Cooperative Multitasking for Heterogeneous Accelerators in the Linux Completely Fair Scheduler

Tobias Beisel, Tobias Wiersema, Christian Plessl and André Brinkmann
Paderborn Center for Parallel Computing, University of Paderborn, Germany
Email: { tbeisel | tobias82 | christian.plessl | brinkman }@uni-paderborn.de

***Abstract*—This paper presents an extension of the Completely Fair Scheduler (CFS) to support cooperative multitasking with time-sharing for heterogeneous processing elements in Linux. We extend the kernel to be aware of accelerators, hold different run queues for these components and perform scheduling decisions using application provided meta information and a fairness measure. Our additional programming model allows the integration of checkpoints into applications, which permits the preemption and subsequent migration of applications between accelerators. We show that cooperative multitasking is possible on heterogeneous systems and that it increases application performance and system utilization.**

I. Introduction

Heterogeneous accelerator environments have become ubiquitous with the advent of multi-core CPUs and general-purpose graphics processing units (GPUs). This heterogeneity is also observable in compute centers, where cluster systems use combinations of multi-core processors, GPUs, and specialized co-processors, such as ClearSpeed CSX or FPGAs, to accelerate scientific applications [1].

The usage of such systems is still limited though, since most accelerators need to be programmed with unfamiliar programming languages and APIs. Developing efficient software for these architectures requires knowledge about the underlying hardware and software components. Hence many recent research efforts address the challenges to ease the development and use of hardware accelerated code. While this research area is of high relevance, we instead approach the challenge of performing scheduling decisions at runtime and treat accelerators as peer computation units that are managed by the operating system (OS) kernel like CPU cores. The goal of scheduling tasks in the context of heterogeneous systems is to assign tasks to compute units in order to enable time sharing of accelerators and to provide fairness among tasks that compete for the same resources.

Scheduling tasks to heterogeneous accelerators raises a number of practical challenges, the most important being that some accelerators such as GPUs do not support preemption and that the migration of tasks between different accelerators is complicated due to largely different execution models and machine state representations. While computations on CPU cores can be easily preempted and resumed by reading and restoring well defined internal registers, most accelerators assume a run-to-completion execution model and do not even expose the complete internal state, nor are they designed to be interrupted. Also, accelerators typically do not have autonomous access to the shared memory space of the CPU cores and explicit, non-uniform communication of input data and results is required. Communication characteristics also determine the granularity of the functionality that can be successfully scheduled without too much overhead. Finally, their completely different architectures and ISAs require a dedicated binary for each combination of task and accelerator, which prevents migrating tasks between arbitrary compute units. Even if a task with the same functionality is available for several architectures and if the internal state of the architecture is accessible, migrating a task between different architectures is far from trivial.

Scheduling tasks to heterogeneous accelerators has hardly been investigated by OS developers so far and is mostly managed as part of the application. This implies that hardware accelerators are used exclusively by one particular application without any time-sharing. The contribution of this work is an extension to the Linux Completely Fair Scheduler (CFS) that 1) provides awareness of installed accelerators, 2) enables scheduling of specific tasks to accelerators, 3) allows time-sharing and task migration using a cooperative multitasking and checkpointing approach.

A comparable approach has, e.g., been made by Lübbers et al. [2], who extended the Linux and eCos operating systems by a uniform interface for software threads on the CPU and hardware threads on FPGA accelerators. Kosciuszkiewic et al. [3] used an existing Linux OS kernel and regarded hardware tasks as a dynamic drop-in replacement for software tasks, such that tasks are universal for soft- and hardware and appear as regular threads to the OS kernel. Our approach is non-preemptive, but allows tasks to release a compute unit upon a request by the scheduler and thus increases the fairness of the scheduling approach. Integrating time-sharing using the Linux thread model on heterogeneous systems is a novel approach that reduces the response times of the tasks and optimizes the accelerator's utilization and the performance of universal applications.

This work has been partially supported by the German Ministry for Education and Research (BMBF) under project grant 01|H11004 (ENHANCE).

Published in: *Proc. Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*
 DOI: 10.1109/ASAP.2011.6043273

II. Linux Kernel Extensions

This section describes our changes to the Linux kernel, enabling scheduling of heterogeneous processing elements.

A. Design Decisions

To allow a holistic view on the applications and its execution environment, we perform the scheduling in the systems kernel. That way the scheduling principles are still hidden from the application developer and the OS can perform global decisions based on the system utilization. Application specific scheduling inputs still have to be provided by the application developer to incorporate application's needs. Therefore we use a hybrid user/kernel level approach to perform heterogeneous scheduling. A specific interface has to be provided to allow communication between the application and the scheduler component.

