Programming and Scheduling Model for Supporting Heterogeneous Accelerators in Linux

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

***Abstract*—Computer systems increasingly integrate heterogeneous computing elements like graphic processing units and specialized co-processors. The systematic programming and exploitation of such heterogeneous systems is still a subject of research. While many efforts address the programming of accelerators, scheduling heterogeneous systems, i. e., mapping parts of an application to accelerators at runtime, is still performed from within the applications. Moving the scheduling decisions into an external component would not only simplify application development, but also allow the operating system to make scheduling decisions using a global view.**

In this paper we present a generic scheduling model that can be used for systems using heterogeneous accelerators. To accomplish this generic scheduling, we introduce a scheduling component that provides queues for available accelerators, offers the possibility to take application specific meta information into account and allows for using different scheduling policies to map tasks to the queues of both accelerators and CPUs. Our additional programming model allows the user to integrate checkpoints into applications, which permits the preemption and especially also subsequent migration of applications between accelerators. We have implemented this model as an extension to the current Linux scheduler and show that cooperative multitasking with time-sharing enabled by our approach is beneficial for heterogeneous systems.

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 do not address the task of creating programs or configurations for hardware accelerators in this paper.

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

Instead, we approach the challenge of performing scheduling decisions at runtime and treating hardware 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 hardware 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. Also, the scheduling process itself is more complex than process scheduling for homogeneous CPU cores, since each scheduling decision requires to incorporate specific hardware characteristics (e. g., the communication bandwidth or memory sizes) and needs to consider the current availability and state of the heterogeneous compute units. In addition, knowledge about the availability and suitability of a task for a particular hardware accelerator is required. This information is highly application specific and has to be provided by the application developer. Scheduling tasks to hardware accelerators has been neglected by OS developers so far and is 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 a general programming and scheduling model for heterogeneous systems and an example implementation in Linux. We provide an extension to the Linux Completely Fair Scheduler (CFS) that 1) provides awareness of installed accelerators, 2) enables scheduling of specific tasks to accelerators, and 3) allows time-sharing and task migration using a cooperative multitasking and checkpointing approach. Our approach is non-preemptive, but allows tasks to release a compute unit upon a request by the scheduler and thus increases the fairness among tasks. Dynamic task migration on heterogeneous systems is a major contribution of this approach.

The scheduler hardware selection decision is based on meta information provided by the applications. While we supply a basic scheduling policy based on static affinities to accelerators, our focus is to provide a framework for heterogeneous scheduling using a time-sharing approach. We evaluate our work with two example applications that prove the usability and benefits of the approach and supply data for an efficiency analysis.

This work is an extension and more comprehensive discussion of a previous work of ours, in which we already presented a prototype implementation of a linux kernel extension supporting heterogeneous systems [2]. In this paper we provide a more general view onto the problem and describe the kernel extension in more detail.

The remainder of this paper is structured as follows. We introduce a general scheduling model in Section II. Afterwards we describe our newly developed Linux kernel extension in Section III and present an according programming model in Section IV. Section V evaluates the contributions with two example applications. After a discussion of related work in Section VI, we finish the paper with discussing future work and drawing conclusions.

II. General Scheduling Model for Heterogeneous Systems

The CFS schedules processes in current Linux SMP systems. CFS is a preemptive scheduler that guarantees fairness with respect to the allocation of CPU time among all processes. The scheduler aims to maximize the overall utilization of CPU cores while also maximizing interactivity. An inherent precondition for the use of such a preemptive scheduler is that processes can be preempted and also migrated between computing resources.

