K.G. Saur Verlag, München, 2009

J. Richling, J.H. Schönherr, G. Mühl, and M. Werner

Towards Energy-Aware Multi-Core Scheduling

Jan Richling has studied computer science and received his PhD from the Humboldt University of Berlin. Currently, he is research assistant at the Communication and Operating Systems group at the Berlin Institute of Technology. His research interests are non-functional properties (e.g., real-time, dependability, energy consumption) with special focus on composability aspects.

Jan H. Schönherr has received his diploma in computer science from the Berlin Institute of Technology in 2002. After having gained five years of industry experience, he is currently employed as a research assistant at the Communication and Operating Systems group at the Berlin Institute of Technology, working on his PhD. His research interrests are focussed on distributed and parallel systems.

Gero Mühl has received a diploma in both computer science and electrical engineering from the University of Hagen. In 2002 he received his PhD from the Darmstadt University of Technology. Currently, he is research assistant at the Communication and Operating Systems group at the Berlin Institute of Technology where he was awarded with the habilitation in 2007. His research interests lie in the field of self-organisation in distributed systems.

Matthias Werner has studied electrical engineering, control theory and computer engineering, and received his PhD from the Humboldt University of Berlin. He worked at the University of Texas at Austin, the Berlin Institute of Technology, with Microsoft Research Lab at Cambridge, and Daimler Research. His research interests are non-functional system properties, especially real-time and dependability as well as architectural and composability aspects. Currently, he is head of the Operating Systems group of the CS-Department at the University at Chemnitz.

ABSTRACT

There are two major current trends that can easily be identified in computer industry: (i) the shift towards massively parallelization fostered by multi-core technology resulting in growing numbers of cores per processor and (ii) the increasing importance of energy-awareness in computing due to rising energy costs and environmental awareness. In current operating systems, these two issues are often addressed independently: one component, the scheduler, assigns processes to cores and a second component manages the power states of individual cores in time. In this paper, we explain why this orthogonal treatment can lead to problems such as a considerably degraded system performance in case of on-demand processor power state management. Furthermore, we present an approach to energy-aware multi-core scheduling at the operating system level avoiding the performance penalty while still saving energy. To corroborate our argumentation and to illustrate the applicability of the presented approach, we give numbers from experiments based on Linux and current multi-core processors.

Index Terms
energy-aware computing, green IT, operating systems, processor scheduling, multi-core processors

I INTRODUCTION

For several decades, raising the clock frequency of processors was an important mean to increase computing power. For example, the clock frequency had been raised nearly by a factor of 1,000 from 4.77 MHz of Intel's 8086 presented in 1978 to about 3.8 GHz of Intel's Pentium IV which arrived in 2005. However, since about 2005 the situation has changed substantially because processors hit a "power wall" that made it impossible to further increase the clock frequency with the same speed as it was possible previously. Now, massive parallelism is seen as the "silver bullet" for the still rapidly growing demand for computing power. Although first implemented in IBM's Power 4 in 2001 [7], dual-cores reached mass market in mid 2005 with the introduction of AMD's Athlon X2 and Intel's Pentium D. The formula was simple because it was possible to stay within the same power envelope for two cores instead of one by reducing the clock frequency by just 200 (AMD) or 600 (Intel) MHz. This development continued, introducing quad-cores to mass market and octacores to smaller segments of it (e.g., Sun's Niagara [2]) and is still evolving. Currently, research projects target as many as 80 cores on a single chip [6], [8].

The global increase of energy costs and environmental awareness brought power consumption of computer systems into public view. Current processors usually still operate near the

PIK 32 (2009) 2
10.1515/piko.2009.0017

"power wall" and consume a considerable proportion of a computer's overall power consumption. Thus, besides the emergence of multi-core processors, a second trend of the last years was to reduce the energy consumption of desktop and server systems by incorporating technologies previously known from mobile systems, namely the ability to dynamically adapt the clock frequency and voltage of a processor to the computing demand and to disable certain units of the processor when they are temporarily not needed.

Initially, frequency scaling was done by affecting the whole chip, i.e., processor. This means that when one core, for example of an AMD K8 Athlon X2, needs to run at a high frequency, all cores have to run at this frequency and the implied high voltage. Intel Core 2 Quad processors are composed of two dual-core dies and allow two independent clock domains consisting of two cores each. However, all four cores are required to run at the same voltage because the socket has only a single power plane. AMD's "native" quad-core K10 implements individual clock generators for each core enabling them to run at different frequencies. Although these processors include two power planes, only one is used to power all cores, so they have to run at the same voltage even if they run at different frequencies; the other plane is used to feed the uncore parts of the processor. The highest frequency then determines the voltage to be supplied to all cores. All these approaches are not optimal since many of today's applications are still single threaded or, in case of a quad-core, have a parallelism less than four. In this case, idle cores are forced to run at high frequency or at least high voltage not saving as much energy as it would be possible. One possible solution, as found in Intel's Core i7, is to combine the approach of a single clock domain with the ability to turn off unutilized cores which avoids wasting energy by idling at high frequency and voltage. In multi-socket systems the situation is different in that each socket forms its own clock domain and has its own voltage regulators. In this case, at least the cores of different sockets can run at different frequencies and voltages.