Tasks that execute on heterogeneous resources have no access to main memory and use a completely different instruction set or execution model than an equivalent task on a CPU. In order to schedule and manage these tasks without requiring a major OS rewrite, we represent each task executing on a hardware accelerator as a thread to the OS. This allows us to use and extend the existing data structures of the scheduler within the Linux kernel. We denote each thread representing a task on an accelerators as a *delegate thread*. Apart from serving as a schedulable entity, the delegate thread also performs all operating system interaction and management operations on behalf of the task executing on the accelerator unit and forwards requests from the kernel scheduler to the task on the accelerator.

Since we build upon the CFS, which requires preemptive multitasking based on a fairness measure our scheduler also has to include means to preempt a task and to migrate it to another computing unit. While non-voluntary preemption on FPGAs is possible, GPUs currently do not directly support it yet, even if it is planned for the future. We propose to use a cooperative multitasking approach using checkpoints to overcome the given limitations. After reaching a checkpoint, an application voluntarily hands back the control to the OS, which then may perform scheduling decisions to suspend and migrate a thread at these points.

B. Scheduler API

With respect to the cooperative use of the scheduler, we provide an interface to the scheduler, which enables user space applications to request (allocate), re-request and free compute units. The allocation call requests and acquires that compute unit, which matches the calling task best by enqueueing the task to the associated waiting queue. The assignment is done using an affinity approach, where a preference for a given accelerator, the current length of the waiting queues, and the load of the compute units are considered.

Our CFS extension allows the migration of threads from one compute unit to another if the application provides implementations for both. Migration of a thread may be performed while it is blocked within a waiting queue or, if it is running on any of the available compute units, when it reaches the next checkpoint. Checkpointing has to be performed when the application can safely interrupt its execution and store its state in main memory, where the checkpoint can be accessed from all accelerators. If the program reaches a checkpoint, it requests to further use the compute unit, but offers to voluntarily release it (cf. Figure 1). The scheduler decides if the task on the compute unit should be replaced by another. The time a task may run on an accelerator follows the CFS approach. It is the sum of the fairness delta, i.e., the time to compute until the (negative) unfairness is equalized, and the accelerator-specific time slice, i.e., the task's "positive unfairness". Dynamic task migration on heterogeneous systems is a major contribution of our work. If the application has either finished its work, unsuccessfully re-requested its compute unit or simply wants to yield, the compute unit is released and handed to the next task or, in case no tasks are waiting on this device, a load balancer is invoked that searches for a suitable task to be migrated to the idle device.

C. Control API

Using most of todays accelerators involves using their proprietary user space APIs to copy code or data to and from the device and to invoke programs on it. Since there are virtually no implementations to communicate efficiently with these devices from the kernel, our extension leaves all interaction with the accelerators to the user space. We provide system calls to add a compute unit, to remove it afterwards, to iterate through all currently added units and to alter a device after it has been added.

III. Programming Model

Figure 1 describes the typical lifecycle of a thread using the provided system calls in our extended CFS. Besides the information about the system status the scheduler needs to have meta information about the thread to be scheduled. As this information cannot be derived easily from a binary, it has to be provided by the application programmer. Additionally, the code to be executed and the needed input data has to be copied to the compute unit after it has been allocated. The delegate thread can then execute its main function in a loop. If a re-request fails before the thread has finished, it writes a checkpoint and waits for the allocation of a new compute unit for taking up its computation.

We provide code templates in C++, which are not mandatory for using the extended functionality of the CFS, but simplify application development. We introduce a *worker* class that is the superclass of the delegate threads in an application. The worker class provides the virtual functions

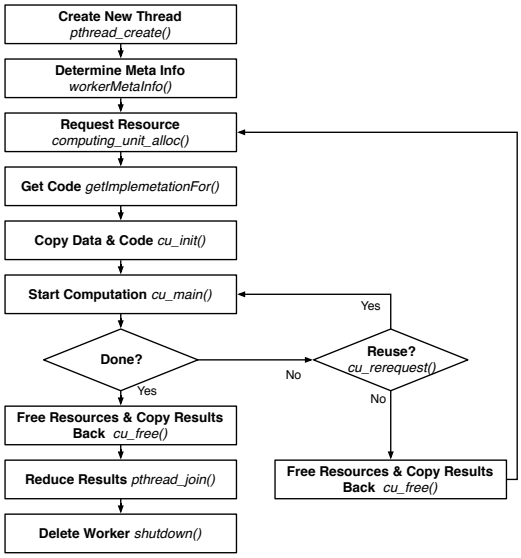

Figure 1. Typical workflow of a delegate thread.

getImplementationFor and *workerMetaInfo*, which have to be implemented in the derived delegate threads.