In this work, we address scheduling in a heterogeneous computer system with non-uniform computing resources. We target computer systems, which include single- or multi-core CPUs operating in SMP mode and an arbitrary combination of additional hardware accelerators, such as GPUs, DSPs, or FPGAs. Scheduling such systems is more difficult due to several reasons:

1) Accelerators typically do not have autonomous access to the shared memory space of the CPU cores and explicit communication of input data and results is required. The most important impact on a scheduling decision is the introduction of data transfer times that rely on available bandwidths and the data to be copied. These overheads have to be known and used as input for a scheduling decision in heterogeneous systems. Further, the communication bandwidth, latency, and performance characteristics of accelerators are non-uniform. These characteristics also determine the granularity of the task that can be successfully scheduled without too much overhead (single operations, kernels, functions/library calls, threads). Scheduling decisions thus usually have to be more coarse-grained than on CPUs.
2) Most accelerator architectures do not support preemption but assume a run-to-completion execution model. While computations on CPU cores can be easily preempted and resumed by reading and restoring well defined internal registers, most hardware accelerators do not even expose the complete internal state nor are they designed to be interrupted.
3) Heterogeneous computing resources have completely different architectures and ISAs. Hence, a dedicated binary is required 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, because the representation and interpretation of state is completely different.

A. Design Decisions

In this section, we discuss and describe basic design decisions made for our scheduling framework.

1) Scheduler Component: Scheduling of homogeneous CPU cores is currently done in the kernel, as all needed input information for the scheduling decision is available to the system, so that the scheduling problem can be completely hidden from the application programmer. The heterogeneous scheduling problem is more complicated, as more decision parameters affect the decision, which are partly not available to the systems scheduler component.

Selecting an appropriate hardware architecture for a task to be scheduled dynamically at runtime is non trivial and has to be performed by a scheduler, which can be located at different locations in the system, either in the application, in user space or in the system's kernel.

To allow a holistic view on the applications and its execution environment, we perform scheduling in the system's kernel by extending the CF scheduler. 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 application and scheduler.

2) Adapting the Operating System: Kernel space scheduling is the current standard in operating systems. To provide support for heterogeneous architectures one could either extend an existing OS or completely rewrite and fully optimize it towards the heterogeneity. While heterogeneous systems will be more and more used in future and become standard in a foreseeable time, we believe that a complete rewrite of

the OS is not needed. An extension to the current system has several advantages: Providing a modular implemented extension to the CFS 1) keeps the management structures as well as the scheduler component exchangeable, 2) makes the changes easily applicable to other OS, and 3) reuses well established and well known functionalities of the current kernel that have been developed over years. That way our kernel extension will help to explore new directions for future OS, but does not yet try to set a new standard.

3) Delegate Threads: Tasks that execute on heterogeneous resources may 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 need to expose the tasks to the OS as known schedulable entities. We therefore 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 in the Linux kernel. We denote each thread representing a task on a hardware 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, such as transferring data to and from the compute unit and controlling its configuration and execution. The delegate threads must be spawned explicitly by the application and thus can also be used for co-scheduling on different architectures. Once created, all threads are treated and scheduled equally by the operating system.

4) Cooperative Multitasking: The CFS implements preemptive multitasking with time-sharing based on a fairness measure. Therefore, our scheduler 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 [3]. Therefore we use the delegate threads to forward requests from the kernel scheduler to the task on the accelerator.

Nevertheless, even enabling preemption on GPUs does not solve the migration problem. The major difficulty is to find a way of mapping the current state of a compute unit to an equivalent state on a different compute unit. To allow preemption and subsequent migration of applications on heterogeneous systems, their delegate threads need to be in a state, which can be interpreted by other accelerators or by the CPU. As it is not possible to interrupt an accelerator at an arbitrary point of time and to assume that it is in such a state, we propose to use a cooperative multitasking approach using checkpoints to resolve these 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. We believe that this currently is the only way to simulate preemptive multitasking on heterogeneous hardware.

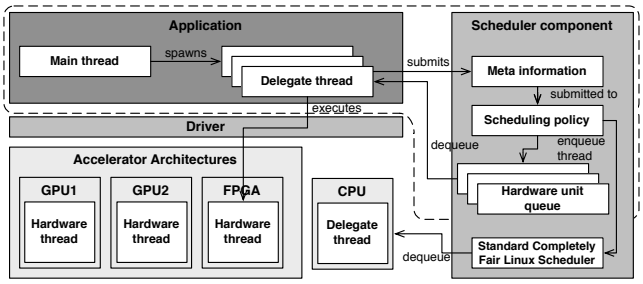

Figure 1. Novel scheduling model for heterogenous systems. New parts are surrounded by dashed lines.

