Load-aware Scheduling for Heterogeneous Multi-core Systems

Mohannad Nabelsee Anselm Busse
Technische Universität Berlin
Berlin, Germany
{nabelsee,anselm.busse}@tu-berlin.de

Helge Parzyjegla Gero Mühl
Universität Rostock
Rostock, Germany
{helge.parzyjegla,gero.muehl}@uni-rostock.de

ABSTRACT

Heterogeneous multi-core systems are becoming more and more common today. To be used to their full potential, the operating system has to be adapted to the new system environment. This is especially true for the scheduler as it is crucial to the overall system performance. In this paper, we present a scheduling approach for heterogeneous systems with two different kinds of cores. One that is very power efficient, but shows only a limited computing power, and the other one that has a very high performance and is very power consuming at the same time. We consider such heterogeneity for a centralized scheduler architecture.

In our approach, we introduce a new load metric in order to classify tasks whether or not they are suited to be executed on a high-performance core. Based on this metric, we present a task state model for scheduling tasks according to their performance classification. We implemented the scheduling approach by extending the Brain Fuck Scheduler (BFS) and evaluated it on an eight core heterogeneous architecture with four low performance and four high-performance cores. The evaluation covers system responsiveness and high load behaviour compared to the vanilla BFS and the decentralized Completely Fair Scheduler (CFS). Even though our approach takes the heterogeneity into account, the results show that it scales better than the vanilla BFS while nearly maintaining its superior responsiveness.

CCS Concepts

•**Software and its engineering** → **Scheduling;** *Multiprocessing / multiprogramming / multitasking;* •**General and reference** → **Metrics;** *Performance;* •**Computer systems organization** → *Multicore architectures;*

SAC 2016, April 04 - 08, 2016, Pisa, Italy
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3739-7/16/04...$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851738

Keywords

Centralized scheduling; Task state model; Heterogeneous multi-core architecture

1. INTRODUCTION

In recent years, two important trends emerged in the development of microprocessors: an increasing number of processing units as well as their differentiation and specialization. While the first trend is reflected by a growing number of processing cores per CPU, the second leads to an increasing interest in heterogeneous systems and system configurations. This heterogeneity takes many forms ranging from multiple processing cores with minor differences in power consumption and/or performance up to processing units based on entirely different instruction set architectures that support alternative programming models. Thus, when combined, the overall system performance is not only gained from the sheer number of available processing cores, but also from the incorporation of specialized processing capabilities tailored to handle particular tasks much more efficiently.

In this paper, we focus on heterogeneous multi-core systems in which cores differ in performance and energy efficiency, e.g., systems based on the ARM big.LITTLE architecture [9] or the nVidia Tegra SoC [16]. These architectures are intended for modern mobile devices (e.g., smartphones, tablet computers) and couple slower, energy-efficient low-performance cores (LPCs) with more powerful and power-hungry high-performance cores (HPCs). This way, peak-performance can be delivered by HPCs for important foreground tasks, while energy is saved when executing remaining background tasks on more efficient LPCs in order to extend the battery life. For this to work, however, the OS scheduler has to assign the right processes to the most appropriate cores, i.e., LPCs or HPCs, in the proper situation.

This paper presents a load-aware scheduling algorithm that distinguishes between low and high performance tasks in order to assign them to different powerful and energy-efficient processing cores. To the best of our knowledge, this is the first *centralized* scheduling approach for such heterogeneous multi-core systems. We primarily profit from the scheduler's global run-queue which notably simplifies process classification and management and which provides more than sufficient scalability to cope with the limited number of processing cores typical for mobile devices. In particular, the paper makes the following contributions:

- a metric for measuring and classifying the performance demand of a task,
- a task state model for a centralized scheduler based on this performance classification of tasks, and
- an implementation and evaluation of such a load-aware scheduler for heterogeneous multi-core systems.

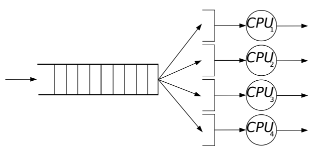
The remainder of the paper is structured as follows: Section 2 compares centralized and decentralized scheduling approaches and explains our decision for a central scheduler based on a global run-queue. In Sect. 3, we introduce a metric for measuring and classifying the performance demand of tasks in order to assign them to HPCs or LPCs, respectively. Furthermore, we extend the scheduler's task state model to make it load-aware with respect to heterogeneous performance demands and processing cores. Section 4 outlines implementation details to integrate the load-aware scheduling concept into the Brain Fuck Scheduler (BFS). Section 5 presents evaluation results based on several benchmarks while our findings are discussed in Sect. 6. Finally, Sect. 7 surveys related work and Sect. 8 concludes the paper.

