Constructing a Uniform Heterogeneous Multiprocessor by CPU Frequency Scaling and Exploring EDF Scheduling on It *

Kecheng Yang
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

We construct a heterogeneous platform on a homogeneous one by leveraging the CPU frequency scaling tools. We implement a full-migration GEDF scheduler as a plug-in to a middleware that runs on the top of `preempt_rt` *Linux kernels. Furthermore, we improve the scheduler by reducing migrations. Both a detailed comparison of a particular example and a schedulability evaluation of randomly generated task sets are provided.*

1 Introduction

Since the advent of multicore chips, the real-time system community has made fruitful results for scheduling problems on multiprocessors. Yet a major body of such results only pertains to homogeneous multiprocessors where every processor is identical in terms of both *functionality* and *speed*. Recently, several emerging techniques may lead the hardware platform to be a heterogeneous multiprocessor where the processors may have different speeds. For example, the big.LITTLE [1] technology proposed by ARM is supposed to integrate relatively slower, low-power processors with faster, high-power ones to balance performance and energy efficiency. As an implementation of the big.LITTLE technology, the Samsung mobile SoC Exynos 5422 [5] consists of four slower ARM Cortex-A7 cores and four faster ARM Cortex-A15 cores. Furthermore, some state-of-art research is exploring scheduling problems on unreliable processors where the processing speed may vary [7], and its multiprocessor extension [10] may directly result in the heterogeneity we mentioned above.

However, the analysis for scheduling problems on heterogeneous multiprocessors are much more complicated than that on homogeneous ones. Interestingly, the analytical gap between heterogeneous and homogeneous multiprocessors is very similar to that between multiprocessors and uniprocessors. The well-known key difference between multiprocessors and uniprocessors is *idleness*, *i.e.*, on uniprocessors, idleness means the whole system is idle since exact one processor exists in the system; whereas on multiprocessors, idleness could mean exact one processor is idle, or exact two processors are idle, *etc.*, which means there are multiple interpretations for idleness. In contrast, the key difference between heterogeneous and homogeneous multiprocessors is *executing*. On homogeneous multiprocessors, a task can only be either executing or not executing. Since each processor is identical, executing on any processor is the same. However, on heterogeneous multiprocessors, since each processor may have a different speed, executing is not sufficient to precisely describe the state of a task, and therefore both the scheduler and the analysis need to consider *which* processor the task is executing on. In this project, we construct a heterogeneous platform by CPU frequency scaling techniques, implement a earliest-deadline-first (EDF) scheduler on it, and explore related problems about both implementation and analysis.

Related Work. EDF scheduling research on heterogeneous platforms was initiated by Funk *et al.*. In [9], they established a feasibility condition for scheduling periodic tasks upon uniform heterogeneous multiprocessors by leveraging the Level Algorithm [11]. Also, they proposed three EDF-based schedulers (concluded in Funk's dissertation [8]) for uniform heterogeneous multiprocessors with different migration constraints: f-EDF (full migration), p-EDF (partitioned, no migration), and r-EDF (restricted migration). More recently, several more complicated EDF-based scheduling algorithms were proposed [12, 16, 17, 18].

*A course project report for Prof. Don Smith's class, COMP 790-042 Operating System Implementation.

However, all of the cited prior work above are rather purely theoretical work. That is, as all of the experiments are simulation-based evaluation, none of them actually set up a uniform heterogeneous platform and implemented the schedulers in a real system.

Contributions. We construct a uniform *heterogeneous* multiprocessor on a *homogeneous* one by leveraging the *CPU frequency scaling* tools. As such tools are usually not designed for heterogeneity but for energy saving by *dynamic* scaling, we study the way to set user-specified *static* CPU frequency via those tools. Also, the hardware constraints (*e.g.*, some cores may share a clock source) could prevent the constructed platform from being *heterogeneous*, even if the CPUs are frequency-scalable. We address this issue as well. By setting up the heterogeneous platform and running the middleware on it, we demonstrate the middleware, which was designed and developed for identical multiprocessors, is applicable under at least some heterogeneous circumstances.

We implement a *full-migration* global earliest-deadline-first (GEDF) scheduler as a plug-in to the middleware. Furthermore, we improve the scheduling algorithm by avoiding unnecessary migrations, which is proved advantageous with respect to run-time overheads and preserves all the theoretical properties of the *full-migration* GEDF scheduler. We evaluate both algorithms by their *measured* schedulabilities.

Organization. In the rest of the paper, we provide background (Sec. 2), explore the construction of the heterogeneous platform (Sec. 3), implement the schedulers (Sec. 4), conduct schedulability experiments (Sec. 5), and conclude (Sec. 6).