In this paper, we focus on the underlying framework and assume that the application developer defines such states in his application and that he provides a delegate thread, which interacts with the scheduler and the accelerator. The structure of the delegate thread (cf. Section IV) as well as providing relocatable system states is generic enough that these can be automatically generated by a compiler in the future.

B. Scheduling Model

From the design decisions above we derive our scheduling model shown in Figure 1 that is not restricted to a certain class of operating systems or scheduling algorithms. Applications using the scheduler may spawn several threads that may possibly run on diverse architectures.

Thread information: As the scheduler needs information about the threads to be scheduled, we store this application provided information called *meta information* about each thread and submit it to the scheduler. The meta information can be individually set for an application. Currently we only use a type affinity towards a target architecture, which can be determined dynamically depending on the input data. Further application specific input data can possibly be determined using profiling prior to the first use of an application. While this is not in the focus of this paper, one could think of useful values like estimated runtimes, required memory sizes or data transfer sizes.

Scheduling: The scheduler component may be located in the kernel space as well as the user space. To assign tasks to certain hardware components, the scheduler has to provide a queue for each available hardware. The application provided meta information is used in a scheduling policy to map newly arriving tasks to one of the queues. Whenever a compute unit runs idle or the currently running task has used its complete time slice, the scheduler may dequeue a waiting task for that specific compute unit. In case this is a hardware task, the delegate thread receives the information that it may run its hardware counterpart. This includes using the proprietary drivers of the hardware, which are inevitable for the communication with some accelerators. As these currently may only be used from user space, this requires

a combined approach using the kernel space and the user space. For CPUs, the standard Linux scheduler is used.

Checkpointing: Checkpointing has to be performed when the application can safely interrupt its execution and store its state in main memory. The state has to be stored by the application itself in data structures of the corresponding delegate thread, which then can be migrated to a different architecture. The checkpoint data of the delegate thread thus has to be readable by all target architectures.

We define a checkpoint as a struct of data structures that unambiguously defines the state of the application. The scheduler does not have any requirements concerning the checkpoint data. Hence, the application has to make sure that all needed data is available in these data structures and thus stored in accessible memory at the end of each thread's time-slice. A checkpoint in most cases is a combination of 1) a set of data structures that define a minimum state that is reached several times during execution, and 2) a data structure that define the position in the code. The checkpoint data of an application is copied to the newly allocated accelerator and copied back to the host's main memory when the application's time slice is exhausted.

Checkpoints are to be defined by the application developer or to be inserted by a compiler. One has to identify a preferably small set of data structures that 1) unambiguously define the state of a thread, and 2) are readable and translatable to corresponding data structures of other compute units. The size of checkpoints may vary to a large extend depending on the application used. While MD5 cracking (cf. Section V) only needs to store the current loop index (i.e., a hash value) and the given search-string, image processing algorithms (e.g., medical image processing) require to store the complete intgrermediate results that might be of large extent. In general, a checkpoint could be simply defined by a list of already processed data sets. Therefore, the choice of the checkpoint is very important and influences the scheduling *granularity*. The checkpoint distance, i.e., the amount of work done between 2 checkpoints stored back, increases with the size of the checkpoint.

We here assume all checkpoints to be small enough to fit into the host's memory. The introduced checkpoint size is known at definition time and may be used to re-determine the scheduling granularity for a task. Please refer to Sections IV and V for examples and implementation details about the meta information and checkpoints, or directly to the example implentations (cf. Section VIII).

III. Linux Kernel Extensions

This section shortly introduces the changes made to the Linux kernel to enable the scheduling of heterogeneous hardware accelerators according to our scheduling model. Please refer directly to the source code for implementation details (cf. Section VIII).