2. SCHEDULING

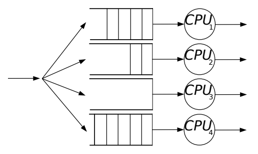
Scheduling is one of the core functions of an OS: tasks (i.e., threads and processes) are assigned to available processing units for execution. Please note that we use the terms thread, process, and task as an equivalent for a unit of execution considered by the scheduler throughout this paper. If a distinction is necessary, we note it explicitly. Usually, executable tasks are organized in a run-queue from which one task has to be chosen to be executed next. The task is assigned to a processing unit and runs until it finishes, it gets blocked (e.g., by an I/O operation), or it has expended all of its time slices and is reinserted in the run-queue to be continued later. Then, the next task is selected from the run-queue for execution.

In case of a multi-processor / multi-core system, a single global run-queue may be sufficient and leads to a *centralized* scheduling approach. Figure 1a shows four processing units that are fed by the same global run-queue. This works well when considering the following two aspects. The first aspect is *cache affinity.* When a task was executed on a particular CPU, it is usually beneficial to reassign it to the same processor as it may find parts of its data still available in the processor's cache. For this reason, cache affinity needs to be modeled explicitly and taken into account to preferentially select certain tasks for particular CPUs. The second aspect is *lock contention* since accessing and modifying the run-queue has to be done under mutual exclusion. Thus, the more processing units are served by the same run-queue, the higher the lock contention which considerably limits the scalability of a centralized scheduler.

The alternative design is a *decentralized* scheduling approach as depicted in Fig. 1b in which each processing unit has its specific run-queue eliminating the main reason for lock contention. Moreover, when returning a task into the same run-queue from which it was chosen, cache affinity comes for free. However, there is a price to pay as an explicit *load balancing* mechanism is now required when some run-queues are nearly empty while others are fully loaded. Please note



(a) Centralized run-queue.



(b) Distributed run-queues.

Figure 1: Scheduler organization for four cores with centralized scheduling (left) and decentralized scheduling (right).

that this also requires locks for synchronizing the access to different run-queues. In fact, a major part of the complexity of a decentralized scheduler is caused by the implementation of a reasonable and efficient load balancing scheme.

As we target mobile devices with a limited number of heterogeneous cores, we do not expect scalability issues caused by lock contention as the main challenge which would justify the complexity of a decentralized approach. Hence, we are heading for a centralized scheduler with a global run-queue.

3. LOAD AWARENESS

The main challenge of systems with HPCs and LPCs is the classification which task is best suited to be executed on which core. On the one hand, a task with a low performance demand should not be assigned to any HPC in order to avoid the waste of energy as the task under-utilizes these cores. On the other hand, a task with a high performance demand should not be scheduled on any LPC in order to achieve an adequate system performance.

3.1 Performance History

To classify tasks, load tracking needs to be introduced into the (centralized) scheduler. In our approach, load is defined by the time a task occupies a core during a certain period. We denote this kind of load as performance history.

3.1.1 *Basic Performance History*

For the performance history of a task, we consider the time the task spent running on a core as well as the time expended in a blocked state while waiting for a signal such as the completion of an I/O operation. The time the task spends waiting in the run-queue to get assigned to a core for execution is not considered. The reason is that the time waiting in the run-queue is not contributed to the behavior of the task itself, but rather to the system's load situation.

For each task, we consider three points in time:

- t_{select}, when the task is selected by a core,
- t_{evict}, when the task is evicted from the core, and
- t_{queue}, when the task is inserted into the run-queue.

Then, the *run time* of the task is calculated as the time between running the task and evicting it from the core ($t_{evict} - t_{select}$), while the *blocked time* is given as the time between evicting the task from the core when blocked and reinserting it into the run-queue when unblocked ($t_{queue} - t_{evict}$). Run time intervals as well as blocked time intervals are accumulated until they reach the length of a measurement epoch with a predefined value:

$$\sum run\ times + \sum blocked\ times \geq epoch\ length.$$

The performance value p_i of epoch e_i with $i \geq 0$ is then defined as

$$p_i = \frac{\sum run\ times\ of\ epoch\ e_i}{epoch\ length}.$$

Separating the measurements into epochs allows to compute a running average that weights recent epochs stronger than more distant ones. This performance history value p_H is obtained by an exponential smoothing of the performance values with a smoothing factor $0 < \alpha < 1$:

$$P_{H_i} = \alpha p_i + \alpha^2 p_{i-1} + \alpha^3 p_{i-2} + ... = \alpha p_i + (1-\alpha)P_{H_{i-1}}.$$

3.1.2 Hardware Assisted Performance History

Although tracking of the blocked times and the run times of each task is necessary, it provides only a rough indication of the performance demand of a task. For a more fine-grained performance observation, hardware assisted measurements can be used. In our exemplary implementation, we additionally use the number of cache misses. Upon a cache miss, the core usually stalls for some cycles until the data is fetched from main memory. Tasks that are memory intensive, thus, waste CPU time if cache misses happen frequently. Moreover, this wasted time is invisible to the OS and cannot be tracked solely by the means described above. The evaluation of different benchmarks, as conducted in [12], shows a correlation between the performance of the benchmarks and the number of CPU stalls due to cache misses, where high-performance tasks have fewer CPU stalls.