2 Background

We consider the scheduling of n sequential periodic tasks on m processors, where $n \geq m$. We specify a task τ_i by (ϕ_i, C_i, D_i, T_i), where ϕ_i is its *phase*, C_i is its *worst-case execution requirement* which is defined as its *worst-case execution time* on a *unit-speed* processor, D_i is its *relative deadline*, and T_i is its *period*. Also, we denote its *utilization* as

$$u_i = \frac{C_i}{T_i}.$$

When every task have a zero phase and a relative deadline equal to its period, the system is called a *synchronous implicit-deadline* periodic task system, where a task can be simply denoted as $\tau_i = (C_i, T_i)$. Moreover, on a heterogeneous platform, $u_i \leq 1$ does not necessarily hold. Needed restrictions on utilizations are given later in Sec. 2.1.

A *job* is an invocation of a task. If a job that has a absolute deadline at time t_d and completes at time t_c, then its deadline is *met* if and only if $t_c \leq t_d$; otherwise, this deadline is *missed*. A job is *pending* if and only if it is released but not complete. A job is *ready* if and only if it is pending and all its predecessors (the previous invocations of the same task) have been complete. If, under a *certain* scheduling algorithm $\mathcal{A}$, no deadline has ever been missed for a system, then this system is *schedulable* under algorithm $\mathcal{A}$; if, under *some* algorithm, no deadline has ever been missed for a system, then this system is *feasible*.

A taxonomy of multiprocessors. The following taxonomy [8, 15] classifies multiprocessor platforms according to assumptions about processor speeds—the *speed* of a processor refers to the amount of work completed in one time unit when a job is executed on that processor.

- **Identical multiprocessors.** Every job is executed on any processor at the same speed, which is usually normalized to be 1.0 for simplicity.
- **Uniform heterogeneous multiprocessors.** Different processors may have different speeds, but on a given processor, every job is executed at the same speed. The speed of processor p is denoted s_p.
- **Unrelated heterogeneous multiprocessors.** The execution speed of a job depends on both the processor on which it is executed and the task to which it belongs, *i.e.*, a given processor may execute jobs of different tasks at different speeds. The execution speed of task τ_i on processor p is denoted $s_{p,i}$.

2.1 Uniform Heterogeneous Multiprocessors

In the rest of this paper, we consider a uniform heterogeneous multiprocessor system π, where processor i is represented by its speed s_i ($1 \leq i \leq m$, $s_i \in \mathbb{R}$). Also, we index the processors in non-increasing-speed order, *i.e.*, $\pi = \{s_1, s_2, \cdots, s_m\}$, where $s_i \geq s_{i+1}$ for $i \in \{1, 2, \cdots, m-1\}$. We consider scheduling a

periodic task set τ on π. We index the tasks in non-increasing-utilization order, *i.e.*, $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, where $u_i \geq u_{i+1}$ for $i \in \{1, 2, \cdots, n-1\}$.

Let $U_k = \sum_{i=1}^{k} u_i$, $S_k = \sum_{i=1}^{k} s_i$. Also, denote the total system utilization as $U_\tau = U_n$ and the total platform capacity as $S_\pi = S_m$. By leveraging the Level Algorithm [11], Funk *et al.* [9] showed that an *implicit-deadline periodic* task system τ is feasible on a uniform heterogeneous multiprocessor system π if and only if the following conditions hold.

$$\begin{cases} U_\tau \leq S_\pi, \\ U_k \leq S_k, \quad \text{for} \quad k = 1, 2, \cdots, m-1. \end{cases}$$

2.2 Middleware

This project is powered by a real-time scheduling middleware developed by Mac Mollison [13, 14]. This middleware has been proved to work on the top of `preempt_rt` kernels in Linux Operating Systems. However, the middleware was designed for identical multiprocessors, and none of prior experiments on this middleware has been conducted on a heterogeneous platform. Thus, this project also proves that this middleware is even more *robust* than it was designed for.

The timing references in the middleware rely on the function `rdtsc()`, which reads the value of the Time Stamp Counter (TSC). Therefore, a significant potential problem on heterogeneous multiprocessors is whether the different processor speeds, or CPU frequencies, would interfere the TSC and cause synchronization or scaling problems. Fortunately, many Intel processors support a constant (or even invariant) TSC technology[6], which ensures that the TSC counts at the rate as the standard CPU frequency no matter what the actual CPU frequency is (and TSC counts across processors are synchronized, if invariant TSC).

Moreover, the middleware package provides several handy tools. In particular, the tool `draw` can draw out a visualized schedule from the middleware running log file `out.trace`, which is very useful for verification and illustration of scheduling algorithms. More details about the settings and architecture of the middleware can be found in a manual and two flow control graphs in `/doc` in the middleware package path.

3 Platform Construction

Although the companies, as mentioned in Sec. 1, do design and produce uniform heterogeneous multiprocessors nowadays, we do not have one in our group. Thus, the first step of this project is to construct a uniform heterogeneous multiprocessors on one of the homogeneous machines that we have. As CPU frequency scaling is the most promising way to do that and most of the machines in our group are equipped with Intel CPUs, we focus on Enhanced Intel SpeedStep® Technology (EIST) [3], which allows the system to dynamically adjust processor voltage and core frequency. To determine if a Intel CPU supports EIST, we can check it out in the Intel ARK website. Then we need to boot into the BIOS and enable EIST function, or it is enabled by default if no such item exist in the BIOS. Finally, in Linux, we can confirm that EIST is enable by typing the command `cat /proc/cpuinfo` and checking if `est` exists in the line of `flags`.