A. Data Structures

Following the goal to extend the current Linux scheduler, we have to make the kernel aware of existing heterogeneous hardware accelerators. The CFS uses its queues and statistics to ensure a fair treatment of all tasks with respect to their priorities. Its queue is ordered by the amount of unfairness, i.e., the time the task would have to execute undisturbed to be treated fair. We extend the kernel with a specific task struct for hardware threads and a semaphore protected queue for each of the available accelerators.

The current implementation of the meta information includes the memory size to be copied and an array of type affinities. The higher a task's affinity to a compute unit is, the better is the estimated performance on this compute unit.

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-based approach, where the given affinity, as well as the current length of the waiting queues and the load of the compute units are included.

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 even if it is running on any of the available compute units. Since there are no means of directly migrating the tasks from one instruction set to another, migration is achieved by a combination of checkpointing and cooperative multitasking.

If the program reaches a checkpoint, it requests (re-requests) to further use the compute unit, but offers to voluntarily release it (also compare Figure 2). The scheduler decides if the task on the compute unit should be replaced by another, which depends on the type of compute unit and on the cost of switching the task. Re-requests inside the time window of an accelerator-specific granularity are always successful and will only be denied after the granularity has expired and if other tasks are waiting for the resource. 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 granularity, i.e., the "positive unfairness" for this task.

To enable dynamic load balancing on CPU cores and GPUs, a load balancing component managing running and queued tasks was introduced. If the application has either finished its work or unsuccessfully re-requested its compute unit, it calls a *free* function. This releases the compute units semaphore and hands it to the next task or, in case no other tasks are waiting on this device, invokes the load balancer.

If a task is waiting for the semaphore of a compute unit and another suitable unit is running idle in the meantime then the load balancer wakes the task with a migration flag. The task then removes itself from the waiting queue and enqueues on the idle compute unit. Using this mechanism the scheduler achieves a late binding of tasks to units, which ensures a better utilization of the resources with only a negligible amount of computation overhead in the scheduler, as most tasks are blocked and thus migrated while waiting on the semaphore of a compute unit. The load balancer at first traverses the run-queues of all other compute units and tries to find the task with the maximum affinity to the idle compute unit. If the balancer is not able to find a suitable waiting task, it parses through all running tasks, which are currently being executed on other units.

C. Control API

Using most of todays hardware 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.

IV. Programming Model

This section describes the design of applications using the provided system calls of our kernel extension. Additionally, we describe a design pattern for implementing an application worker thread (delegate thread), which is not mandatory for using the extended functionality of the CFS, but simplifies application development.

A. User Application Workflow

Figure 2 describes the typical lifecycle of a thread in our extended CFS. Besides the information about the system status the scheduler needs to have meta information about the thread to be scheduled. Additionally, the code to be executed and the needed input data has to be copied to the compute unit after it has been acquired by using the blocking allocation call. The worker then can execute its main function in a loop. If a re-request fails before the worker is done, it writes a checkpoint and waits for the allocation of a new compute unit for taking up its computation.

B. Worker Implementation

We provide code templates in C++ to 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 *getImplementationFor* and *workerMetaInfo*, which have to be implemented in the derived delegate threads.