For a more fine-grained performance measurement, we let the performance monitoring unit (PMU) of the CPU count the number of cache misses when executing a task. The obtained value is multiplied with the average stall time specific to the actual hardware architecture. The resulting time is also considered as blocked time and subtracted from the run time of the task when calculating the performance value p_i of the epoch. This results in more realistic performance values for tasks that, at first glance, seem to use a core heavily.

3.2 Qualification Thresholds

The classification of tasks is based on the performance history value p_H. We use specific thresholds, i.e., the *up-migration threshold* and the *down-migration threshold*, for qualifying and disqualifying tasks as high-performance tasks, respectively. With this hysteresis, we prevent oscillations.

Moreover, these qualification thresholds are crucial to the effectiveness of our approach. Unfortunately, they cannot be determined completely automatically, because it depends on the use-case which tasks are considered important enough to be executed on a HPC. Therefore, we allow the user to specify a task that is exemplary for HPC execution and typical for the particular system use-case. Observing this task enables us to derive the required qualification thresholds and tuning the system to the given workload.

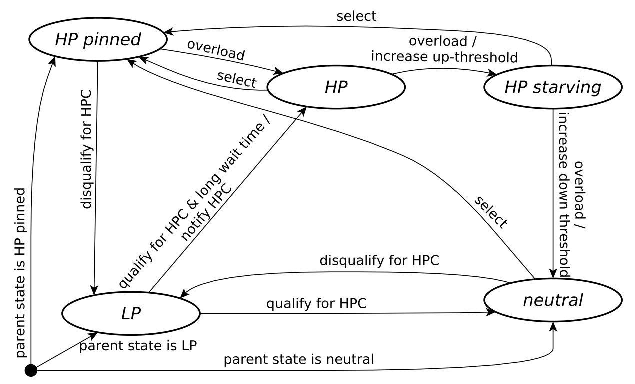


Figure 2: Task state diagram regarding HPC qualification.

3.3 Task States

For scheduling, it is not sufficient to simply divide the tasks into a low and a high performance group. Within each group, we have to take the cache affinity into account and allow a task to run on the same core whenever possible. Furthermore, assume a situation in which only high performance tasks are available. Assigning them solely to HPCs underutilizes the system and may lead to an avoidable HPC overload. To tackle these issues, we introduce a task model with five states that is described in the following.

3.3.1 Low Performance State

In the *low performance* (LP) state, a task does not surpass the qualification thresholds for HPCs. Therefore, a task in the LP state is never selected by any HPC for execution.

3.3.2 High Performance State

The *high performance* (HP) state is assigned to tasks that qualify for execution on HPCs. Furthermore, the task has not recently been run on an HPC and, thus, has no cache affinity for any of these cores.

3.3.3 HP Pinned State

In the *HP pinned* state, a task is qualified for HPC execution and it has also been recently run on an HPC so that it is cache affine to the particular core as some of its data may still be available in the core's cache. Hence, the particular core preferably selects such a pinned task. In case of an overload, i.e., there are more pinned tasks than the core can execute, the task falls back to the HP state.

3.3.4 HP Starving State

If the system experiences heavy load, it is possible that a task is qualified, but not selected by any HPC. In this case, it is assigned the *HP starving* state to avoid starvation. HP starving tasks are preferably selected by the HPCs.

3.3.5 Neutral State

If a task remains in the HP starving state for a longer time, it is moved to the *neutral* state. In this state, a task can be selected by both HPCs and LPCs. This avoids, among other things, situations in which all HPCs are overloaded while the LPCs are idle.

3.3.6 Initialization and Qualification

When a task is created, it needs an initial state. Its state is inherited from the parent task, e.g., if the parent task is in HP pinned state, the new task also starts in this state. If a task changes its behavior [19, 20] and, therefore, its suitability and qualification for HPC, a state transitions occurs in the task model. If the task gets qualified for HPCs, it is moved from LP state to neutral state or directly to HP state in case it has already waited for a certain amount of time. When the task gets disqualified for HPC execution, it is moved either from HP pinned state or neutral state to LP state. Please note that a task cannot transit from HP state or HP starving state to LP state as it has not been executed lately and, therefore, not changed its behavior. All described states and possible state transitions are summarized and illustrated in Fig. 2.