3.1 Kernel Configuration

In order to construct a heterogeneous platform, we have to enable the following options in the Linux kernel configuration, which are critical for enabling CPU frequency scaling in Linux. Tables 1 and 2 provide the details about governors and drivers[4].

```
Power management and ACPI options  --->
     [*] ACPI (Advanced Configuration and
      Power Interface) Support  --->
         <*>   Processor
     CPU Frequency scaling  --->
         [*] CPU Frequency scaling
             Default CPUFreq governor (ondemand)  --->
                 Select a default governor; see Table 1.
                 Default is:
                 ondemand
             x86 CPU frequency scaling drivers  --->
                 Select a driver, see Table 2.
```

In this project, based on the machines we work on, the `acpi-cpufreq` driver[1] is preferable. Also, we should choose the *'userspace' governor* to manually scale CPU frequencies, since our goal is to construct a uniform heterogeneous multiprocessor.

3.2 Cpufrequtils

When obtained hardware supports and proper Linux kernel configuration, to make a easier access to the Linux kernel cpufreq subsystem, we also need a userspace tool, which is the `cpufrequtils` package. To install this package, we need the command `sudo apt-get install cpufrequtils`. Once the package is installed, we should have the two commands, `cpufreq-info` and `cpufreq-set`,

[1] Other drivers, *e.g.*, `intel_pstate`, may cause problems; we will discuss them in Sec. 3.4.

Option	Module	Supported Processors	Note
'performance' governor	cpufreq_performance	Sets the frequency statically to the highest available CPU frequency.	
'powersave' governor	cpufreq_powersave	Sets the frequency statically to the lowest available CPU frequency.	cannot be set as default governor
'userspace' governor	cpufreq_userspace	To set the CPU frequency manually or when a userspace program shall be able to set the CPU dynamically.	
'ondemand' governor	cpufreq_ondemand	Does a periodic polling and changes frequency based on the CPU utilization.	recommend
'conservative'governor	cpufreq_conservative	Similar to ondemand. The frequency is gracefully increased and decreased rather than jumping to 100% when speed is required.	

Table 1: Options for `Default CPUFreq governor`.

Option	Module	Supported Processors	Note
Intel P state control	intel_pstate		provided by Intel, no manually scaling governor such as 'userspace'
Processor Clocking Control interface driver	pcc-cpufreq		
ACPI Processor P-States driver	acpi-cpufreq	Intel Core, Intel Core 2, Intel Atom, Intel Pentium M	
AMD Opteron/Athlon64 PowerNow!	powernow-k8	AMD Opteron, AMD Athlon 64, AMD Turion 64	
Intel Enhanced SpeedStep (deprecated)	speedstep_centrino	Intel Pentium M (Centrino), Intel Xeon	deprecated, use ACPI Processor P-States driver instead
Intel Pentium 4 clock modulation	p4-clockmod	Intel Pentium 4, Intel XEON	causes severe slowdowns and noticeable latencies

Table 2: Options for `x86 CPU frequency scaling drivers`.

to determine (the former) current CPUfreq settings and to modify (the latter) them.

The command `cpufreq-info` returns CPUfreq information for each core, such as

```
driver,
maximum transition latency,
hardware limits,
available frequency steps,
available cpufreq governors,
current policy,
current CPU frequency,
```

in which we should confirm that `acpi-freq` is the current applied driver and the current governor (shown in `current policy`) is 'userspace', and `current CPU frequency` provides one indicator for the current processor speed[2].

The command `cpufreq-set` is used to select CPUfreq governors, and to set CPU frequencies if the governor is 'userspace' or CPU frequency ranges if some other governor is selected. The syntax of the `cpufreq-set` command is `cpufreq-set [options]`. Table. 3 shows the details about `options`[2]. Note that omitting the -c or –cpu argument is equivalent to setting it to zero and the `-f` option cannot be combined with any other option except the `-c`. Thus, the specifically commands we use are in the following formats.

```
cpufreq-set -c 0 -g userspace
cpufreq-set -c 0 -f 2.5GHz
```

[2]This indicator may be imprecise, or even incorrect; we will discuss this in Sec. 3.4.

[options]	Description
-c <CPU>	number of CPU where cpufreq settings shall be modified
-d <FREQ>	new minimum CPU frequency the governor may select
-u <FREQ>	new maximum CPU frequency the governor may select
-g <GOV>	new cpufreq governor
-f <FREQ>	specific frequency to be set; requires userspace governor to be available and loaded
-h	prints out the help screen

Table 3: Details for [options] of cpufreq-set.