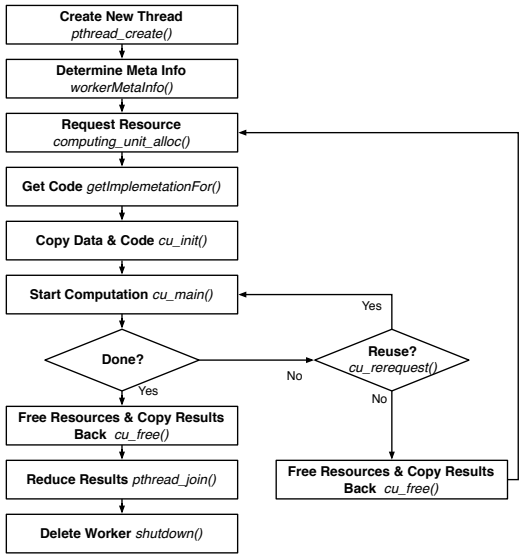

Figure 2. Typical workflow of a delegate thread.

```
void Worker_example::workerMetaInfo(struct meta_info *mi){
 mi->memory_to_copy=0;              // in MB
 mi->type_affinity[CU_TYPE_CUDA]=2;
 mi->type_affinity[CU_TYPE_CPU]=1;
}

void* Worker_example::getImplementationFor(int type,
     functions *af)
switch(type) {
  case CU_TYPE_CPU:
   af->init=&Worker_example::cpu_init;
   af->main=&Worker_example::cpu_main;
   af->free=&Worker_example::cpu_free;
   af->initialized=true;
   break;
  case CU_TYPE_CUDA:
   ... //similar
  default:
   af->initialized=false; }
```

Listing 1. Example implementation for mandatory worker functions.

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. The example *type_affinity* in Listing 1 defines the GPU to be twice as suitable for the worker as the CPU. Setting an affinity to zero tells the scheduler that no implementation for the specific compute unit exists. The application developer does not have to know the exact performance difference between implementations. The affinity only gives an approximate hint of how much the implementation for one compute unit outperforms the other.

The *getImplementationFor* function fills the functions array *af* with pointers to the implementation for the allocated compute unit type *type*. The worker implementation has to provide the three functions *cu_init*, *cu_main*, and *cu_free* for all supported compute units. While the CPU does not require anything to be copied, all other accelerators usually

```
typedef struct md5_resources {
 std::string hash_to_search;
 unsigned long long currentWordNumber;
 bool +;
} md5_resources_t;
```

Listing 2. Example checkpoint for MD5 cracking.

need the data to be copied explicitly. The *cu_init* function allocates memory on the compute unit and copies needed resources (including the checkpoint) to it. These resources can be 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 after finishing a computation or being denied a re-request and the memory on the compute unit can be freed, which has to be implemented in the *cu_free* function.

In addition to the code framework, the worker class provides system call wrappers to the Scheduler API.

The provided programming model is generic and can be easily used by a scheduler different than the CFS, e.g., by a new scheduler component to be used from user space. Applications that do not use the programming model are simply executed by the usual CFS scheduling mechanisms.

V. 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, which can be used as a checkpoint specifying the progress of the search. Listing 2 shows how checkpoints usually are defined. The actual checkpoint within this struct is the *currentWordNumber*, which saves the current status of the computation. The other information is needed to restore the complete state of the application, i.e., by storing the needed inputs (*hash_to_search*) and the general status (*foundsolution*).

We chose the interval in terms of searched strings between two checkpoints different for the CPU (e. g., 500 strings) and the GPU (e. g., 1 billion) in order to consider the costs for re-transferring the kernel to the compute unit and doing a re-request at a checkpoint that 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 in the test case, hence the meta data is set up to express a clear affinity to the GPU.

The vertical axis in Fig. 3 represents the search space that has been searched, while the horizontal axis denotes the runtime in seconds. We run 15 GPU affine threads, with a limited queue length of 5 for the GPU. As we spawn 15 threads at the same time, 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, the granularity, and the time needed to reach the next checkpoint. 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 enqueued threads are migrated to the GPU and afterwards compute much faster. Fig. 3 shows that cooperative multitasking is possible using the introduced checkpoints. Although the CPU does not lead to great speedups one can see that the heterogeneous system is fully utilized and computing resources are shared among the available tasks.

