Subproject C2: On-The-Fly Compute Centers I: Heterogeneous Execution Environments

Tim Hansmeier[1], Tobias Kenter[2], Marius Meyer[2], Heinrich Riebler[2], Marco Platzner[1], Christian Plessl[2]

1 Department of Computer Science, Paderborn University, Paderborn, Germany
2 Paderborn Center for Parallel Computing and Department of Computer Science, Paderborn University, Paderborn, Germany

1 Introduction

Subproject C2 investigated the execution of configured IT services in OTF Compute Centers with heterogeneous computing nodes combining CPUs, GPUs, and FPGAs. The key idea in the subproject is that a service can exist in multiple variants that are specifically tailored for different processor or accelerator architectures. While the execution of these variants leads to the same functional behavior, the non-functional properties, such as energy consumption or latency, may differ considerably. We can exploit this fact by creating variants of services for different hardware architectures at compile time and chosing the optimal variant at runtime according to resource availability to improve performance or efficiency, for example.

In Subproject C2, we have developed methods for this purpose. Specifically, we have studied, how we can create programming models that enable and exploit dynamic dispatching of services (which are themselves part of composed services) to different execution resources; how we can model, optimize, and empirically validate the benefits of dynamic dispatching of services; and how we can develop novel hardware architectures and runtime systems for increasing the effectiveness of this approach.

Over the three funding periods for CRC 901, we have studied the aforementioned idea of dynamic dispatching of services to heterogeneous resources, putting a different emphasis in each of the funding periods, which is summarized in the following.

In the *first funding period*, fundamental architectures and basic mechanisms for heterogeneous migration of services between different computing resources were developed. Heterogeneous migration includes transmodal migration between software and hardware as a special case. We have implemented a system based on POSIX threads with a scheduler for heterogeneous systems and a programming model tuned to it. The approach supports

tim.hansmeier@uni-paderborn.de (Tim Hansmeier), kenter@uni-paderborn.de (Tobias Kenter), marius.meyer@uni-paderborn.de (Marius Meyer), heinrich.riebler@uni-paderborn.de (Heinrich Riebler), platzner@ubp.de (Marco Platzner), christian.plessl@uni-paderborn.de (Christian Plessl)

the implementation and execution of OTF services on different target architectures by a checkpointing mechanism. This makes it possible to interrupt a running service, save its state, and migrate it.

Furthermore, basic concepts for an on-the-fly hardware acceleration approach have been developed. Here, implementations for the different architectures do not have to be created manually but are generated automatically for a limited set of application classes. These methods open up fundamentally new possibilities for dynamically allocating services to the available computing resources and migrating between resources as needed such that application- and system-level goals can be optimized, e.g., throughput, energy consumption, etc. The necessary adaptation of the application to the given programming model and the requirement for architecture-independent checkpoints, however limit the usability and productivity of the approach.

For the *second funding period*, we have therefore chosen programmability and efficiency as the focus topics. By aligning our programming model with OpenCL as a standardized programming interface for heterogeneous computing, the effort required to program heterogeneously migratable services was significantly reduced. Building on this programming model, scheduling algorithms were developed to optimize runtime and energy consumption. Programming for heterogeneous migration is elaborated in Section 2, and a new runtime system able to automatically generate OpenCL accelerator code from sequential CPU code is discussed in Section 5. For scheduling and migration decisions, it is useful to have as precise information as possible about how well individual services are suited for execution on different target architectures. For the necessary off-line characterization of services, we have developed the Ampehre framework, which allows precise measurements of many system parameters of the heterogeneous computing node. The Ampehre measurement framework has been employed to create a highly accurate energy model for task execution on heterogeneous compute nodes [LP18] and was instrumental in developing schedulers utilizing heterogeneous task migration to minimize runtime and energy [LP17], [LP20]. A more detailed description of Ampehre is given in Section 4.