3.3 Speed Test

As mentioned in Footnote 2, the current CPU frequency gathered by tools may be incorrect. Therefore, to make sure the platform is really *heterogeneous* as set, we also write a little speed-test program to verify the processor speeds. The program is just simply working loops, and we use time() (relative rough) and rdtsc() (more precise) to get time stamps before and after the loops and to calculate the running time for the working loops. Then we use the Linux bash command taskset -c <CPU> to enforce the whole task to run on a specified processor without migration. By compare the output running times, we can verify if the processors really run at different speeds and the relative speeds between processors. The speed test codes are speedtest.c and start. Furthermore, we can use the command htop to monitor the real-time CPU usages, to see the rough task completion time on each processor, and hence to determine if the platform is actually heterogeneous.

3.4 Machine-Specific Details

In this sub-section, we describe our efforts to construct the platform on several existing machines in chronological order. All of the first five attempts failed for various reasons, but the last one has succeeded.

pound@cs.unc.edu. Pound is a four-core machine. It has a Intel® Core™ i7-920 Processor, which consists of four 2.66GHz cores that support EIST technology. On Pound, we have to boot into BIOS and enable the EIST option to support CPU frequency scaling. Then the acpi-cpufreq driver is automatically loaded on each processor, provided related Linux kernel configuration options are enabled. Moreover, cpufrequtils works perfectly on Pound, and it shows that each core has ten speed steps can be selected within the range from 1.60GHz to 2.66GHz. After governors and frequencies are set, both cpufreq-info and cat /proc/cpuinfo suggest the four cores are running on different speeds. However, when the speed test is applied, it shows that the four cores are actually running at the same speed. Moreover, by monitoring the htop, it appears that the tasks on the four cores complete simultaneously, which also implies that, in fact, the platform is not heterogeneous. It is probably because the four cores are on a same chip and therefore may share a clock source. Thus, the attempt on Pound has failed due to hardware constraints.

ludwig@cs.unc.edu. Ludwig is a 24-core machine. It has four Intel® Xeon® Processor L7455, each of which consists of six 2.13GHz cores. Unfortunately, both Intel official website and the flags in /proc/cpuinfo suggest that the processors do not support EIST technology. Thus, Ludwig is not suitable for this project.

koruna@cs.unc.edu (failure). Koruna is a eight-core machine. It has two Intel® Xeon® Processor E5420, each of which consists of four 2.5GHz cores. In BIOS, there is no option related to EIST technology, but in /proc/cpuinfo we can clearly see est in the flags line, which means EIST is implicitly enabled on these processors. However, when we tried to boot into a kernel where the ACPI Support is enabled, Koruna was always stuck on the kernel booting. Even worse, after several attempts, we could not see the grub menu while rebooting the machine, and therefore could not boot into a previously working kernel to continue trying. Koruna has a Dell Remote Access Control (DRAC) server and a physical monitor in the machine room, but both of them were stuck without showing the grub menu, no matter we reboot the machine by the powercycle command in DRAC or by pushing the physical reboot button in the machine room.

bonham@cs.unc.edu. Bonham is a 12-core machine. It has two Intel® Xeon® Processor X5650, each of which consists of six 2.66GHz cores. Each individual core on Bonham is very similar to that on Pound, so the CPU frequency scaling should also work well on Bonham. Furthermore, Bonham has two CPU chips, or groups, so it should be able to perform heterogeneously by CPU frequency scaling. However, Glenn Elliott has been conducting extensive experiments on Bonham and he is graduating. Thus, I would not intro-

duce any interference to his work and therefore have given up working on Bonham.

zildjian@cs.unc.edu. Zildjian is a 36-core machine, which is a brand new machine purchased in late 2014. It has two Intel® Xeon® Processor E5-2699 v3, each of which consist of 18 2.30GHz cores. On these processors, EIST is implicitly enabled, and therefore no BIOS option needs to be change. Also, `est` appearing in the `flags` line in `/proc/cpuinfo` confirms this. The problem on Zildjian is that we could only boot it into a Linux kernel of version 3.13 or higher; otherwise, the kernel booting would be stuck on the message "`i8042: No controller found`". Meanwhile, in a Linux kernel of version 3.9 or higher, `intel_pstate` rather than `acpi-cpufreq` is the default CPUfreq driver, which does not allow us to manually scale the CPU frequencies. Some Internet posters suggest that adding "`intel_pstate=disable`" to the kernel booting command line or just not selecting the `intel_pstate` driver in the kernel configuration would solve this problem and give the `acpi-cpufreq` driver back, but it does not work on Zildjian. Once disabled the `intel_pstate` driver, we did not get the `acpi-cpufreq` driver back but just saw the message "`no or unknown cpufreq driver is active on this CPU`" on every core. Thus, the attempt on Zildjian has also failed.