3.4 Adaptivity

The transition of a task from HP pinned state to HP state is the first indication of an overload situation. A task in HP starving state has even been further downgraded from HP state. This happens only in an actual overload situation as the task has been downgraded from the HP pinned state over HP state to HP starving state without getting the chance to run on an HPC. In this case, we increase the up-migration threshold which leads to fewer tasks that are newly classified as high-performance tasks. Furthermore, whenever a task in HP starving state becomes neutral and is selected by an LPC, we are still in an overload situation. To counter this, we also increase the down-migration threshold which leads to more tasks being disqualified for HPC and being moved to the LP state. This relaxes the overload situation immediately. Whenever an HPC is idle, i.e., there is no overload on the HPCs anymore, both thresholds are decreased again until they reach their initial values.

4. IMPLEMENTATION

This section highlights the details of our prototypical implementation. To realize the Multi-Core Load-Aware Scheduler (MCLAS), we used the BFS v0.446 as starting point. The BFS [11] implements an earliest deadline first (EDF) scheduling for choosing the next task for each core. Deadlines in BFS support scheduling decisions and should not be confused with deadlines in real-time systems used to guarantee the execution of a task in time. The BFS, for example, adjusts deadlines to model a task's stickiness to a particular core which is why BFS deadlines may differ for each core.

The vanilla BFS uses an unsorted list as run-queue. Thus, task insertion into the run-queue has a complexity of $\mathcal{O}(1)$ while task selection has a complexity of $\mathcal{O}(n)$ with n being the number of tasks in the system. Before we adapted BFS to heterogeneous multi-core scheduling, we first improved the task selection by replacing the unsorted list with a by-deadline-sorted set that both employs a red-black tree for the sorting and uses a list structured for fast iteration. This increases the insertion complexity for tasks a little to $\mathcal{O}(\log n)$, while the task selection complexity is drastically reduced to $\mathcal{O}(k)$ with k being the number of cores in the system. That is because there are at most k sticky tasks with deadlines that are interpreted differently, i.e., postponed when selecting the task for a core different to whom they stick. If these k tasks are placed at the beginning of the sorted set, all of them are inspected. Otherwise, selecting the first task in the sorted set is sufficient because it has the earliest deadline. As the number k of cores in the systems is constant and small compared to the number n of active tasks, the selection complexity is, thus, equivalent to $\mathcal{O}(1)$.

Thereafter, we implemented slightly different selection routines for LPCs and HPCs. The basic algorithm for task selection from the sorted run-queue is presented in Listing 1 in the appendix, where we also outline the differences for LPCs and HPCs. Additionally, we also implemented the measuring logic for determining whether or not a task is suited for execution on an HPC. As already mentioned earlier, the up-migration and the down-migration thresholds are crucial parameters for qualifying a task for HPC execution. To stay in line with the initial goal of BFS, which is to relieve the user from the need of tuning the system performance by various scheduler configuration parameters, we implemented an easy-to-use calibration logic. The user has to specify a task running on the system that is suited for HPC execution. The calibration logic observes and analyzes this task for 16 epochs in order to derive the up-migration and the down-migration threshold from the averaged measurements in a way that every task with a similar performance history is also regarded as HPC suited task.

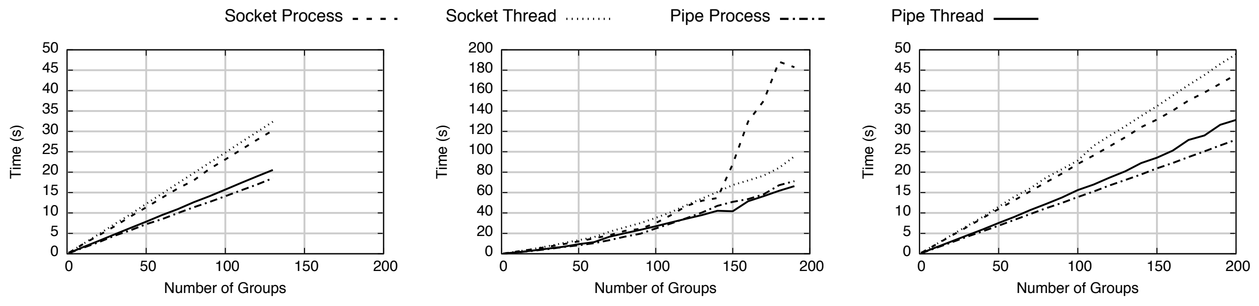
The weight α with which the current epoch contributes to the performance history P_H of a task is set to $\alpha = 0.5$. Furthermore, the residence time of a task in HP state is adapted relatively to the number of HPCs in the system, i.e., each HPC should have a chance to pick up the task for execution before the task is further downgraded.