The approach of dynamically adjusting the frequency – and voltage if supported – of individual cores actually saves energy. However, it can lead to severe performance problems when these issues are not adequately addressed by the scheduler. Ignoring power management, scheduling tasks on a multi-core is similar to classical SMP scheduling that makes decisions on *where* and *when* a task is executed. The question of *where* also deals with fairness and is usually answered by using the processor with the lowest load, targeting an equal distribution of load among the processors. Considering power management that is able to adjust frequencies of individual cores, a central assumption of SMP scheduling is invalidated since cores, due to their different operating frequencies, are no longer equal as it is assumed by SMP (*symmetric* multiprocessing). We will show that not considering this fundamental change can lead to severe performance degradations. The goal of our work is to address this problem and to introduce an approach to conserve energy while not degrading performance.

The remainder of this paper is organized as follows: In Section II, we further introduce the problem and illustrate it using a real-world example and corresponding measurements as well as considerations of existing solutions in current processors. We continue with related work in Section III and present an approach of energy-aware multi-core scheduling in Section IV. Its potential is evaluated in Section V with initial experiments. Finally, our conclusions are presented.

II PROBLEM STATEMENT

If the objective is to optimize the ratio between useful amount of computation and total power spent (i.e., the computational performance per Watt for the whole computer, not just the processor), the best solution is often to set all cores to the highest frequency possible. Then, the calculation can be finished rather early allowing to switch off the whole computer afterwards. However, this scenario is not very realistic since many computers (especially servers) run 8 to 24 hours per day, where most cores are idle most of the time. Following this observation, the objective should be to save as much energy as possible while avoiding a performance penalty when executing applications. However, current implementations of power management fail to achieve this goal and can cost a remarkable amount of computing performance as shown next.

Imagine a situation in a multi-core system, where the level of application parallelism is less than the number of cores and where power management (i.e., frequency and voltage scaling) is applied per core.[1] In this case, some cores are utilized running at a high frequency and voltage to maximize performance, while others are idle running at the lowest frequency and voltage to save energy. Whenever a new process is created, most operating system schedulers assign it to the core with the longest idle time due to load balancing and fairness. As a consequence, this core increases its clock frequency and voltage. This process takes some time (e.g., about 600 µs in AMD's K8 processor and about 100 µs in current quad-cores from both AMD and Intel) because the hardware must increase voltage and frequency in small increments stabilizing inbetween. Although some processors such as Intel's Core 2 are able to execute code for certain periods of the transition (e.g., while the voltage is ramped up but before the frequency is raised), the processor runs at the desired frequency only when the transition has finally finished. Furthermore, the decision to change to a higher power state is usually not taken by the scheduler but a separate component. In Linux, for example, a governor periodically polls the load of individual cores and decides on switching clock rates – for Linux 2.6.25 on an AMD Phenom quad-core of the first generation and many Intel Core 2 processors this polling interval defaults to 200 ms with a minimum of 100 ms, on an AMD Phenom II quadcore it defaults to 80 ms with a minimum of 40 ms, while for an AMD K8 it defaults to 1240 ms with a minimum of 620 ms. Therefore, the overall switching time also includes the time it takes to actually recognize the higher load. As modern processors continue operation while switching and switching times are decreasing, the polling interval becomes the main part. This is not a severe problem if the runtime or length of activity phases of an average task is much higher than the overall switching time as the processor is then utilized enough to prevent a transition back to a lower power state. Also, it is not a problem if the overall load is high enough to keep all cores at maximum frequency. If, however, both assumptions do not hold (i.e., the system is executing an application with an average parallelism less than the number of cores and individual tasks have runtimes or activity phases in the order of the overall time needed for frequency switching) the system suffers from a performance degradation.

This degradation is actually caused by a combination of two effects. The first one, which we will refer to as Problem A, is that

1 The following argumentation also holds if only frequency scaling is applied per core as in case of AMD's K10 because voltage scaling does not affect performance.

new workload is assigned to cores residing in the lowest power state and is, thus, executed slow (until the load, if kept up long enough, is recognized and these cores actually reach a higher power state). The second effect, Problem B, relates to inaccurate load measurement by the governor: It is very likely that a core became active or inactive somewhere in the middle of the polling interval. Therefore, the average load during the last polling interval, which is measured by the governor, does not reflect the actual value. This, in turn, might cause the governor to make wrong decisions. For instance, if a scheduler reassigns long-running tasks to different cores as it can be observed with Windows XP, Windows Vista, and Windows 7 Beta, this can even lead to a single-threaded application running permanently with the lowest available frequency on a quad-core. Again, the degradation becomes worse with low parallelism and short jobs, as none of these jobs might take long enough in order to trigger a transition to a higher power state.