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 here are defined by the pair of current remainder and potential

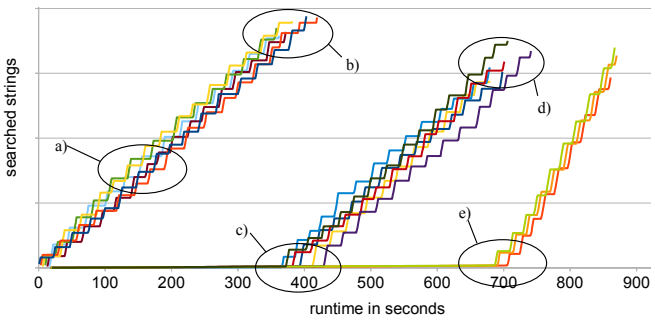

Figure 3. Running 15 independent MD5 instances.

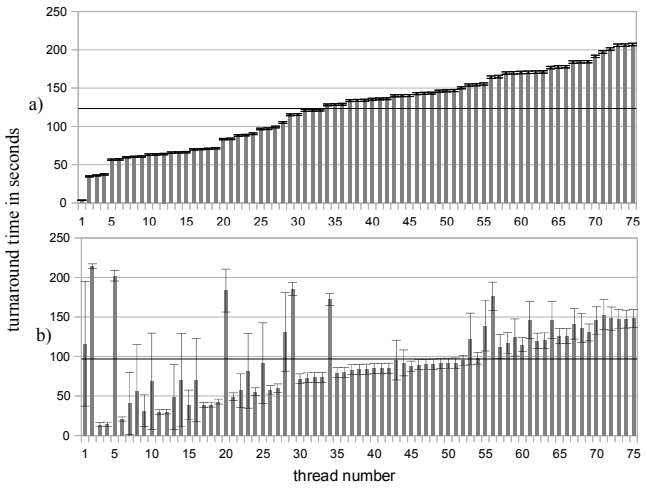

Figure 4. Turnaround times for 75 concurrently started threads without time-sharing (a)) and with time-sharing and 4s granularity (b)) using 25 MD5 and 50 PF threads.

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 1000 divisors apart.

B. Scheduler Evaluation

The following setup was used for the evaluation:

- SMP system with 2-way 4-Core Intel Xeon CPU E5620 @ 2.4GHz, hyperthreading enabled, 12 GB DDR3
- NVIDIA Geforce GTX 480, 480 thread-processors, 1536 MB GDDR5 memory
- Ubuntu Linux 10.04.1 LTS, 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 [4]. The effect of time-sharing on turnaround times can be seen in Fig. 4. 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 started at the same time and scheduled one after another, long running tasks (e.g., tasks 1, 2, 5, 20) block all other tasks, such that the average turnaround time is increased. Using time-sharing, short tasks are not blocked, so that the average turnaround time is lower. Subfigure b) shows that the tasks are not finished in the same order as they are started. Longer jobs do not block the shorter ones, as their time slice is of the equal length as that of a short job. This increases interactivity, as response times are decreased. After 150 seconds only long running threads remain in the system.

In addition to the reduced average turnaround times, the overall performance of several running applications may be increased, if using more than one compute unit. This is shown in Fig. 5, which depicts the total runtime of a varying number of PF applications on the GPU alone and on both

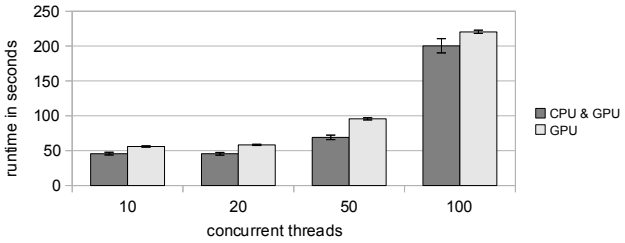

Figure 5. Average runtimes of different counts of PF instances on a GPU and on a combination of GPU and CPU cores.