5. EVALUATION

This section explains the evaluation of MCLAS. We start by describing the experimental setup. Next, we present a stress test that examines how the scheduler behaves under heavy load. Further benchmarks show the performance of MCLAS under normal load and evaluate context switching time and interrupt latency which is inspired by [14].

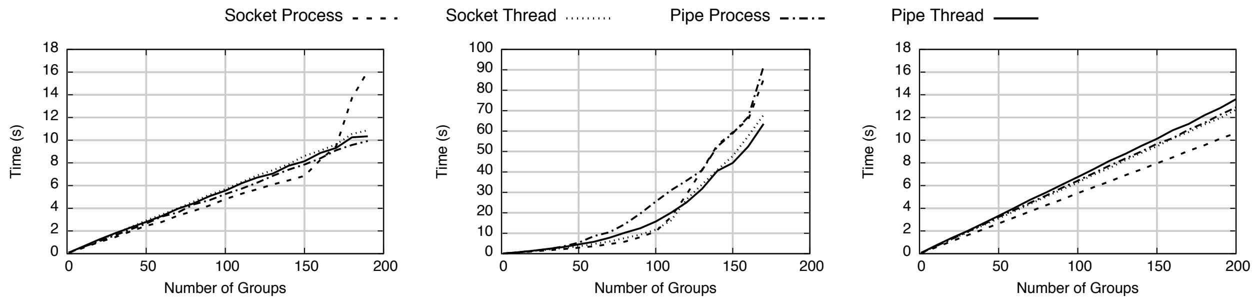
5.1 Experimental Setup

MCLAS is implemented based on the BFS v0.446 and executed within the Linux kernel v3.13. We use Gentoo as userland for the armv7a instruction set supporting a hardware floating point unit. As hardware platform, we use the Optimus Board from Merrii [4] that is equipped with an Allwinner A80 system on chip [1]. The Allwinner A80 implements an ARM big.LITTLE architecture with a heterogeneous configuration of four Cortex-A15 and four Cortex-A7 cores. The former has a default clock frequency of 1.8 GHz, while the latter has 1.2 GHz. The board has a main memory of 2 GiB. Each benchmark is conducted for three different schedulers: the default Completely Fair Scheduler (CFS) of Linux, the vanilla BFS, and our MCLAS scheduler.



(a) CFS (b) BFS (c) MCLAS

Figure 3: Scalability of CFS (a), BFS (b), and MCLAS (c) with 1 core. The systems with CFS and BFS both crashed before reaching 200 groups. Note the different scaling of the y-axis for BFS.



(a) CFS (b) BFS (c) MCLAS

Figure 4: Scalability of CFS (a), BFS (b), and MCLAS (c) with 8 core. The systems with CFS and BFS both crashed before reaching 200 groups. Note the different scaling of the y-axis for BFS.

5.2 Stress Test

The purpose of the stress test is to examine how the different schedulers manage the selection of processes for an increasing number of active processes. The stress test is intended to show how well a scheduler copes with a large amount of processes and how well these processes complete their work. We gradually increased the number of active processes in the system. First, the measurement was conducted using just one core so that only one run-queue is used in the decentralized CFS scheduler. The centralized schedulers have only one run-queue by definition. Thereafter, we repeated the measurement with all eight cores to evaluate the impact of parallelism on schedulers and workload. For conducting the stress test, we used the `hackbench` benchmark [2] which is a Linux kernel benchmark and stress test tool.

By default, `hackbench` creates a set of pairwise groups with each group consisting of 20 processes. The first group consists of sending processes while the second group contains the corresponding receivers. Each sending process sends messages to all receiving processes. The completion time of this communication is measured by `hackbench` and averaged. `Hackbench` can be configured to use threads or processes and, for the communication, it allows to choose pipes or sockets. Moreover, the number of process sets can be configured. We ran the `hackbench` benchmark with 1, 10, 20, 30,. . . , 200 sets of process groups, thus creating 40, 400, 800, 1200,. . . , 8000 processes, respectively. We evaluated both communication via pipes and via sockets.

The measurement for a single core was performed by setting the affinity of `hackbench` to a particular LPC. The results for this LPC core are presented in Fig. 3. For the multi-core measurement, no core affinity was set. Hence, the processes were allowed to be executed on any of the eight cores (including HPCs and LPCs) available in the system. The obtained results are shown in Fig. 4.

5.3 Context Switching Time

We evaluate the context switching time with the `lat_ctx` benchmark from the LMbench suite [15]. This benchmark connects processes through a ring of pipes. A token is passed from process to process forcing a context switch. The minimal communication time is determined by first evaluating the communication time of one process without context switching. This influences the results in case of multiple cores, where the communication time between cores varies depending on cache distance. Furthermore, the benchmark mimics real processes that perform calculations and consume cache memory by processing data.

