device drivers for clock- and power-gating.

Our new *idle* system call allows OSCAR *compiler-generated* power control code to directly call *idle* for clock- and power-gating - while keeping caches hot.

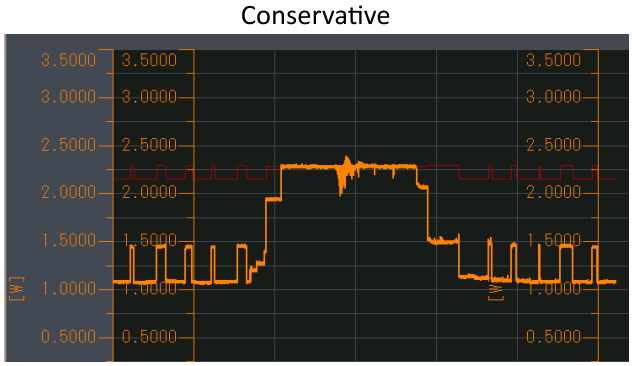
For applications that have not been compiled with OSCAR, we have developed an experimental *autoidle*- function that can be enabled at run-time. If the kernel *autoidle*-flag is set, a processor will immediately switch to lower frequencies and/or clock



(a) Auto-Idle
Our new *auto*-idle enabled *Linux*-kernel exploits the low-latency clock- and power-gating capabilities of the RPX-SoC. When *idle*-threads are activated then we immediately power-gate processors. If an application thread is activated we ramp-up DFVS to pre-specified values. For latency sensitive applications that perform their activities in bursts - for example in sensor networks - this behaviour may be better suited than the standard *Linux* governors.



(b) On-demand governor
The *Linux on-demand*-governor adapts DFVS to system load. Like the *auto*-idle enabled *Linux*-kernel it quickly ramps up DFVS but does not immediately reduce power if load drops.



(c) Conservative governor
The *Linux conservative*-governor is slower in its response to load changes than the two previous approaches. Furthermore, it oscillates if the system is unloaded.

Fig. 5 Operating-system Power Control - Auto-Idle, On-Demand, Conservative
The three graphs show power [W] over time as the system transitions from unloaded to loaded and back again to unloaded. We conducted the measurements using the Renesas RPX-prototype board which supports inductive power measurements of the SoC.

gate the processor - when the kernel *idle* task is scheduled. If the kernel *idle* task relinquishes control, then the previous frequency will be restored immediately. The initial base frequency is fixed but configurable.

On the power scope auto-*idle* has a binary ”on/off” pattern, whereas the *Linux on-demand*- or *conservative*-governors need more time to track application activity - see Figure 5.

We think that our experimental *autoidle*-mode may be useful to save power in event-triggered applications. In the following section we introduce a power-adaptive kernel interface for barrier synchronization.

7. Adaptive and power aware kernel barrier

We have implemented a barrier-system call within the *Linux*-kernel similar to the *gcc*-OpenMP barrier[*1]:

Threads that arrive at our kernel barrier spin for some time before blocking in *idle*. As described earlier, in *idle*-mode processors are either clock- or power gated. The last thread to arrive at the barrier will wake-up all waiting threads and reset the barrier.

Our adaptive power optimizations adaptively set the frequencies of the first threads to arrive at the barrier to reduced values while they spin. Reducing the frequency also helps other threads on RPX for example - since the voltage controller is shared - thus voltages can only be dropped if all threads fall below certain frequency levels.

If all but one threads have arrived at the barrier, then we boost the frequency of the last thread in order to finish the barrier quicker. This behaviour may be beneficial if static power is high and excessive waiting burns power.

Since the barrier is implemented within the kernel it is possible for processes to synchronize and not only for threads. Ideally, OSCAR applications will not need the power-adaptive features of our kernel barrier - since the static task schedule will automatically issue near-optimal power control commands. However, on some hardware architectures with complex memory architectures and interference from other unrelated tasks it may be possible that the static schedule is disturbed. Our adaptive barrier can help to dynamically fix such situations until the threads synchronize again. In the next section we discuss how we try to keep interference from unrelated *Linux* applications to a minimum.

8. Task-processor binding

OSCAR applications assume processors to be under their full control in respect to scheduling and power control. On *LWOS* - see Section 3 - only one application is running at the time and this assumption holds. On *Linux* - however - the situation may be very different. It is up to the *Linux* scheduler to decide when and what tasks to execute and migrate among available processors.

Therefore, we have devised a kernel modification which keeps all *Linux* background tasks on processor zero. Thus the remaining processors are ”free” for OSCAR-applications.