A prominent example of an application demonstrating this degradation is the compilation of a large piece of software, composed of hundreds or thousands of individual source files. Dependencies among those files restrict parallelism, meanwhile modern processors are able to compile average source files in fractions of a second. A good test scenario for this kind of application is the Linux kernel. It can be compiled parallelized with restricted parallelism (`make -jN`). Furthermore, it consists mainly of C-code that can be compiled very fast. Fig. 1 shows the results of a simple experiment to demonstrate the performance degradation due to both problems identified before. A Linux kernel is compiled using the same configuration and different make-parallelisms (from 1 to 12, further increasing the parallelism slows down the compilation again on our test machine) on a dual quad-core (8-way) machine with one core per clock-domain. In order to avoid impact of caching and I/O, compilation takes place on a RAM disk and each run is executed three times. For each parallelism, the compilation is executed with (a) the ondemand governor, which switches frequencies of individual cores of the two AMD Opteron 2352 used for this experiment dynamically between the lowest (1.05 GHz) and highest frequency (2.1 GHz) depending on the load of a core, and (b) the performance governor, which keeps all eight cores at the highest frequency. The polling interval of the on-demand governor is kept at the default of 200 ms which is similar or greater than the time needed to compile most of the source files.

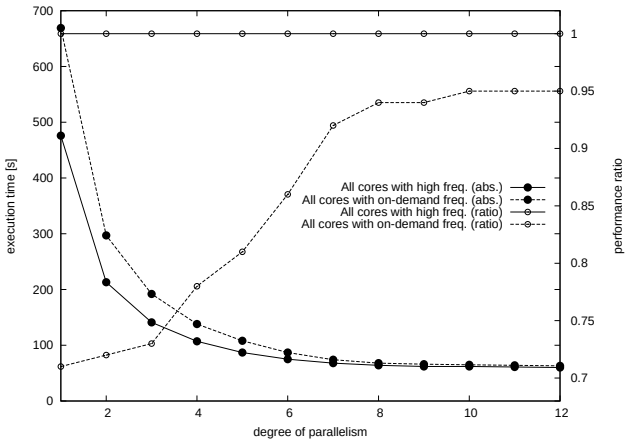

Fig. 1 Compile times of a Linux kernel with different frequency governors on a dual AMD Opteron 2352 system (dual quad-core, 2.1 GHz highest frequency, 1.05 GHz lowest frequency)

The figure clearly shows the performance degrading effect described above. Furthermore, it also shows that the effect is more pronounced if there is a large distance between the number of available cores and the parallelism of the application (in this case consisting of many small parts that are executed as individual processes). This again proves the importance of solving these problems: Surfacing already on current quad-cores, it will certainly get worse in future as it can be expected that the gap between number of cores and parallelism of such applications widens because of an increasing number of cores. Parallelism, on the other hand, is mostly always limited by dependencies among different tasks or threads of an application. This is also true for the experiment described above: Although most of the time even a parallelism of 12 can be seen, there are also intervals during kernel compilation with a parallelism much smaller. Results showing a similar performance degradation can also be derived for Windows. For example on Windows Server 2003 R2, the execution of a single-threaded compilation is delayed by 26% on a single quad-core machine using individual frequencies per core.

As Intel's Core i7 has only one clock domain with the additional ability to halt an unused core completely, this eliminates the impact of Problem A nearly completely in case of single-socket systems. However, it does not solve Problem B: For a processor with only one clock domain, the frequency is derived by calculating the maximum of the loads of the individual cores during the last interval. In case of short running processes with a low parallelism, this again results in a too low maximum and therefore too low new frequency. Initial tests show that this problem really exists: A Core i7 940 quadcore (Turbo Boost and Hyper-Threading disabled) compiles a Linux kernel with a parallelism of one with the on-demand governor 12% slower compared to the performance governor. For an AMD Phenom 9750 processor with individual clock domains, however, the slowdown is 27% (low frequency is half of high frequency). First tests with AMD's new Phenom II 940 show that the problem even increases to 34% due to a higher distance between high and low frequency (3 GHz vs. 800 MHz). Therefore, this comparison enables us to differentiate between the effects of Problem B alone (Core i7) and Problem A and Problem B together (AMD K10). It also shows that restricting the number of clock domains to one does not solve all problems. Especially, in case of multi-socket systems Problem A still exists even with only one clock domain per socket.