The *workerMetaInfo* method implemented by a worker instance includes the mandatory values for the meta information, which ensures that only compatible compute units are assigned to the thread. In our examples, we used an affinity value to give an approximate hint of how much the implementation for one compute unit outperforms the other.

The *getImplementationFor* function provides a function pointer to the worker implementation for the allocated compute unit type. It has to provide the three functions *cu_init*, *cu_main*, and *cu_free* for all supported compute units. *cu_init* allocates memory on the compute unit and copies needed resources to it. These resources are used in the computation performed in the *cu_main* function, which does all the work between two checkpoints. The resources have to be copied back to main memory and freed after finishing a computation or being denied a re-request, a functionality which has to be provided in the *cu_free* function.

We define a checkpoint as a struct of data structures that defines an application's state. The checkpoint data is copied to the compute unit within the *cu_init* function, is provided for each call of the *cu_main* function and copied back to main memory when the *cu_free* function is called. We assume that suitable checkpoints are defined by the application developer, which needs to identify a preferably small set of data structures that 1) unambiguously define the state of a thread and 2) can be translated to matching data structures of other compute units.

IV. Evaluation

In this section we present applications using the extended scheduler, which dynamically switch between CPU cores and a GPU. We evaluate the overheads and show how adjusting the scheduler parameters affects the runtimes.

A. Example Applications

We used a brute-force MD5 hash cracker (MD5) and a prime number factorization (PF) as example applications to evaluate our scheduler extension. Both applications were implemented in C++ for the CPU and CUDA C to be run on NVIDIA GPUs. In both cases we examine the execution of several application instances concurrently.

1) MD5 Cracking: MD5 brute-force cracking enumerates all possible strings of a fixed length with letters of a given alphabet and computes their MD5 hash value until a match with a target hash value is found. Each string is identified by an ascending unique number that can be used as a checkpoint specifying the progress of the search. We chose the interval between two checkpoints differently for the CPU and the GPU in order to consider the costs for retransferring the kernel to the compute unit and doing a re-request at a checkpoint, which are much higher on the GPU.

The meta information for this application is very simple, as practically no memory has to be copied. The affinity can be set depending on the size of the string length and the used alphabet, which defines the problem size. The performance differences between CPU and GPU are known to be very high here, hence the meta information is set up to express a clear affinity to the GPU.

The vertical axis in Fig. 2 represents the search space that has been searched, while the horizontal axis denotes the runtime in seconds. We ran 15 GPU affine threads, with a limited queue length of 5 for the GPU. As we spawn 15 threads, 9 threads initially run on CPU cores, 6 (1 running, 5 queued) on the GPU. Each thread is displayed by a different color. The GPU threads can be identified by the fast progress in the search space. The ascend of the curves representing CPU threads can hardly be seen, only a minor progress can be noticed in region e). The ability of the tasks to be stopped and replaced for later continuation on the GPU can e.g., be seen in region a), where 6 GPU threads share the computation time based on their fairness, their time slices, and the time needed to reach the next checkpoint.

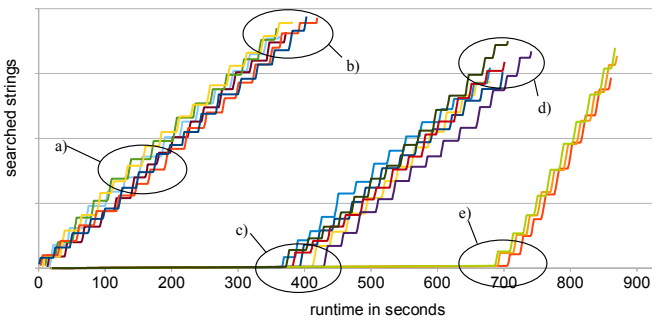

Figure 2. Running 15 independent MD5 instances.

In this example the load balancer refills the GPUs queue as soon as a thread finishes its computation (regions b) and d)). Regions c) and e) show how CPU threads are migrated to the GPU to afterwards compute much faster. Thus, cooperative multitasking with time sharing is possible using the introduced checkpoints.

2) Prime Factorization: Prime factorization algorithms decompose numbers into their prime factors. Our sample application searches through all possible divisors of the number up to its square root. Whenever a divisor is encountered the number is divided by it as often as possible. Hence the application yields the divisor and its multiplicity. It then continues the search, now using the current remainder and its square root instead of the original number.

Checkpoints are defined by the pair of current remainder and potential divisor. Between two checkpoints the algorithm advances the divisor to the value of the second checkpoint, while keeping the remainder correct during the process. In our version the checkpoints are $1,000$ divisors apart.