For the evaluation, the benchmark was executed with rings of 2 to 32 processes each with a data size of 32 KiB. The affinity of the benchmark was set to one LPC in order to avoid using a wrong estimation of communication time, which is done by only one process in `lat_ctx`. Each measurement was repeated 30 times and the results were averaged. Figure 5 presents the results for the `lat_ctx` benchmark.

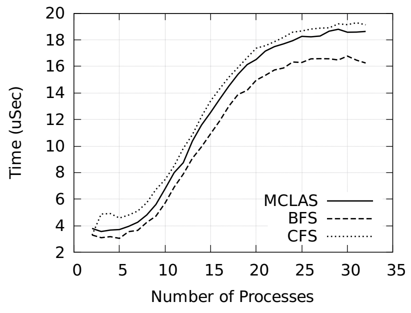


Figure 5: Context switching time with one core.

Latency μs	Min			Avg			Max		
	∀	LPC	HPC	∀	LPC	HPC	∀	LPC	HPC
idle system									
CFS	5	6	5	8	9	7	44	44	10
BFS	6	6	6	7	8	6	23	23	13
MCLAS	7	7	7	8	9	7	48	48	13
MCLAS-IRQ	6	7	6	8	8	7	25	25	14
low load									
CFS	5	6	5	8	9	7	61	61	39
BFS	6	7	6	8	10	7	254	254	21
MCLAS	7	7	7	9	11	7	177	177	19
MCLAS-IRQ	7	7	7	9	11	7	127	127	18
high load									
CFS	6	7	6	14	18	10	63	63	40
BFS	6	8	6	14	21	7	108	108	37
MCLAS	7	8	7	20	27	12	126	126	45
MCLAS-IRQ	7	9	7	17	25	8	66	66	42

Table 1: Measurements for the `cyclictest` benchmark.

5.4 Interrupt Latency

The interrupt latency is defined as the time that a system needs to respond to an interrupt. Thus, it measures the responsiveness of a system: the lower the interrupt latency gets, the more responsive the system becomes. For evaluating the interrupt latency, we used the `cyclictest` benchmark [6]. This benchmark measures the latency required to respond to an interrupt in the following way: It programs a periodical timer interrupt and measures the time between starting this periodical timer and the happening of the interrupt. The latency is the difference between the configured interrupt period and the measured time.

`Cyclictest` was set to create one thread per core with a hard affinity such that each thread measures the latency of the corresponding core. The measurement is performed in a loop of 10,000 rounds with an interval of 10 ms for the timer interrupt. We measured the latency in three different load situations: idle, low load, and high load. The load was constant during each measurement. The low load was playing a 480p movie with mplayer [5]. The high load was induced by transcoding a 720p movie into 360p using libav [3].

In a heterogeneous system, processors with different clock frequencies also have different processing times and, thus, latencies. To make the test representative, we measured the latency for each core separately during the test and grouped the results of similar cores together giving a latency value for LPCs and HPCs, respectively. Furthermore, we show the latency over all cores to provide a metric for each scheduler in general. Each value is presented by a column in Table 1.

In MCLAS, we followed a conservative approach when using HPCs for general tasks in the system, such that interrupts are handled only by LPCs by default unless explicitly requested. To inspect the impact of this configuration on the latency of interrupt handling, the same test was done on MCLAS with this feature disabled. The results are given in the rows of Table 1 that are labeled with "MCLAS-IRQ".

6. DISCUSSION

The stress test performed in 5.2 increases the number of active processes in the system in order to evaluate the scheduler's performance. More active processes lead to longer run-queues and, thus, to more overhead for inserting and/or selecting processes. Increasing the number of processors, however, aggravates the lock contention.

Setting the affinity of `hackbench` to one core causes all tasks to be inserted into one run-queue in case of CFS. BFS and MCLAS have only one run-queue by definition. The results depicted in Fig. 3a show that CFS has a linear scalability. Please note that the stress test failed for any number of sets higher than 130 with an out-of-memory error which killed the benchmark measurement. The test was repeated four times with very similar results for up to 130 sets of groups. BFS has a suboptimal scalability as shown in Fig. 3b. The benchmark also suffered from an out-of-memory problem after finishing 190 sets of groups. MCLAS shows a linear scalability as graphed in Fig. 3c and the stress test completed up to 200 sets of process groups without any error.

The benchmark runs with eight cores show a similar trend. CFS keeps the linear shape as shown in Fig. 4a while having a lower completion time when compared to the single core measurement. The benchmark was killed by the system after finishing 190 sets of groups due to an out-of-memory error. BFS as graphed in Fig. 4b performed worse requiring even higher completion times than in the single core case. Hence, lock contention has a significant impact on BFS. MCLAS manages to keep its linear scalability as presented in Fig. 4c while improving its completion times compared to the single core measurement. Thus, the benchmark with MCLAS is able to benefit from the increased parallelism provided by multiple cores. The benchmark was completed with 200 groups using MCLAS without any error.