III RELATED WORK

Although the area of energy awareness in general and energy-efficient scheduling in particular is rather new – the first papers on this subject dates from the mid 1990s – there exists already an extensive number of publications in this area.

Since sensor (and actor) networks are systems that are especially energy-critical, a remarkable amount of research is done in this area. For an overview see [9]. The objective of a sensor network is in most cases to perform its duty as long as possible with its limited amount of energy. This is achieved by putting as many sensor nodes as possible into adequate sleep modes while still keeping the network's functionality as required, i.e., some predefined QoS objectives. Also, tasks requiring energy may be rotated among multiple nodes to prolong the lifetime of individual sensor nodes and, therefore, the complete network. In our scenario, we do not have to deal with limited energy. Nevertheless, processors as Intel's Core i7 allow to switch off

individual cores completely, thus, we are presented with a similar problem not network wide, but in a single multi-core processor (or a set of multi-core processors inside a system).

Focusing on single processors, energy can be saved by scaling down the processor's frequency and voltage. This has been researched for real-time systems, where it is possible to lower down these settings to the point where given deadlines are just fulfilled [5]. A similar approach is also possible for non real-time systems: Process Cruise Control [10] adjusts the processor's frequency for individual threads. Depending on the characteristics of a thread, it is possible to save energy without a significant loss of performance (e.g., slowing down memory intensive applications and thereby masking the memory latency partly). Such approaches are orthogonal to our problem as we focus on multiple processors or cores. Indeed, it is possible to realize them on top of our proposed solution.

In [4] tasks are scheduled on multiprocessor systems in a way that the energy consumption is evenly distributed among all processors. This avoids temperature peaks in individual processors, allowing savings in the cooling infrastructure without performance losses due to processor throttling. This is contrary to our approach (cf. Section IV) as we concentrate workload on only a subset of processors to slow down others and, thus, save energy efficiently, i.e., without impacting performance. Still, it is possible to adapt this temperature awareness to complement our energy awareness.

Scheduling on multiprocessor architectures, where individual processors run at individual but fixed speeds, is considered in [3]. Similar to our approach, they ascertain that today's schedulers cannot handle processors running at different frequencies well. Our scenario differs in that we are able to control the frequencies of individual processors. Therefore, we deal with dynamic asymmetry instead of static asymmetry and use both the assignment of tasks to cores as well as the ability to influence that asymmetry as means for our solution.

Bircher and John [1] analyze the effect of dynamic power management on multi-core processors on performance and energy consumption. Similar to us, they observe that the scheduling of an operating system can have a negative effect on performance when dynamic power saving is applied. However, they state that the main cause for performance degradation is the interaction between active and idle cores caused by cache probing. As cache probing only occurs between cores that share some data, this problem is avoided by our approach because we concentrate the workload on a subset of cores (running at high frequency) and therefore limit cache probe traffic in a way that it only occurs between cores at high frequency. Furthermore, a main cause for this problem is eliminated by latest-generation processors: Due to inclusive L3 cache (Intel) and copying L2 cache content of inactive cores to L3 cache (AMD) cache probing becomes independent of inactive cores.

IV SCHEDULING

Energy consumption is a non-functional property. Such properties are often neglected in the first place, and realized only subsequently. This can also be seen in the way operating systems such as Linux deal with controlling core frequencies: A governor monitors the behavior of cores and their load using polling and makes decisions based on those observations. Obviously, this idea follows a strict "divide-and-conquer"-approach separating the (functional independent) issues of scheduling tasks and controlling energy consumption. However, this introduces problems such as those described in Section II. Moreover, it also increases response times for interactive users, because an increase of frequency always follows an increase of load. To be more precisely, it occurs after the governor has figured out by polling that load has increased. Tasks that require high computation power for a very limited amount of time are therefore finished before frequency increases. This becomes even worse with larger polling intervals. E.g., Linux's on-demand governor uses a default interval of more than one second on an AMD K8 processor (this is because the default interval is 2,000 times the transition latency of the CPU in order to limit load introduced by polling – a K8 has a much larger transition latency than, e.g., a K10 or an Intel Core 2[2]).

The non-functional problem described above could partially be attenuated by increasing the polling frequency of the governor. However, this also increases the overhead. It is our believe that the problem has to be targeted at the point where additional load is recognized first (i.e., inside the scheduler) and not by reconstructing information outside the scheduler that is already available inside of it. Especially, this way Problem B is targeted directly.

In the following, we introduce our approach of defining high/low-sets in Section IV-A. Next, we give the sketch of our approach in Section IV-B together with ideas for additional optimizations in Section IV-C. Section V presents results of three initial experiments showing the applicability of our ideas.