In *Linux* each process has a *task_struct*. We extended this *task_struct* with an OSCAR-flag and patched all places where the *Linux* scheduler may migrate threads. Thus at run-time we can ensure that *Linux* application will never be spawned or migrated to processors under OSCARs control.

The following source code fragment shows how our new system call binds OSCAR processes via our *SF_BIND* command - before executing the *sched_setaffinity* call.

```
cpu_set_t set;

CPU_ZERO(&set);
CPU_SET(core, &set);

// mark as OSCAR task
syscall(CPUFREQ, SF_BIND, core);

int rv = sched_setaffinity(
     getpid(), sizeof(cpu_set_t), &set);
....
```

To test our approach we have written a small test application that binds to processors other than processor zero and calls our new *idle* system call - see Section 6. Thanks to our kernel modifications we were able to stay in *idle* for up to 30 seconds without any interruptions. On processor zero where all background tasks and daemons are located this would be impossible. On processors 1-3 our modified RPX-*Linux* faces few disturbances and therefore provides a suitable environment for statically scheduled OSCAR applications. In the next section we introduce our new kernel system calls for taking processors completely offline.

9. Processor hot-plugging from user-space

The *Linux* kernel supports processor hot-plugging from user-space via an ”online” pseudo-file. Applications can open this file and read- and write to it similar to the default DFVS user-space pseudo file mentioned earlier. The standard kernel includes many unnecessary *wait*-statements that we could remove safely for the RPX-SoC. We were able to reduce the transition times from 2 seconds down to a few milliseconds. On RPX *Linux* however - the processor hot-plug device driver is not yet able to exploit power- or clock gating if processors are taken offline. Nonetheless, it was important to see that we were also able to speed up these operations after careful analysis of kernel- and platform specific driver code. In the following section we discuss security issues of user-space power control.

10. Security

The *Linux* kernel requires root status to let user-space applications write to pseudo-files that provide interfaces to device drivers. Our system call has currently no security checks which is fine for prototyping, testing and closed embedded systems. In the future we may include checks based on group permissions. OSCAR compiled applications could - for example - belong to an OSCAR group to automatically gain access to user-space power control. On the hardware side security is rather coarse grained. Privileged instructions for clock-gating - for example - can usually not be made accessible to selected applications but only to the kernel. For user-space device drivers it will be necessary to define fine-grained security models in order to provide safe access to hardware settings. In the next section we discuss the challenge of synchronizing state between the kernel- and user-space device drivers.

11. Synchronizing state between kernels and user-space device drivers

All kernel based interfaces for power control drivers - such as our new system call for DFVS maintain a correct view of hardware states within the kernel. User-space device drivers - however - may cause inconsistencies between user-space- and kernel-

[*1] See *libgomp* source from *http://gcc.gnu.org/ for the barrier implementation.*. Currently two targets are supported *Linux* and *POSIX*. The *Linux* target uses the FUTEX-system call for fast synchronization.

device drivers. During testing we avoided inconsistencies by configuring the *user-space* governor of *Linux*. The *user-space* governor does not actively change frequencies- or voltages. Furthermore, our user-space device driver restores frequencies- and voltages - so before- and after executing OSCAR-applications.

For smart phones- and tablet PC- operating systems such as *Android* this approach may be to static. It may be necessary to switch between different governors depending on active applications. Many applications may be suitable for execution with the *ondemand, conservative* or *interactive*-governors that *Andoid* and *Linux* provide. OSCAR applications - however - always require the *user-space* governor. A power management middleware that automatically switches among governors is still missing on those operating systems. In the following section we reflect upon some user-space power control issues - that we have been faced with in the previous sections - more deeply.

12. Experiences from the user-space power control front-line

There are several challenges surrounding *user-space* power control ranging from hardware issues to security which we have discussed in the previous sections. Currently, existing SoCs have to be carefully analyzed and possibly changes must be made to kernels in order to work around hardware limitations. Unfortunately, user-space power control is not even an after thought in architecture and operating systems.

RPX - our prototype processor - allowed us to re-map frequency- and voltage-registers into user-space. Other architectures may require privileged instructions to set register values. On RPX - for example - clock gating requires privileged instructions. To fully exploit clock gating on RPX we would need to run our applications in privileged-mode along side with the kernel.

Another, easily overseen aspect is if processors can configure DFVS only for themselves, or also for other processors[*2]. On some architectures certain processor specific registers can only be changed reliably if instructions execute on the target processors. For RPX under *LWOS* - for example - it is necessary to wait 1μs after writing to a frequency register of another processor. On *Linux* - however - the RPX processor is configured differently and only local processors can reliably change their own frequencies. The low-level RPX *Linux* device driver migrates itself to the target processor if necessary. However, task migration can be very costly - on RPX $>100\mu$s for example. The *Linux eSPARC* DFVS device driver - in comparison - must execute a minimum number of NOPs on the target processor after frequency changes. This can only be guaranteed if interrupts are disabled - something which is normally not possible from user-space. The next section concludes our paper.