koruna@cs.unc.edu (success). After a departmental power outage in summer 2014, we found that, although either rebooting via DRAC or the physical button does not recover the stuck machine, a power cut-off (power outage or physically pulling the power cable out) does restore the grub menu on DRAC. Then we could continue attempts on Koruna again. Finally, we figured it out that, to make the kernel booting normally while enabling the `ACPI Support`, we should disable *all* sub-options that may be enabled by default in `ACPI support` other than `Processor`, such as, `AC Adapter`, `Battery`, and `Fan`. Then, a 3.0 Linux kernel with `preempt_rt` patches works perfectly on Koruna. The driver is `acpi-cpufreq` and `cpufrequtils` works well. Furthermore, both our speed test and the monitoring `htop` method confirm that the processors are indeed at different speeds, *i.e.*, the platform is actually heterogeneous. One flaw might be that the processors on Koruna only support two frequency steps, 2.5GHz and 2.0GHz, so the heterogeneity may not be dramatic. However, we believe that is adequate for this project. Moreover, via the tool `./workdir/tools/coreinfo` in the middleware, we have found that physical CPUs 0, 2, 4, 6 are in the same core package and therefore have to be at the same frequency while physical CPUs 1, 3, 5, 7 are in the other one. We can adjust the core order in the middleware configuration file `./workdir/conf/system.conf` to be "`cpu_for_worker = 0,2,4,6,1,3,5,7`". Then, we have the first four cores at the same speed and the other four at the other speed.

In conclusion, we have a eight-core uniform heterogeneous multiprocessor on Koruna, where four cores are running at the frequency of 2.5GHz and the other four are at 2.0GHz. The Linux kernel is a `preempt_rt` kernel of version `3.0.17-rt33`. Henceforth, we can model Koruna as a uniform platform π, where $s_1 = s_2 = s_3 = s_4 = 1.25$ and $s_5 = s_6 = s_7 = s_8 = 1.0$.

4 Algorithms and Implementation

In this project, we focus on the *full-migration* global earliest-deadline-first (GEDF) scheduling on the uniform heterogeneous multiprocessor we constructed.

According to [8], the *full-migration* GEDF scheduler give higher priorities to jobs with earlier deadlines, and deadlines not only determine *which* jobs execute, but also *where* they execute, *i.e.*, the earlier the deadline the job has, the faster the processor it is scheduled on. More formally, the scheduling rules are as follows.

- At any time instant, if there are at most m tasks with a ready job, then all of them are scheduled; if there are more than m tasks with a ready job, then the m ones with the earliest deadlines are scheduled. Deadline ties are broken arbitrarily.
- For any two tasks τ_i with current absolute deadline d_i scheduled on processor s_p and τ_k with current absolute deadline d_k scheduled on processor s_q, where $p < q$, we have $d_i \leq d_k$ (note that, given how we order processors in Sec. 2.1, $s_p \geq s_q$).

The following example illustrates full-migration GEDF scheduling.

Ex. 1. Recall that in Sec. 2, we specify a task by $\tau_i = (\phi_i, C_i, D_i, T_i)$. Considering the full-migration GEDF

scheduling of the task set (parameters are in milliseconds)

$$\begin{aligned}\tau = \{\ &\tau_1 = (0, 1, 3, 10),\\ &\tau_2 = (0, 2, 4, 10),\\ &\tau_3 = (0, 3, 5, 10),\\ &\tau_4 = (0, 4, 6, 10),\\ &\tau_5 = (0, 5, 7, 10),\\ &\tau_6 = (0, 6, 8, 10),\\ &\tau_7 = (0, 7, 9, 10),\\ &\tau_8 = (0, 8, 10, 10)\ \}\end{aligned}$$

on the uniform platform $\pi = \{s_1 = s_2 = s_3 = s_4 = 1.25, s_5 = s_6 = s_7 = s_8 = 1.0\ \}$, Fig. 1(a) provides the theoretical schedule produced by the full-migration GEDF scheduler. We only show the first period, since after that the schedule repeats.