To improve the efficiency of on-the-fly acceleration with FPGAs, overlay architectures were developed that trade off between maximum specialization and full programmability but can be configured much faster. This allows FPGAs to largely avoid the long synthesis and implementation times associated with full specialization. To investigate this approach, hand-designed overlays have been studied and methods for automatically configuring overlays have been developed. This approach is presented in Section 3.

Finally, the focus of the *third funding period* was on mastering the complexity of modern architectures and runtime systems in heterogeneous OTF compute center architectures, where compute nodes must run composed services with varying requirements and optimization goals. The development of increasingly complex runtime systems with centralized scheduling no longer seems promising for such systems. The heterogeneity in the OTF compute centers leads to large amounts of diverse information and optimization goals, which makes designing a centralized scheduler an infeasibly complex task. Instead, we studied concepts of self-aware computing to provide runtime systems with an increased degree of autonomy and learning capability. We experimented with learning classifier systems, in particular XCS, as algorithmic methods for achieving the required learning capability for heterogeneous compute nodes [Han21]. We extended XCS to allow them to adapt their parameters at runtime [HKP20a], which is highly effective in dynamic environ-

ments [HKP20b]. Furthermore, we empirically evaluated different strategies for switching between exploration and exploitation [HP21] and presented an approach for providing safety guarantees [HP22] for XCS. We also came up with an embedded implementation of XCS [HBP22] to evaluate its resource consumption.

We extended our consideration from a single heterogeneous compute node, each with one FPGA, to an entire cluster of such compute nodes. To exploit this kind of infrastructure, models and procedures to partition the composed services onto a multi-FPGA cluster in a meaningful way are required. Therefore, we developed an OpenCL benchmark suite [MKP20] to determine and capture the relevant performance characteristics for the execution of services on such a system, which can serve as a basis for more accurate models for dynamic composition of services in the spirit of the OTF concept. Major aspects were the configurability of the benchmarks [MKP22] and the utilization of direct and highly efficient FPGA-to-FPGA interconnect options in addition to a classical architectural approach in which FPGAs communicate only via CPUs [MKP23]. The benchmarks in the suite are designed to support these various communication approaches and produce comparable performance results for all considered communication infrastructures.

2 Programming for Heterogeneous Migration

Computing nodes are increasingly heterogeneous and augment CPUs with accelerator technologies such as GPUs and FPGAs. To benefit from such a computing environment, developers must identify hotspots or tasks in their applications, port those to the available accelerators, and finally optimize them to achieve high performance. Additionally, scheduling techniques are needed that distribute the workload of one or several applications to the heterogeneous resources, subject to an optimization objective such as minimization of runtime or energy consumption.

The introduction of OpenCL as a programming language greatly simplified the use of accelerator technologies. With OpenCL compilers available for CPUs, GPUs, and even FPGAs, application code is basically executable on all these resources without any additional porting effort. However, since OpenCL is not performance-portable, developers must still optimize their task implementations to achieve good performance for the different resources.

Current accelerators rely on a run-to-completion execution model, where a task assigned to an accelerator computes there until termination. This is in strong contrast to CPU-based computing, where operating systems provide preemptive multitasking, and might severely impact system performance since a running task cannot be migrated to a better-suited resource in a later execution phase.

In this section, we give an overview of our novel OpenCL-based programming framework that overcomes the limitations of the run-to-completion approach. We introduce a programming pattern and execution model for tasks that allow us to migrate them between the resources of a heterogeneous compute node at predefined states without losing their computational progress. While we focus on OpenCL, the approach is more general and supports programming languages with host-centric execution models, also including OpenMP, OpenACC, CUDA, and the Maxeler MaxJ hardware description language. Developers only need to provide functionally identical task implementations for the resources. Furthermore,

our work includes an interface for inter-process communication between the tasks and a scheduler framework. Using this interface, schedulers can decide and execute task-resource assignment, including heterogeneous migration.

Applications following a host-centric programming style are bipartite: The *host code* is responsible for resource management, which includes allocating local memory on the accelerator, transferring data to and from the accelerator, and triggering computations. The computations on the accelerator are denoted as *kernel code*. The CPU plays a special role since it is the host and can at the same time also execute kernel code.