The stress test proved the stability of the MCLAS implementation compared to both other schedulers. It also showed that, with MCLAS, tasks benefit from the increased parallelism of multiple cores while this turns into a problem for BFS. On a single core, the benchmark variant with processes that communicate via pipes achieved the lowest execution time in case of both CFS and MCLAS. This behavior is also observed for BFS until 100 sets of process groups. Hence, on a single core, communication with pipes is most efficient. When using all eight cores, however, MCLAS favors socket communication over pipes and processes over threads. This is different for CFS which also favors processes over threads, but pipes are more advantageous for threads and sockets for processes. Processes communicating via pipes suffered heavily under BFS. However, BFS is not designed to be used in stressed environments which is confirmed by the measurements. Nevertheless, the results for MCLAS also show that centralized scheduling can be used even in such situations. The decentralized CFS scheduler completed the stress test faster because of a better distribution of the workload among the processors, but due to its higher memory requirements the benchmark crashed before completion.

When designing a scheduler, one also needs to pay attention to the context switching time if user interaction is of importance. The context switching time contributed to the scheduler is consumed for selecting the next task that is executed next. On interactive systems, tasks are expected to be switched more frequently in order to timely respond to any user interaction. The higher the context switch time, the slower system since more time is wasted on context switching rather than on the actual progress of the tasks. BFS is designed to switch tasks often in order to be very responsive while featuring a low context switching time for a reasonable number of processes. This makes BFS suitable for interactive mobile devices. MCLAS extends BFS and also aims at keeping the context switching time as low as possible. However, MCLAS requires the tracking every task for HPC classification which is necessarily done at context switches causing additional expenses.

The measurements of the context switching time is shown in Fig. 5.3. BFS has the shortest context switching time. MCLAS has a longer context switching time than BFS due to the overhead of load tracking and classification of tasks. Nevertheless, MCLAS has still a lower context switching time than CFS. The context switching time increases with a growing number of processes. This is because the processes are using the cache memory and the scheduler data is, thus, evicted from the cache leading to a slower scheduler.

The discussion of the interrupt latency measurements is done separately for LPCs and HPCs. This also allows to consider the different handling of LPCs and HPCs by MCLAS. In fact, the obtained results prove that the interrupt latency is low for all three schedulers with values close on average. Our Implementation, MCLAS shows low latency competing with BFS. Handling interrupts by LPCs only has, on average, no significant impact on the latency except for high load situations. In these situations, the latency of handling the interrupts for LPCs was reduced by 7% and for HPC by 33%. This behavior is due to the distribution of interrupt handling among the cores such that idle HPCs participate in interrupt handling instead of waiting for the LPCs to handle the interrupts for them. The conservative approach to serving the interrupts mainly by LPCs leads to higher latency in high load situation with the goal of decreasing the usage of HPCs to conserve energy. This behavior should be further investigated with respect to the noticeable latency by user in relation to the energy saved which we left for future work.

7. RELATED WORK

The authors of [8] identify three challenges for heterogeneous multiprocessing from the OS perspective:

- the OS has to keep track of the dynamic state of every CPU and its computational resources,
- the OS has to discover the specific resource and performance demands of each task, and
- the OS has to assign each task to the proper core in an efficient way.

Our approach tackles the last two challenges by classifying tasks with respect to their performance demands and by assigning them to an HPC or LPC, respectively, according a corresponding task state model.

The authors of [18] present a scheduler for heterogeneous same instruction set architectures. The presented scheduler relies on thread signatures that are profiled offline in order to keep scheduling decisions simple and efficient. In contrast, we use an on-line approach that is yet simple without prior knowledge about the tasks.

In [7], the authors use a dynamic approach that observes the behavior of each task for a proper assignment by using an instructions per cycle (IPC) ratio. We also rely on observed behavior, but use measured run times adjusted by the number of cache misses for each task.

The approach of [10] monitors code segments on synchronization points and whenever scheduling time quantum expires in order to identify critical sections so that these sections are executed on HPCs. Their metric depends on time and instruction counters. We do the observations on thread level rather than on code segments. Thus, we also detect lagging threads, that are identified by their approach, with a simpler estimation which counts the number of cache misses.

The authors of [12] solely use the hardware performance monitoring unit (PMU) to estimate the suitability of a task for HPCs and LPCs. They count CPU stalls, similar to our approach, by querying the PMU on context switches and use the results to calculate a task-specific bias which influences load balancing decisions. They use thresholds for the stalls themselves to decide on the suitability for HPC or LPC and leverage a decentralized scheduler.