A High/Low-Sets

The effects described in Section II can be shown very clearly by monitoring individual core frequencies during kernel compilation with a parallelism less than the number of cores (`make –j`*N*, with *N* less than the number of cores) using the on-demand governor. Moreover, it can be seen that there are always two sets of cores: one set with low frequency and one with high or highest frequency. However, as described in Section II, the membership is constantly changing and does not necessarily reflect the actual load distribution. This is due to the fact that frequencies are controlled based on local observations (load of individual cores) and the impact of a fairness-based task distribution.

This leads to the basic idea of our approach: We define two sets of cores and run cores from one set (high set, called *H*) always at the maximum frequency and cores from the other set (low set, called *L*) always at the lowest frequency. Workload is only scheduled on cores from the high set. Instead of managing frequencies locally, we now change set membership based on information regarding actual parallelism inside the scheduler. That way, membership only changes if parallelism changes – in case of an application that requires constant parallelism nothing changes at all avoiding the problems described in Section II.

In a multi-socket system this approach can, however, cause asymmetric heat distribution inside the computer resulting in noise (one fan running at maximum speed while others are

2 Repeating the experiment from Section II using the default polling interval of an K8 (1240 ms) leads to results much worse. However, as K8 dual-cores cannot change frequencies of its two cores independently, this mainly matters for multi-socket systems – single-socket K8 systems are affected by Problem B only.

turned off) or shorter lifetimes of individual fans. This is because we abandon the fairness-based approach that considers all cores and tries to use them equally. Instead, we introduce asymmetry up to a point where one core does all work while others are idling. However, this minor problem can be solved easily by changing set membership depending on temperature or by applying a long-term fairness scheme that reassigns cores at intervals in the order of minutes. Another option would be to use the results from work such as [4] for assignment inside the high set.

A further point in favor of our approach is that cores from the set *L* are idling for long intervals. This gives us the additional freedom of completely disabling cores in set *L* if supported by hardware (such as for Core i7). Although this is also possible using the fairness-based approach, it can be assumed that reenabling a core needs some time which again leads to an effect similar to Problem A described in Section II.

The main challenge of the set-based approach is to develop an algorithm that effectively decides on set membership.

B Approach

To realize our idea we have to implement three mechanisms that closely cooperate and that are part of the operating system scheduler:

- **Set Control:** First, the sets *H* and *L* have to be established. In order to do this, we have to solve two subproblems:
  - A reasonable size of the set *H* has to be derived from the given task set.
  - The question which core should enter or leave *H* in case of changes in set size has to be solved. From a theoretical point of view, this can be done randomly, but there are practical arguments in favor of more deterministic selections.
  - **Task Assignment:** The second mechanism implements SMP-scheduling of the given task set on the cores of set *H*. Here, we propose to use the regular scheduler of the operating system with the adoption that the size of the processor set changes over time.
  - **Frequency Control:** Finally, the frequencies of all cores have to be adopted based on their assignments to sets *H* and *L*. This is based on the size of *H* defined by set control and uses the existing functions to control frequencies and voltages of cores.

Task assignment and frequency control can use well-established approaches. Selecting set members based on set size seems to be a minor problem that we address in Section V-F. Therefore, we concentrate on the question of calculating a reasonable size of *H* at runtime using a given task set and their parameters in the following.

Currently, we are considering two approaches:

- **On-demand with spare:** The obvious approach is to change the size of *H* according to the actual demand. However, for the reasons discussed in Section II it is not sufficient to let the power management governor determine the demand using polling. The threads of the applications in consideration would have done most of their work before the governor is even informed about their existence.

  Load is introduced by threads becoming active. A (new) thread becomes active if it enters the ready state, i.e., as an result of a thread creation (`fork`, `clone` in case of Linux) or a deblock operation. By instrumenting these state change operations, we determine the needed size of *H* at the time additional load occurs. To avoid a negative impact of frequency transition itself, we choose the size of *H* as a value that is always one more than currently needed, i.e., we add a spare core. That way, we introduce a small additional waste of energy for the benefit of a better response time that is similar to the respond time of the symmetric case with all cores running at maximum frequency.

  In order to maintain the size of *H* according to actual demand, it is also necessary to instrument the opposite state change operations. However, with respect to response times this operation is not as time critical as increasing *H*. For energy consumption, the opposite is true.
- **Support by application:** The best a-priori information on the degree of parallelism of an application is probably owned by the application's developer. Our second approach is to keep this information as metadata. When an application starts or the application's degree of parallelism changes, it is transmitted to the operating system. In our current design, there is a special system call for that purpose. Nevertheless, it could be possible to "enrich" an existing call with an appropriate parameter. In this way, we can avoid the spare core in *H*. However, this approach requires support by the programmer (or development tools, if the metadata can be generated automatically) and is, thus, not feasible in short range. Moreover, for optimal operation all applications should behave this way.