In our approach, we store relevant task state information in memory and transfer this information between the host memory and an accelerator's local memory. Through such state transfers, migration can be implemented even between very diverse resources such as FPGAs and GPUs. This technique is often referred to as *checkpointing* and poses two challenges: First, task developers or automated tools need to identify checkpoints in application tasks that can be mapped to other resources in order to continue the task computation without loss in their computational progress. Moreover, minimal task states are favorable since state transfers are expensive and constitute overhead. Second, the checkpointing frequency must be carefully selected to balance between the overhead incurred by checkpointing and the ability of being able to quickly migrate when needed.

A possible method to enable task migration by checkpointing is adapting the *loop strip mining* transformation to a task's kernel code. The loop of a data-parallel kernel is split into an outer and an inner loop, which is vectorizable. The outer loop is then run as host code, and the adjusted kernel comprising the inner loop is called from the host. This way, the adjusted kernel works on blocks of data successively and after each kernel execution, i.e., an iteration of the outer loop, the checkpoint can be transferred. Since the inner loop is kept in vectorized form, a checkpointed task implementation can provide high performance for data-parallel tasks if the *checkpoint distance* is sufficiently large.

Our programming pattern for heterogeneous task migration supports checkpointing and comprises five stages:

1. The *bookkeeping* stage is the task preparation stage, where we allocate memory space in the host memory and read input data from the hard disk or the network interface. This stage is resource-independent and therefore involves only activities handled by the CPU.
2. The *init* stage allocates memory space in the local memory of the accelerator and transfers the checkpoint to this memory.
3. The *compute* stage executes the kernel code. Each time the compute stage is called, the kernel processes the next block of data and stores its progress as updated checkpoint. Furthermore, the kernel must be able to report its computational progress to enable the host to keep track of the overall task computation.
4. The *fini* stage is the counterpart of the init stage and transfers the checkpoint back into host memory before releasing the accelerator device.
5. The *cleaning* stage is the final one and the counterpart of the bookkeeping stage, as it writes the computation results to the hard disk or the network interface. This stage is resource-independent and thus exclusively executed on the host processor.

Based on this pattern, a task is migrated from resource A to resource B by calling the fini stage for the task implementation running on resource A followed by executing the init stage of the task implementation for resource B. The resulting *migration overhead* comprises two parts: The first and frequently dominating part is the time to transfer the checkpoints during the fini and init stages. The second part includes additional steps for preparing the target resource, such as the reconfiguration of an FPGA device.

We explain the *lifecycle* of a migratable task using the example shown in Figure 39, which lists the pseudo code for the four major software components involved. `main.cpp` instantiates `ExampleOCL` configured for CPU usage and resource-specific implementations for GPU and FPGA. `ExampleOCL` itself loads the checkpointed OpenCL kernel `example` and compiles it for execution on the CPU. The kernel illustrates checkpointing by iterating over successive sections of a strip-mined loop, with `iters_per_checkpoint` specifying the size of data processes per kernel execution. The task's progress can be determined by comparing the `progress` counter with the `num_of_checkpoints`. Note the three resource-specific stages implemented in `ExampleOCL`. The init and fini stages are transferring the checkpoint between the host memory and the local memory of the device. Since host memory and CPU-related memory are identical, the checkpoint is not copied. The `compute` method is then working on the checkpoint by only reading and writing data in the local memory. After adding all resource-specific implementations of the task in `main.cpp`, the task executor `TaskExec` is called. The pseudo code in Figure 39 also illustrates the execution of `execute_online()`, interacting with a scheduler connected via Inter-Process Communication (IPC).