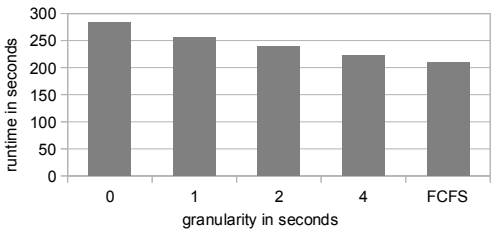

Figure 6. Mean of total runtime for 30 runs with 25 MD5 threads (string length 6) and 50 PF threads on the GPU.

the GPU and the available CPU cores. As can be seen, the average runtime of all instances can be reduced by using the scheduler extension. All threads compete for the use of the GPU, but profit from the fallback computation on the CPU.

2) Overheads: Fig. 6 shows the influence of the granularity of time slices on the runtime of the example applications. All tasks in this test were run on the GPU. Decreasing the granularity raises the total runtime, as task switching overheads are introduced. Introducing time-sharing is therefore a trade off between overheads and interactivity, as a higher granularity decreases the response times of the threads and FCFS scheduling obviously has the smallest overhead.

This is also emphasized in Fig. 7, which shows the average turnaround time depending on the used granularity. Using a granularity of 0 leads to fast-paced task switching and a very high interactivity 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 the fact that long running tasks are blocking short tasks.

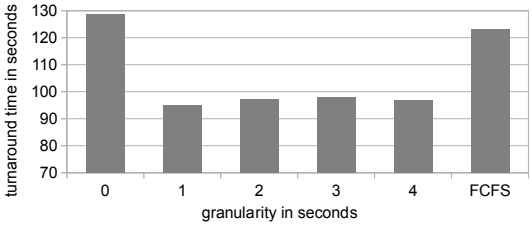

Figure 7. Avg. turnaround times per thread with 25 MD5 threads (string length 6) and 50 PF threads on the GPU.

VI. Related Work

Heterogeneous systems are widely examined in research. Many groups work on the programmability and utilization of these systems (e.g., [5], [6], [7]). OpenCL [8] is a recently well discussed approach that aims on using heterogeneous architectures. Our approach is more general. We do not rely on a complete library, but we provide a simple programming model, allow easy integration of further architectures into our scheduling framework, and allow multi-user scenarios and dynamic runtime scheduling. OpenCL does not support task scheduling. This work addresses OS integration of accelerators by using delegate threads and the problem of reading back their state to the system.

Delegate threads have also, e.g., been used by Bergmann et al. [9], which discuss an approach using "ghost processes" to make reconfigurable System-on-chip (rSoC) hardware processes appear like software processes in a Linux environment. Software ghost processes are associated to hardware processes, used to control these and allow communication with the associated hardware threads. Ghost processes load modules to a FPGA and set up the communication channels. The authors used processes combined with inter process communication (IPC) instead of threads to encapsulate hardware processes. This makes the handling of hardware tasks less lightweight and more complicated for OS integration. Scheduling hardware accelerators has not been discussed.

Lübbers et al. [10] extended the Linux and eCos operating systems by a uniform interface for software threads on the CPU and hardware threads on FPGA accelerators. They extended the multi-threaded programming model to heterogeneous computing resources. Every hardware thread is associated with exactly one software thread, which allows communication between FPGA threads and OS data structures. Cooperative multitasking has been discussed to be possible by storing state information on FPGAs.

Other groups also present work on enabling the OS to read the hardware state. Early work has shown that migrating the state of an application between heterogeneous CPU cores is possible. [11] presents a technique that allows objects and threads to be migrated between machines using heterogeneous nodes at native code level. They introduce so called "bus stops" as machine-independent formats to represent program points. We extend this idea to use time-sharing for current hardware architectures like FPGAs and GPUs. In contrast to GPUs, preemption has been shown to be possible on FPGAs (e.g., [12]). as well as non-preemptive multitasking (e.g., [13]). Nevertheless, none of these approaches has extended the OS scheduler to become responsible for hardware scheduling.

So et al. [14] also modify and extend a standard Linux kernel with a hardware interface. They use a message passing network to provide conventional IPC mechanisms to FPGAs. Communication between hardware and software processes was implemented by FIFOs and mapped to file system-based OS objects. FPGA processes are bound via the Linux */proc* directory and behave similar to software processes. FPGA resources are provided as virtual file system.