C Optimizations

As an extension to the approach described above, it is possible to define a third set ("green set") of cores which are operating at the most energy efficient frequency/voltage pair for given tasks. This information can, for example, be derived using Process Cruise Control [10].

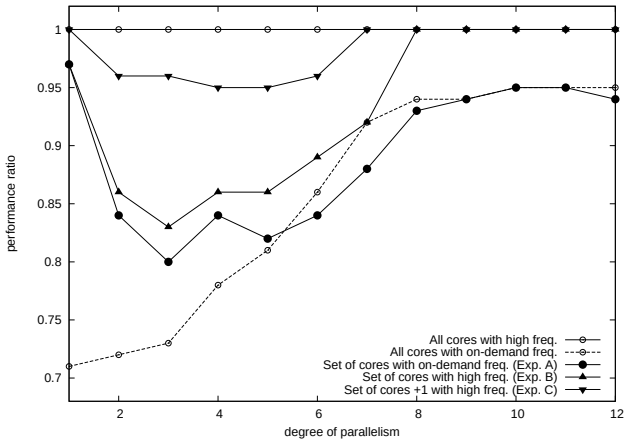

Fig. 2 Performance ratios of execution times of `make -jN` *in dependence of N for different policies. Other parameters are the same as in Fig. 1.*

This approach can be varied based on the understanding of energy-efficiency: If a system is never turned off, energy efficiency is calculated based on the difference in energy consumption compared to idle mode, while the energy consumption of the whole system has to be considered if a system is turned off af-

ter computation. For most processors, in the first case the lowest frequency is most efficient while for the latter a higher frequency is better.

V EXPERIMENTAL RESULTS AND DISCUSSION

In order to show the applicability of our approach, we present the results of three initial experiments that target at the example from Section II by applying the approaches described in Section IV-B manually. Furthermore, we examine Problem B in isolation as seen on processors with a single clock domain in Section V-E. We are aware that these experiments cover only a small subset of possible applications and do not address other issues such as locality which will be discussed in Section V-F. For future work, we plan to conduct more experiments based on an implementation of the presented approach inside the Linux scheduler.

Our goal is to improve performance in cases where parallelism is lower than the number of cores and not all cores run at their maximum frequency in order to save energy. Therefore, running all cores at their highest frequency defines the base case for our comparison with respect to performance. Fig. 2 depicts the results of the experiments compared to this base case and to the results of the on-demand frequency governor from Fig. 1.

The application is again compiling a Linux kernel (using the same parameters as in Section II) with different parallelism defined by the parameter N of `make –jN`. This is similar to the second approach in Section IV-B because the known application parallelism is used to estimate the size of set *H*.

A Simple Sets with polling-based on-demand governor

Experiment A evaluates the idea of having two sets in a simple way. It applies the original polling-based on-demand governor to all cores while restricting the execution of tasks to the processor cores in set *H*. The execution restriction is achieved by the command `taskset` which defines an affinity mask for the compilation processes containing only cores in *H*.

The results of Experiment A depicted in Fig. 2 show that this approach behaves significantly better than the original on-demand approach for small parallelisms, while still being slower than the performance achieved with the performance governor which runs all cores at the highest frequency possible. For parallelisms between 6 and 7, it is a little worse than the on-demand approach without core restrictions. This may be caused by `make` conflicting with the compiler processes it starts while being limited in the number of cores. With our approach this maps to a not accurately estimated parallelism resulting in a too small size of set *H*. For parallelisms from 8 onwards, the approach is equal to the on-demand approach (which is obvious because set *H* includes all cores in that case), but still worse than the performance governor because phases with lower parallelism are still affected by the problems identified in Section II.

B High/Low Sets

Experiment B completely removes the polling-based on-demand governor and directly implements the idea of high/low sets as introduced in Section IV-A: all cores from set *L* run always at lowest frequency (1.05 GHz in case of our test machine), while cores from set *H* are pinned to the highest frequency (2.1 GHz). Similar to the first experiment, `taskset` is used to restrict process execution to set *H*.

The results of Experiment B depicted in Fig. 2 show that there is an improvement compared to Experiment A in the range of parallelism from 2 to 7. This is due to the fact that problems caused by transition delays are now avoided as frequencies are not switched during the experiment. However, there is still a gap to the optimum that has probably the same origin as described in the previous section: Set *H* is too small to provide enough computing power to both `make` and the compiler processes. For a parallelism of 8 or higher, this problem vanishes and the optimum is reached. This is obvious because in this case all cores are used running at their maximum frequency.

However, the results also show that this approach performs always better or equal to the original on-demand governor while still saving energy by not running all cores at their maximum frequency.

C High/Low Sets with Spare

Experiment C applies the idea of adding a spare as described in Section IV-B. Again, we assume the parallelism to be known and equal to the parameter *N* of `make –jN`. This is done by repeating Experiment B with an appropriate change in size of set *H* (namely *N*+1 cores).