The code listed in Figure 39 correlates to the task lifecycle depicted as a flowchart in Figure 40. The dotted shapes clarify the mapping between the pseudo code and the flowchart. The first activity is calling the `bookkeeping` method, which is executed by the host processor and prepares the task for execution. Then, the task enters an execution loop where it remains until the entire data is processed, i.e., the task execution state is `FINISHED`. The first step in the execution loop is to wait for resource assignment, which is implemented as a blocking IPC receive call that returns the assigned resource from the scheduler. Next, the `init` method of the chosen task implementation is called and the current checkpoint is copied into the target local memory. The following do-while loop iteratively calls `compute` and informs the scheduler about the task execution progress. While `compute` actually executes resource-specific kernels on the CPU or accelerators, denoted by dark blue, red, and green colored box fillings, the `init` and `fini` box fillings are kept in light colors to depict that the devices are active by copying checkpoint data or reconfiguring the FPGA.

After the task execution progress has been sent to the scheduler, the task execution state is checked. In case the task execution has finished, we exit the do-while loop and call `fini` for the current task implementation, release the resource, and finally execute `cleaning`. If the task execution has not been finished yet, we communicate with the scheduler to figure out whether a task must be migrated. If so, the fini stage is called, the resource is released, and a new resource assignment is requested in the next iteration of the execution loop. After a new resource has been assigned, the task again calls the `init` method for the new resource.

The programming framework has been implemented in C++ and allows an easy integration of new computing resources by overriding corresponding class methods. Based on the