13. Conclusion

In this paper we have proposed to use auto-parallelizing compilers to generate task- and *power-control* schedules. The generated schedules can be configured for very high time resolutions down to nanoseconds. Upcoming- and existing research processors already offer *low latency* DFVS, clock- and power-gating However, current applications and operating systems cannot exploit these capabilities fully. Our DFVS-case study showed that existing overheads can be reduced to negligible amounts - if hardware and operating systems are flexible enough. Furthermore, operating systems and hardware must ensure that statically scheduled applications are not disturbed by unrelated applications, or kernel-threads that can be migrated, postponed or deactivated. In this paper we have made contributions to this area. We want to raise awareness among processor architects and hope they will enable us to exploit low-latency compiler-controlled power control in parallel applications.

[*2] For OSCAR compiled applications it is generally sufficient if underlying drivers respect specified hardware behaviours. The OSCAR power control functions are specified in the official OSCAR-API which can be downloaded from our website.

References

[1] T. D. Burd and R. W. Brodersen, "Design issues for dynamic voltage scaling," in *Proceedings of the 2000 international symposium on Low power electronics and design*, ser. ISLPED '00. New York, NY, USA: ACM, 2000, pp. 9–14.

[2] G. Gammie, A. Wang, M. Chau, S. Gururajarao, R. Pitts, F. Jumel, S. Engel, P. Royannez, R. Lagerquist, H. Mair, J. Vaccani, G. Baldwin, K. Heragu, R. Mandal, M. Clinton, D. Arden, and U. Ko, "A 45nm 3.5g baseband-and-multimedia application processor using adaptive body-bias and ultra-low-power techniques," in *Solid-State Circuits Conference, 2008. ISSCC 2008. IEEE*, feb. 2008

[3] G. Delagi, "Harnessing technology to advance the next-generation mobile user-experience," in *Solid-State Circuits Conference ISSCC, 2010 IEEE International*, feb. 2010

[4] H. Mair, A. Wang, G. Gammie, D. Scott, P. Royannez, S. Gururajarao, M. Chau, R. Lagerquist, L. Ho, M. Basude, N. Culp, A. Sadate, D. Wilson, F. Dahan, J. Song, B. Carlson, and U. Ko, "A 65-nm mobile multimedia applications processor with an adaptive power management scheme to compensate for variations," in *VLSI Circuits, 2007 IEEE Symposium on*, june 2007, pp. 224 –225.

[5] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, june 2011, pp. 365 –376.

[6] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253 –266, feb. 2010.

[7] S. Jain, S. Khare, et al., and S. Borkar, "A 280mv-to-1.2v wide-operating-range ia-32 processor in 32nm cmos," in *Solid-State Circuits Conference (ISSCC), 2012 IEEE International*, feb. 2012, pp. 66 –68.

[8] L. K. 2.6.27, "Documentation/cpu-freq/," *www.kernel.org*, 2008.

[9] V. Pallipadi, S. Li, and A. Belay, "cpuidle?Do nothing, efficiently. . ." *Proceedings of the Linux Symposium*, vol. 2, 2007.

[10] S. Siddha, V. Pallipadi, and A. V. D. Ven, "Getting maximum mileage out of tickless," *Proceedings of the Linux Symposium*, vol. 2, 2007.

[11] *IEEE International Solid-State Circuits Conference, ISSCC 2010, Digest of Technical Papers, San Francisco, CA, USA, 7-11 February, 2010*. IEEE, 2010.

[12] K. Uchiyama, , F. Arakawa, H. Kasahara, T. Nojiri, H. Noda, Y. Tawara, A. Idehara, K. Iwata, and H. Shikano, "Heterogeneous Multicore Processor Technologies for Embedded Systems," *Springer New York*, 2012.

[13] Y. Wada, A. Hayashi, T. Masuura, J. Shirako, H. Nakano, H. Shikano, K. Kimura, and H. Kasahara, "A parallelizing compiler cooperative heterogeneous multicore processor architecture," *T. HiPEAC*, vol. 4, pp. 215–233, 2011.

[14] A. Hayashi, Y. Wada, T. Watanabe, T. Sekiguchi, M. Mase, J. Shirako, K. Kimura, and H. Kasahara, "Parallelizing compiler framework and api for power reduction and software productivity of real-time heterogeneous multicores," in *LCPC*, 2010, pp. 184–198.

[15] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara, "Oscar api for real-time low-power multicores and its performance on multicores and smp servers," in *LCPC*, 2009