This approach improves performance significantly by reducing the gap to the optimum to 5% and less (Experiment C in Fig. 2). For parallelism of 1 and 7 and larger it even reaches the optimum. For parallelism in range from 2 to 6 that led to worst performance degradations in Experiments A and B, a significant improvement is achieved.

The existence of this 5% performance gap, however, shows that the simple approach of estimating the parallelism by using the parameter *N* of `make –jN` is not sufficient in all cases. Therefore, more fine grain means of adjusting the size of set *H* as described in Section IV-B are needed.

Table I Energy consumption of `make –j1` *for different policies.*

Scenario	Run-time	Power	Energy
Performance Governor	6:04 min	136 W	13.77 Wh
High/Low Set	6:11 min	121 W	12.55 Wh
High/Low Set with Spare	6:04 min	126 W	12.80 Wh
On-Demand Governor	7:44 min	113 W	14.64 Wh
All cores low frequency	11:47 min	79 W	15.57 Wh

D Energy Consumption

The final goal of our work is to save energy with no or almost no impact on performance. So far, we have evaluated the impact on performance with the presented experiments. In order to give initial results on energy consumption, we ran a set of experiments using an AMD Phenom 9750 processor measuring the energy consumption of the whole system. In future, we will extent this by more fine grained measurements (e.g., the processor isolated). Table I shows the results of compiling a kernel

with a parallelism of one in five different scenarios: (1) performance governor, (2) “High/Low Set” (with one core in the high set), (3) “High/Low Set with spare” (two cores), (4) standard on-demand governor and, for comparison, (5) running all four cores at low frequency. It can be seen that “High/Low Set” leads to the lowest energy needed for finishing the task although it is neither the fastest nor has the lowest power requirements. Therefore, this approach delivers the best performance per watt ratio. Furthermore, “High/Low Set with spare” performs equal to performance governor but decreases energy consumption.

E Processors with a single Clock Domain

In order to isolate the effects of the two problems described in Section II, we executed some initial experiments using an Intel Core i7 940 with Turbo Boost and Hyper-Threading disabled. A Core i7 has only a single clock domain, so all cores must run at the same frequency. However, individual cores have the ability to halt completely resulting in a situation similar to Problem A due to the time required to resume execution if a task is scheduled on a halted core. However, as re-enabling is done in hardware, the effect is expected to be neglectable compared to the delay caused by software governors. Our first results confirm this: using the on-demand governor instead of the performance governor to compile a kernel with a parallelism of one decreases performance by 12% due to Problem B. “High/Low Set”, on the other hand, performs similar to the performance governor and “High/Low Set with spare” even slightly better (around 0.5-1%). Therefore, our approach is suited both for solving Problem B as well as addressing processors such as Intel’s Core i7 that disable individual cores if unloaded. With enabled Turbo Boost, “High/Low with spare” even outperforms the performance governor by 3% as the number of active cores is reduced and therefore the efficiency of Turbo Boost gets increased due to improved locality of execution.

F Further Aspects

As mentioned in Section IV-B, there are further aspects that have to be considered. Here, we mainly approach two problem domains: The first one deals with fairness aspects and related side conditions of the assignment of cores to the set *H* (with a given size of *H*), while the second considers application aspects that are not covered by our scenario of having a large number of short-living processes (or, by having processes that belong only for short time intervals to the active set).

Although we give up the approach of being fair, this is only true with respect to the workload distribution among individual processor cores. Fairness among processes is still present. Especially, our approach does not affect the assignment of a task to a core within the set *H*, it only restricts the available cores from all cores to members of the set. Therefore, in case of a fully loaded machine our approach is not any different from the established approach. Dropping fairness regarding cores does only matter for aspects such as cooling or memory access in case of NUMA machines. However, both problems can be solved easily by influencing the decision which cores enter or leave set *H* – we already discussed the issue of cooling in Section IV-A. With respect to a NUMA system, it seems reasonable to construct *H* according to application requirements. For processes that are memory-independent, it makes sense to represent all NUMA partitions more or less equally in *H*.

Another issue related to the assignment of cores to set *H* is cache locality. One problem area of our approach are scenarios where the overall number of running processes is larger than the number of cores but having a much smaller parallelism (this applies, e.g., to I/O-intensive processes). In this case, our approach introduces more task switches on individual cores increasing the number of cache misses. On the other hand, there are also cases where restricting the number of used cores improves cache efficiency by locality.

Beside the assignment of cores to *H*, we target onto special use patterns, namely, a large number of processes being only active for a very short time. However, as the runtime of processes is usually not known in advance, the impact onto other scenarios is important. For long running processes the problems introduced in Section II do not apply as it is simple to execute them on the same core in case of parallelism less or equal than the number of cores. On the other hand, even this seems to be an issue on operating systems (e.g., Windows), where tasks “jump” frequently between cores. In such cases, our approach of dropping fairness may also improve behavior of long running processes.