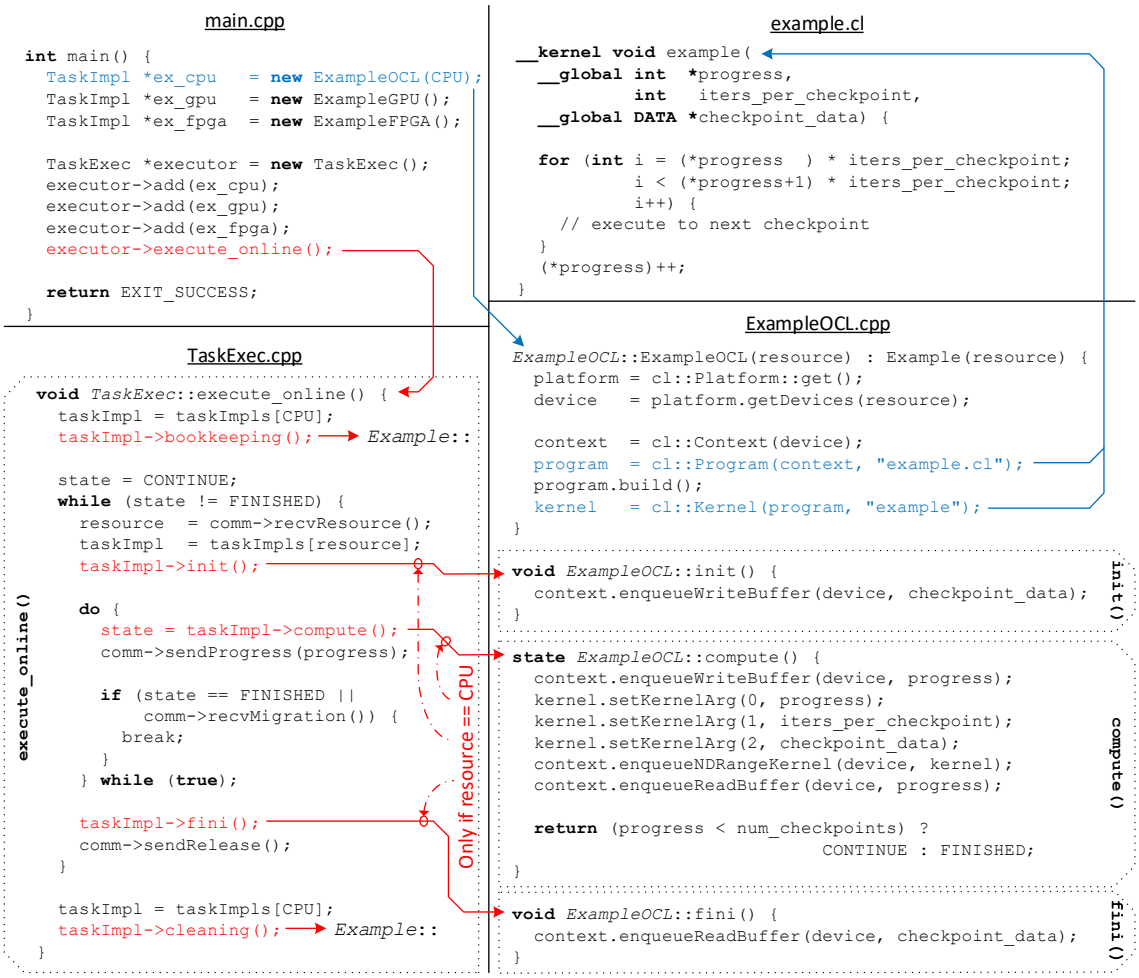

Figure 39: *Major software components for a migratable task.*

programming framework, two schedulers for heterogeneous compute nodes have been realized that demonstrate the potential of heterogeneous migration in terms of runtime and energy minimization. In [LP17], the reMinMin scheduler has been presented based on a static list scheduling approach for energy minimization. In [LP20], MigHEFT focused on scheduling migratable task graphs to heterogeneous resources.

3 Analyzing FPGA Overlays as Target for OTF Hardware Accelerator Generation

Overlays are configurations for FPGAs that are not fixed to a specific task, but instead provide a limited form of programmability, more abstract than that of the underlying FPGA fabric. Compared to highly optimized application-specific libraries, overlays enable significantly more applications as candidates for FPGA acceleration in an OTF context, because overlays can be more broadly applied. They thus provide a purchasing argument for FPGAs for OTF compute centers that need to aim for good utilization of their hardware over time. Also, in comparison to the synthesis of FPGA designs from high-level language code, where OpenCL and recently SYCL are particularly promising as a description language, overlays can help to avoid the extremely long synthesis runtimes of several hours