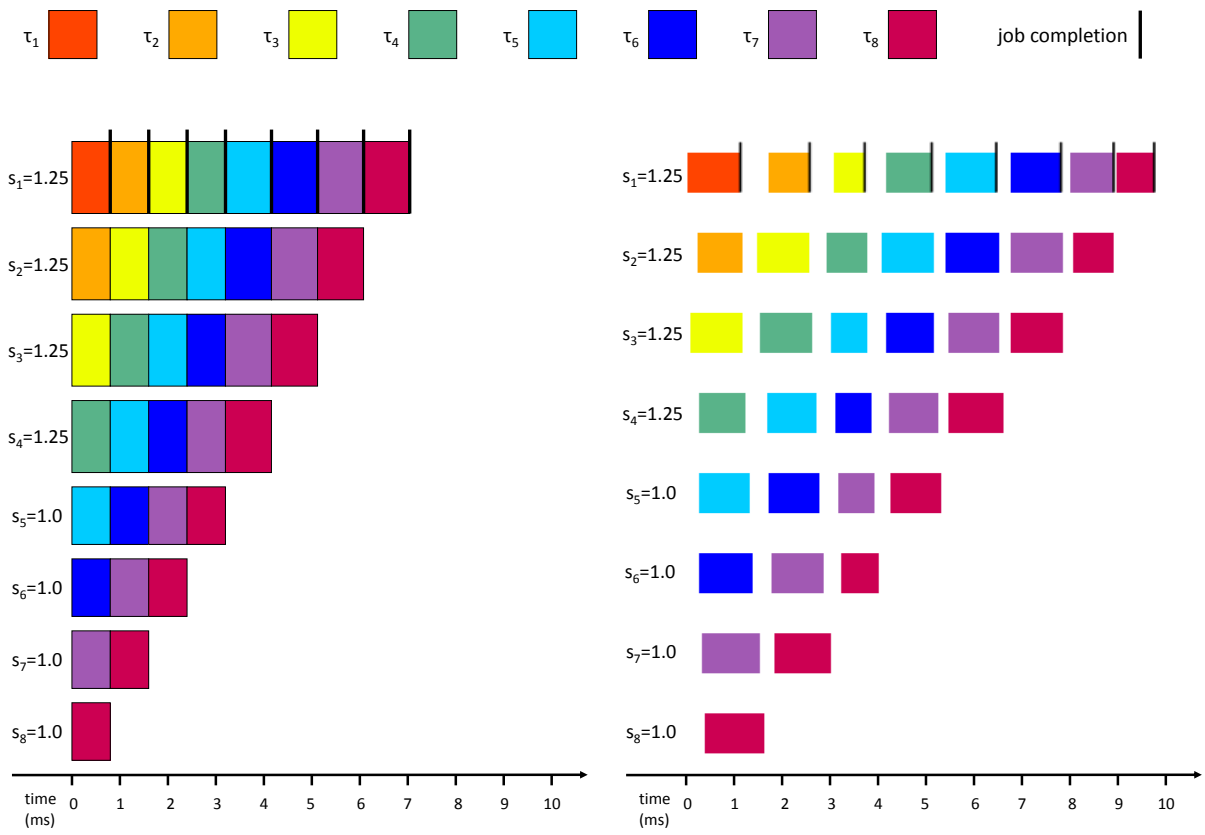
Implementation. We implement a full-migration GEDF scheduler as a plug-in to the middleware. We have two task queues to maintain, two kinds of signals to handle, and one core request to deal with. Namely, the two queues are a release queue and a ready queue, each of which is implemented by a binomial heap. The release queue contains all tasks that their current job has not released yet, where node values are next release times; the ready queue contains all tasks that their current job has already released, including currently executing ones, where node values are current priorities (absolute deadlines). In the main algorithm function `schedule()`, we get the top m tasks in the ready queue, and make the decision that let the m tasks execute on the m processors in order. Moreover, the processor that triggered `schedule()` can do the task switch locally and send interrupt signals to all other processors that need a task switch. When a processor get a interrupt signal, it switch its task according to the decision from the most recently `schedule()` call. Also, when the timer is fired, a timer signal will be sent. When a processor get a timer signal, it moves all ready tasks in the release queue to the ready queue, set a new timer, and call `schedule()`. The only core request in this implementation is "`R_COMPLETION`", which is triggered when a job completes. When a core request is received, the processor updates the parameters of this task structure. Furthermore, we remove the node that corresponds to this task from the task queues, and then enqueue the updated one to either the release queue or the ready queue again by `enqueue()`, depending on the updated release time.

A significant problem of this algorithm is the task switch to a current executing task. This is a new issue in heterogeneous platform. On identical multiprocessors, we have no point to migrate a currently executing task because every processor is the same, *i.e.*, any migration is associated to a preemption, a task will not migrate unless it was previously preempted, and is not currently executing. However, on a heterogeneous platform, we may migrate a currently executing task from a higher-indexed (slower) processor to a lower-indexed (faster) one, and such migration could be chained, *i.e.*, several tasks are shifting among processors.

Switching to currently executing tasks are problematic because the task switch can only switch to a task with a previously saved context, while a task only saves its context when it is preempted. To deal with this problem, our approach is to switch those currently executing tasks that are going to migrate to idle first, and then switch them to their proper processors. In our implementation, there are two arrays that describe the `real` task on each processor and the `wanted` task by each processor. The task switches are triggered by a difference between these two array, and the `wanted` array is updated when `schedule()` is returned, representing the scheduling decision. Therefore, whenever a task switch happens, we check the `real` array to see if the `wanted` task is executing somewhere else. If not, we switch to the `wanted`; otherwise, we switch to idle. Then we check if the `old_task` (the one switched off) is `wanted` by some other processor. If so, we send an interrupt signal to that processor, telling it that "the task you want is now ready to switch". The code of this plug-in scheduler is `fgedf.c`.

Fig. 1(b) is the actual schedule of Ex. 1 under the full-migration GEDF scheduler in the system. As seen, each time a lot of tasks are shifting, the scheduling overheads are dramatic. That is because of signal transmission latency and signal synchronization mechanism. They raise the overheads significantly when a lot of signals are simultaneously transmitting among cores.