Integrating time-sharing using the Linux thread model on heterogeneous systems is a novel approach that increases the interactivity of the system and optimizes the components utilization and the performance of universal applications. None of the previous approaches presents such a global view on the system allowing a holistic scheduling decision by using meta information of the applications as well as incorporating the systems status. This is possible by providing an extension of the operating systems scheduler that can easily be extended to also support other hardware resources.

VII. Conclusions And Future Work

We presented an general model to perform scheduling of tasks on heterogeneous components. We introduced the use of cooperative multitasking 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.

Considering the fact that preemption is not possible on GPUs and reading back the state of accelerators is generally challenging, we introduced a programming model that uses checkpoints to define an unambiguous state of a running application, allowing its suspension and later continuation based on a time-sharing paradigm. This approach is in line with the goal of the current Linux scheduler to provide a fair treatment of available tasks and to increase interactivity.

Our programming model does not yet completely decouple the application development from the use of heterogeneous systems, but relieves the programmer from managing the scheduling of independent threads within the application. Nevertheless, we assume that this decoupling can be achieved by compiler extensions. The Linux kernel handles hardware threads as if they were software threads, which is a continuation of the traditional CPU scheduling approach. We have shown that the automatic migration of threads is possible and that task switching overheads are acceptable.

In future work, we will compare this work to a similar user space scheduling library approach, simplify the usage of our programming model by automatic detection and extraction of checkpoints in applications, and provide example applications incorporating FPGAs.

VIII. Source Code

To promote the uptake of our work by other researchers and users, we made our extended kernel available to the general public as open-source 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] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann, "Cooperative multitasking for heterogeneous accelerators in the linux completely fair scheduler," in *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*. IEEE Computer Society, September 2011.

[3] T. Tamasi, "Evolution of Computer Graphics," in *Proc. NVISION 08*, 2008.

[4] W. Stallings, *Operating Systems: Internals and Design Principles*, 6th ed. Pearson Prentice Hall, 2009.

[5] N. Moore, A. Conti, M. Leeser, and L. King, "Vforce: an extensible framework for reconfigurable supercomputing," *IEEE Computer*, vol. 40, no. 3, pp. 39–49, 2007.

[6] F. Blagojevic, C. Iancu, K. Yelick, M. Curtis-Maury, D. Nikolopoulos, and B. Rose, "Scheduling dynamic parallelism on accelerators," in *Proc. of the 6th ACM conference on Computing frontiers*. ACM, 2009, pp. 161–170.

[7] R. Chamberlain, M. Franklin, E. Tyson, J. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. Shands, and N. Singla, "Auto-Pipe: Streaming Applications on Architecturally Diverse Systems," *IEEE Computer*, vol. 43, no. 3, pp. 42–49, 2010.

[8] A. Munshi, *The OpenCL Specification Version 1.2*, November 2011.

[9] N. Bergmann, J. Williams, J. Han, and Y. Chen, "A process model for hardware modules in reconfigurable system-on-chip," in *Proc. Dynamically Reconfigurable Systems Workshop (DRS) at Int. Conf. on Architecture of Computing Systems (ARCS)*, 2006, pp. 205–214.

[10] 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.

[11] B. Steensgaard and E. Jul, "Object and native code thread mobility among heterogeneous computers (includes sources)," in *Proc. of the 15th ACM Symp. on Operating systems principles*. ACM, 1995, pp. 68–77.

[12] L. Levinson, R. Männer, M. Sessler, and H. Simmler, "Preemptive multitasking on FPGAs," in *Proc. Symp. Field Programmable Custom Computing Machines (FCCM)*. IEEE CS, 2000.

[13] H. Walder and M. Platzner, "Non-preemptive multitasking on FPGAs: Task placement and footprint transform," in *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Architectures (ERSA)*. CSREA, 2002, pp. 24–30.

[14] H. So and R. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," in *Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2006, pp. 1–6.