Another scenario that we consider is interactive usage. Due to the power of modern computers, this kind of usage usually does not introduce high load on processors causing them to switch to low frequencies. If the polling interval of a frequency governor is relatively large, interactive usage may even not lead to a higher frequency, so that user initiated actions are executed at low frequency only. While this does not matter in many cases, reducing speed of a “visible” action to half can be noticed by most humans. In fact, using an interactive application on a K8 using the on-demand governor with a polling interval of 1240 ms “feels” less responsive than running the machine at its highest frequency all the time. It can be expected that many users disabled power management due to this reason. Since interactive processes usually stay in active state only for a limited time (in response to user actions), they are also appropriate for our approach. This way, such applications are assigned to a core from set *H* and are in consequence executed at highest frequency. This leads to faster response times at the cost of slightly increased energy consumption as at least one core must be kept at high frequency.

VI CONCLUSIONS

In this paper, we have addressed problems resulting from the fact that, in current operating systems, scheduling tasks to cores and power management per core are managed by different components although these issues are not independent of each other. In lightly loaded systems, these problems lead to a significantly degraded performance compared to a disabled power management. They will even become more prominent in future with an increasing number of cores per processor because it can be expected that application parallelism will not raise with the same speed. To tackle these problems, we propose to treat power state management (i.e., frequency and voltage scaling) together with task assignment inside the scheduler. This way, we can react to new load demands directly when they are injected into the system, thereby avoiding the delay caused by the usual polling approach.

As a solution that can easily be integrated in current operating systems, we propose to give up fairness and to apply a set-based approach which partitions the cores into two sets running

at low and high frequency/voltage pairs, respectively. We react to changing application parallelism by adapting the size of these sets. This is done inside the scheduler enabling fast and fine-grained reactions. The experiments we conducted with an initial version of our approach show that the performance degradation can nearly be avoided while still saving energy. Furthermore, by having a spare core in the high set, we improve interactive response times noticeably.

Currently, the energy saving of our approach mainly stems from the difference between idle consumptions at lowest and highest frequency. However, as latest-generation processors allow to disable cores completely, we expect larger energy savings for such processors with even smaller impacts on performance. Our approach ideally fits such a scenario as we do not suffer from the delay to reactivate a core.

REFERENCES

[1] W. Lloyd Bircher and Lizy K. John: Analysis of dynamic power management on multi-core processors. In: ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, pages 327-338, New York, NY, USA, 2008. ACM.

[2] Poonacha Kongetira; Kathirgamar Aingaran; and Kunle Olukotun: Niagara: A 32-way multithreaded sparc processor. IEEE Micro, 25(2):21-29, 2005.

[3] Tong Li; Dan Baumberger; David A. Koufaty; and Scott Hahn: Efficient operating system scheduling for performance-asymmetric multi-core architectures. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pages 1-11, New York, NY, USA, 2007. ACM.

[4] Andreas Merkel and Frank Bellosa: Balancing power consumption in multiprocessor systems. SIGOPS Oper. Syst. Rev., 40(4):403-414, 2006.

[5] Padmanabhan Pillai and Kang G. Shin: Real-time dynamic voltage scaling for low-power embedded operating systems. SIGOPS Oper. Syst. Rev., 35(5):89-102, 2001.

[6] Larry Seiler; Doug Carmean; Eric Sprangle; Tom Forsyth; Michael Abrash; Pradeep Dubey; Stephen Junkins; Adam Lake; Jeremy Sugerman; Robert Cavin; Roger Espasa; Ed Grochowski; Toni Juan; and Pat Hanrahan: Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph., 27(3):1-15, 2008.

[7] Joel M. Tendler; J. Steve Dodson; J.S. Fields Jr.; Hung Le; and Balaram Sinharoy: Power4 system microarchitecture. IBM Journal of Research and Development, 46(1):5-26, 2002.

[8] S. Vangal; J. Howard; G. Ruhl; S. Dighe; H. Wilson; J. Tschanz; D. Finan; P. Iyer; A. Singh; T. Jacob; S. Jain; S. Venkataraman; Y. Hoskote; and N. Borkar: An 80-tile 1.28tflops network-on-chip in 65nm cmos. Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International, pages 98-589, Feb. 2007.

[9] Lan Wang and Yang Xiao: A survey of energy-efficient scheduling mechanisms in sensor networks. Mob. Netw. Appl., 11(5):723-740, 2006.

[10] Andreas Weissel and Frank Bellosa: Process cruise control: event-driven clock scaling for dynamic power management. In: CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, pages 238-246, New York, NY, USA, 2002. ACM.