Improvement. Due to the significant overheads, we are going to improve the scheduler somewhat. Notice that in this experiment platform, we actually have only



(a) theoretical full-migration GEDF schedule

(b) actual full-migration GEDF schedule

(c) theoretical improved full-migration GEDF schedule

(d) actual improved full-migration GEDF schedule

Figure 1: Schedules for scheduling the task set in Ex. 1.

Task	Response Time (ms)		
	in theory	full-migration	improved
τ_1	0.80	1.11	1.14
τ_2	1.60	2.56	1.88
τ_3	2.40	3.73	2.67
τ_4	3.20	5.12	3.54
τ_5	4.16	6.47	4.57
τ_6	5.12	7.83	5.55
τ_7	6.08	8.94	6.53
τ_8	7.04	9.80	7.51

Table 4: Response-time comparison.

two different speeds—$s_1 = s_2 = s_3 = s_4 = 1.25$ and $s_5 = s_6 = s_7 = s_8 = 1$. Thus, in our experiments, it is in fact unnecessary to migrate a task among the first four processors or the last four processors, as a *general* full-migration GEDF scheduler does. That is, we can group the processors by speeds, and we only commit a migration if it is an inter-group migration.

Fig. 1(c) depicts the theoretical schedule of the task set in Ex. 1 under this migration approach. As seen, all tasks' response times or even execution progresses are identical to that in Fig. 1(a), despite the migration rules have been changed. In fact, any theoretical scheduling results for full-migration GEDF will also follow in this less-migration schedule, because we can inter-change these two schedules by dynamically re-indexing processors, and re-indexing two processors of the same speed makes no difference in theory.

We implement these migration rules by a swap procedure of the `wanted` array in `schedule()`. The code of this improved scheduler is `fgedf-impv.c`. Fig. 1(d) is the actual schedule of the task set in Ex. 1 under the improved scheduler in the system. As seen, the migrations are reduced dramatically, and the response times are improved significantly. Table. 4 shows a response-time comparison, in which for each task, its theoretical response time, actual response time under full-migration GEDF, and actual response time under improved migration rules are provided.

5 Schedulability

To evaluate the implemented schedulers, we conduct schedulability experiments. In these experiments, we do run the generated task sets in the actual system, and record the scheduling log files. In contrast to many analytical schedulability tests in theoretical work that evaluate *guaranteed* schedulability, we focus on *measured* schdulability. That is, we actually run the tasks in our constructed system for a reasonable time duration, and then check if any deadline has ever been missed. If so, we mark this task set is not schedulable; otherwise, we mark this task set is schedulable. Moreover, we only consider *synchronous implicit-deadline* periodic tasks, *i.e.*, for any i, $\phi_i = 0$ and $D_i = T_i$ (recall that a task is specified as $\tau_i = (\phi_i, C_i, D_i, T_i)$).

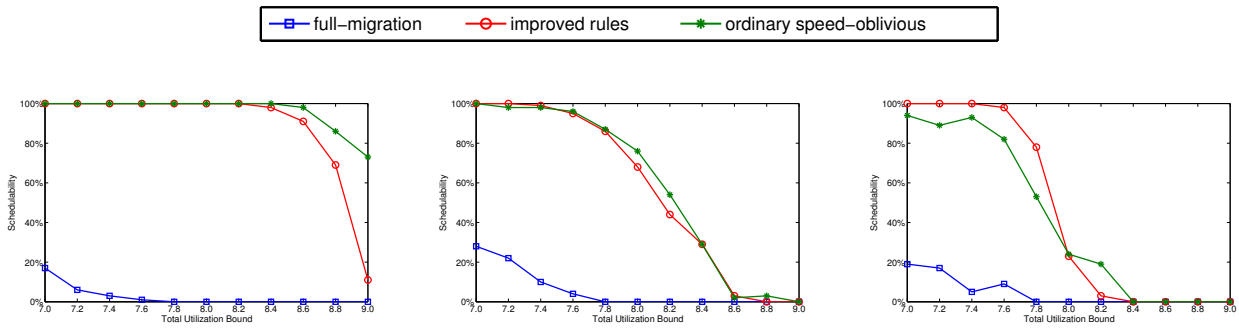
The constructed system consists of four processors with a speed 1.25 and other four processors with a speed 1.0, so the platform total speed, or capacity, is 9. We randomly generate task sets with a total utilization in $[7, 9]$[3] with an incremental step of 0.2. For each designated total utilization, we generate 100 task sets, and we run each task set for 10 seconds, *i.e.*, $10,000$ms. In terms of randomly generating the utilization of each individual task (u_i), we have three different range settings: light $[0.1, 0.5]$, medium $[0.5, 0.9]$, and heavy $[0.9, 1]$[4]. Then, we choose a task's period (T_i) from a log-uniform distribution with the range $[10\text{ ms}, 1000\text{ ms}]$, *i.e.*, we first uniformly choose an x from $[1, 3]$ and then 10^x is the period. Now, the execution cost of the task (C_i) can be computed from its utilization (u_i) and period (T_i). Note that we approximate all periods and execution costs to integers since the middleware only supports integer parameters. Therefore, the actual utilization of a task could be slightly lower than its designated utilization. Nevertheless, it has little affect on our experiments; we only need to interpret the designated total utilization as a total utilization bound instead of an exact total utilization.