B. Scheduler Evaluation

Our evaluation was performed with the following setup:

- SMP system with 2-way 4-Core Intel Xeon CPU E5620 @ 2.4GHz, hyperthreading enabled, 12 GB DDR3
- NVIDIA Geforce GTX 480, 1536 MB GDDR
- Ubuntu Linux 10.04.1 LTS, modified 2.6.32-24 kernel
- CUDA 3.1, GCC 4.4.3

1) Performance Evaluation: The turnaround time is the interval between the submission of a process and its completion. The effect of time sharing on turnaround times can be seen in Fig. 3. Subfigure a) shows the turnaround times of 25 MD5 and 50 PF instances with one thread spawned by each instance in batch mode. As tasks are scheduled one after another, long running tasks (e.g., tasks 1, 2, 5) block all other tasks, such that the average turnaround time is increased. Using time sharing, short tasks are not blocked any more, such that the average turnaround time is lower. Subfigure b) shows that the tasks are not finished in the same order as they are started. Shorter jobs are not blocked, as their time slice is of equal length as of long jobs.

2) Overheads: Decreasing the size of the time slices raises the total runtime, as task switching overheads are introduced. Introducing time sharing is therefore a trade off between overheads and response times, where the latter are reduced by short time slices and FCFS scheduling (state of the art on GPUs) obviously has the smallest overhead. This is also emphasized in Fig. 4, which shows the average turnaround time depending on the used size of the time slices. Using time slices of 0 leads to fast-paced task switching and short response times and thus introduces huge overheads. On the other hand, submitting all tasks at the same time to the GPU queue and using FCFS for the tasks on the GPU results in higher average turnaround times due to long running tasks blocking short ones.

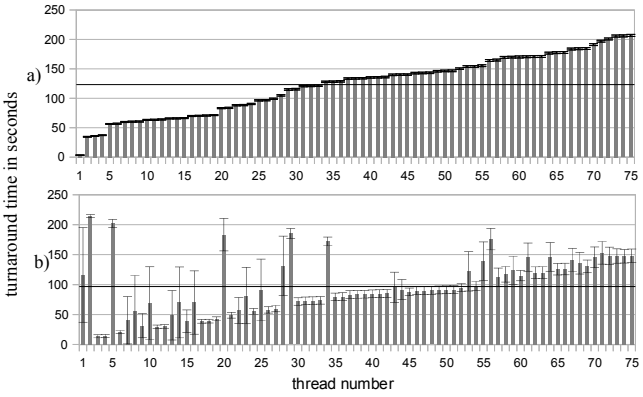

Figure 3. Turnaround times for all threads using 25 MD5 and 50 PF threads with 4s time slices a) without time-sharing and b) with time-sharing.

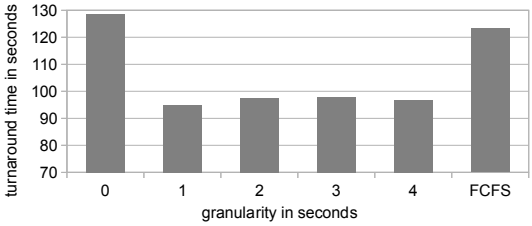

Figure 4. Avg. turnaround times per thread with 25 MD5 threads (string length 6) and 50 PF threads on the GPU.

V. Conclusions

We presented an extendable approach to integrate the scheduling of heterogeneous accelerators into the Linux kernel. We introduced the use of cooperative multitasking with checkpoints to heterogeneous systems and provided an implementation that allows the preemption and a subsequent migration of threads between GPUs and CPU cores. The migration is done automatically based on an affinity metric associated with the compute units and the current system status. This not only reduces the average load on the CPU while at least preserving the application's performance, but also allows intelligent task scheduling to maximize application performance and system utilization. The developed source code is available at: http://github.com/pc2/hetsched.

References

[1] K. Volodymyr, R. Wilhelmson, R. Brunner, T. Martïnez, and W. Hwu, "High-Performance Computing with Accelerators (Special Issue)," *IEEE Computing in Science & Engineering*, vol. 12, no. 4, Jul./Aug. 2010.

[2] E. Lübbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Trans. on Embedded Computing Systems (TECS)*, vol. 9, no. 1, pp. 1–33, 2009.

[3] K. Kosciuszkiewicz, F. Morgan, and K. Kepa, "Run-time management of reconfigurable hardware tasks using embedded Linux," in *Int. Conf. on Field-Programmable Technology (ICFPT)*. IEEE CS, 2007, pp. 209–215.