The Linaro HMP approach [13] provides an implementation of a complete scheduler for heterogeneous multi-core architectures that follows a decentralized approach. For this work, we build on experience with the Linaro scheduler.

In [17], a decentralized scheduler is presented that aims for fairness on heterogeneous cores. Our approach relies on a centralized scheduler and prefers to use the HPCs conservatively instead of considering fairness.

8. CONCLUSIONS

In this paper, we tackled the problem of heterogeneous multi-core scheduling for systems with two types of cores that differ in performance and power efficiency. With MCLAS, we implemented a centralized scheduler that extends the BFS scheduler to handle such systems. Therefore, it incorporates a metric to classify tasks whether or not they are suited to be executed on HPCs. Furthermore, we presented a task state model that is based on this metric and allows for efficient scheduling of tasks while avoiding task starvation and system underutilization. With performance benchmarks, we showed that MCLAS does not fall behind CFS and a vanilla BFS that both do not support heterogeneous multi-core scheduling. Therefore, the presented scheduling approach is well suited for real-world challenges.

9. REFERENCES

[1] A80 Processor. http://www.allwinnertech.com/en/clq/processora/A80.html. Last visited: 20.09.2015.

[2] The hackbench scheduler benchmark and stress test. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c. Last visited: 20.09.2015.

[3] libav, open source audio and video processing tools. https://libav.org/. Last visited: 20.09.2015.

[4] Merri evaluation board – A80 OptimusBoard. http://www.merrii.com/en/pla_d.asp?id=173. Last visited: 20.09.2015.

[5] Mplayer, a movie player. http://www.mplayerhq.hu. Last visited: 20.09.2015.

[6] rt test utils. http://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git/. Last visited: 07.12.2015.

[7] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers (CF '06)*, pages 29–40, New York, NY, USA, 2006. ACM.

[8] Fred A. Bower, Daniel J. Sorin, and Landon P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28(3):17–25, May 2008.

[9] Peter Greenhalgh. big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. White paper, ARM Limited, September 2011.

[10] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Utility-based acceleration of multithreaded applications on asymmetric cmps. *SIGARCH Computer Architecture News*, 41(3):154–165, June 2013.

[11] Con Kolivas. BFS cpu scheduler v0.304 stable release. http://lwn.net/Articles/357451/. Last visited: 7.12.2015.

[12] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, pages 125–138, New York, NY, USA, 2010. ACM.

[13] Linaro. ARM big.LITTLE MP Kernel tree. https://git.linaro.org/arm/big.LITTLE/mp.git. Last visited: 07.12.2015.

[14] Martin Lowinski. Analysis and enhancements of scheduling-strategies for multicore systems. Master's thesis, Fakultät für Informatik, Technische Universität München, Germany, March 2013.

[15] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC '96)*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[16] NVIDIA. Nvidia tegra multi-processor architecture. White paper, NVIDIA Corporation, February 2010.

[17] Juan Carlos Saez, Adrian Pousa, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias. ACFS: A Completely Fair Scheduler for Asymmetric Single-isa Multicore Systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*, pages 2027–2032, New York, NY, USA, 2015. ACM.

[18] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Operating Systems Review*, 43(2):66–75, April 2009.

[19] Timothy Sherwood and Brad Calder. Time varying behavior of programs. Technical report, Department of Computer Science and Engineering, University of California, San Diego, CA, USA, 1999.

[20] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 176–188, New York, NY, USA, 1991. ACM.

APPENDIX

MCLAS extends the selection algorithm of BFS in order to handle heterogeneity based on the following considerations. For HPCs there are two considerations:

1. No soft affinity is respected for tasks in sate *HP*, selection is only done by deadline.
2. No tasks with in *LP* state are considered for selection.

For LPCs, the situation is different and the selection scheme is more complex to fulfill the following goals:

- In overload situations, LPCs are allowed to take over after giving HPCs a reasonable chance.
- However, no tasks are allowed to starve.
- LPCs may spend more time for selecting a task in order to do balance load in case of overloaded HPCs.

Listing 1: The selection algorithm of MCLAS.

```
task p, ps ← NULL, edp ← NULL
deadline ed ← ∞
foreach p in sorted run−queue do
    if p is sticky to other core then
        if ps ≠ NULL ∧ deadline(p) < ed then
            'last sticky is absolute earliest
            edp ← ps
            break
        elseIf ps = NULL ∨ vDL(p) < ed then
            'p is earlier sticky
            edp ← p
            ps ← p
            ed ← vDL(p)
        end if
    else
        'search ends
        if ps = NULL ∨ vDL(p) < ed then
            edp ← p
        end if
        break
    end if
done
return edp
```

deadline(p) denotes the deadline of task p, while vDL(p) is the deadline of task p postponed relatively to the cache distance between the current core and the core to which p is sticky.