In this project, the schedulability of an algorithm is defined as the percentage of schedulable task sets under that algorithm among all generated task sets of a certain designated total utilization. Fig. 2 shows the measured schedulability under strictly full-migration GEDF, full-migration GEDF with improved migration rules, or ordinary speed-oblivious GEDF, where the last one is the general GEDF designed for identical multiprocessors, *i.e.*, the scheduler never migrates a task that is current executing, even if it is executing on a slower processor and a fast processor is available.

As seen, the purely full-migration GEDF has a

[3] We only focus on those "hard to schedule" cases, *i.e.*, the total utilization is approaching the system capacity

[4] Since we have processors with a speed 1.25, we may support some "very heavy" tasks with a utilization in $(1, 1.25]$. However, it exceeds the speed of the slower processors, and therefore needs a more complicated feasibility-aware generation pattern. Thus, we do not consider this situation in this project.



(a) light tasks (b) medium tasks (c) heavy tasks

Figure 2: Measured schedulability.

very poor performance even compared to a speed-oblivious scheduler, which means the migration overheads significantly outweighs the theoretical advantages by those migrations. Furthermore, even when the improved migration rules are applied, the improved full-migration scheduler only slightly outperforms the speed-oblivious one in the case that individual task utilizations are heavy. Thus, this project shows that those theoretically beneficial migrations may not be worth to be committed in real implementations. Nevertheless, such observations and the conclusion are based only on the platform we construct, where the speed difference is only 2.5 GHz vs. 2.0 GHz. Maybe a larger speed gap would make the merits of those migrations outweigh the overhead incurred by them, which may potentially lead to interesting future work.

6 Conclusion

By leveraging CPU frequency scaling techniques, we have constructed a uniform heterogeneous multiprocessor in an identical multiprocessor system. We have implemented a full-migration GEDF scheduler as a middleware plug-in, and have improved its migration rules. We have also evaluated the schedulers by providing response-time comparisons in a particular example and by measuring the schedulability of randomly generated task sets.

Acknowledgments

The author would like to thank Mac Mollison for his patient and detailed instructions for installing the middleware, valuable discussions about this project, and directions for working on Koruna and Pound. The author also would like to thank Bryan Ward for his suggestions and directions for working on Zildjian.

References

[1] big.LITTLE Processing. http://www.arm.com/products/processors/technologies/ biglittleprocessing.php.

[2] cpufreq-set Linux man page. http://linux.die.net/man/1/cpufreq-set.

[3] Enhanced Intel SpeedStep® Technology. http://www.intel.com/cd/channel/reseller/ASMO-NA/ENG/203838.htm.

[4] Power management/Processor Gentoo Wiki. http://wiki.gentoo.org/wiki/Power_management/Processor.

[5] Samsung's Exynos 5422 & The Ideal big.LITTLE: Exynos 5 Hexa (5260). http://www.anandtech.com/show/7811/samsungs-exynos-5422-the-ideal-biglittle-exynos-5-hexa-5260.

[6] Time Stamp Counter - Wikipedia. http://en.wikipedia.org/wiki/Time_Stamp_Counter.

[7] S. Baruah and Z. Guo. Mixed-criticality scheduling upon varying-speed processors. In *34th RTSS*, 2013.

[8] S. Funk. *EDF Scheduling on Heterogeneous Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2004.

[9] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. In *22nd RTSS*, 2001.

[10] Z. Guo and S. Baruah. Mixed-criticality scheduling upon varying-speed multiprocessors. In *12th DASC*, 2014.

[11] E. Horvath, S. Lam, and R. Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):3243, 1977.

[12] H. Leontyev and J. Anderson. Tardiness bounds for EDF scheduling on multi-speed multicore platforms. In *13th RTCSA*, 2007.

[13] M. Mollison and J. Anderson. Bringing theory into practice: a userspace library for multicore real-time scheduling. In *19th RTAS*, 2013.

[14] M. Mollison and J. Anderson. Virtual real-time scheduling. In *7th OSPERT*, 2011.

[15] M. Pinedo. *Scheduling, Theory, Algorithms, and Systems*. Prentice Hall, 1995.

[16] K. Yang and J. Anderson. An optimal semi-partitioned sche-duler for uniform heterogeneous multiprocessors. In *submission*.

[17] K. Yang and J. Anderson. Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *12th ESTIMedia*, 2014.

[18] K. Yang and J. Anderson. Soft real-time semi-partitioned scheduling with restricted migrations on uniform heterogeneous multiprocessors. In *22nd RTNS*, 2014.