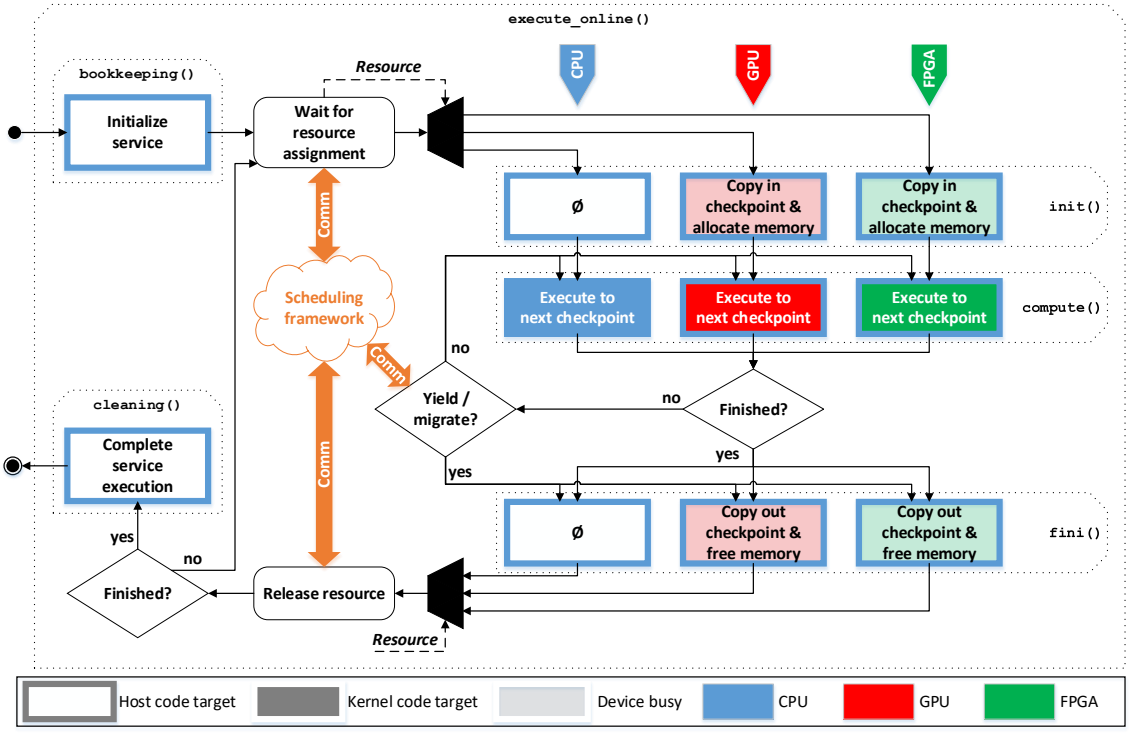

Figure 40: *Lifecycle of a migratable task.*

up to days, which are typical for FPGAs. Paired with suitable compilation approaches, they can also reduce the demand for manual development or optimization ahead of OTF deployment.

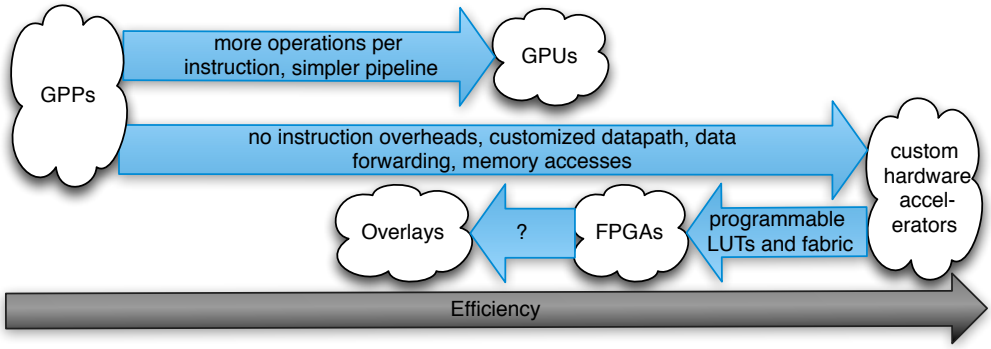

Figure 41: *Qualitative illustration on the impact of architecture features of accelerators on efficiency. Quantifying the performance overheads of FPGA overlays was the central research question of this contribution.*

We distinguish between processor-like instruction-programmable overlays and structurally programmable or configurable overlays. For both approaches, diverse architectures have been presented that gain their efficiency from various combinations of parallelism, pipelining, and targeted data access and reuse. These are already being investigated in the academic environment from various aspects such as productivity, portability, and scalability. For instruction-programmable overlays, it was however largely unclear until our work how close the performance of such architectures comes to that of fully specialized FPGA

implementations. Figure 41 illustrates the context of this research question in comparison to alternative accelerator architectures.

To answer this question, we have implemented a diverse set of computational tasks with identical interfaces on an overlay-based FPGA system and with highly specialized FPGA designs. The tasks here are all runtime-intensive steps (often referred to as kernels) from an application to compute stereo correspondence, which in turn is the most important and costly intermediate step to compute depth from a pair of stereo images. By following the best quality published algorithm [MSZ+11] in this area until recently, we achieve, on one hand, a comparison between overlay and specialized kernels that is not biased by FPGA-specific optimizations on the algorithm level and, on the other hand, the most accurate stereo matching implementation with FPGA acceleration to this time. In contrast, other FPGA implementations in this area (e.g. [SHW+14; JM14; TLLA14]) adapt the work steps and their sequence to the target architecture to varying degrees, thus achieving higher performance or lower space requirements on the FPGA with reduced result quality.

Due to the availability of suitable development tools and runtime environments to effectively implement the respective approaches with overlay and specialized kernels, the two approaches were implemented on two different target platforms: The Convey HC-1 with an instruction-programmable FPGA overlay with vector architecture on one hand and the Maxeler MPC-X platform with its own description language for highly specialized dataflow kernels on the other. Both represent state-of-the-art systems with a high-performance server processor and FPGA accelerator at the respective time of acquisition.

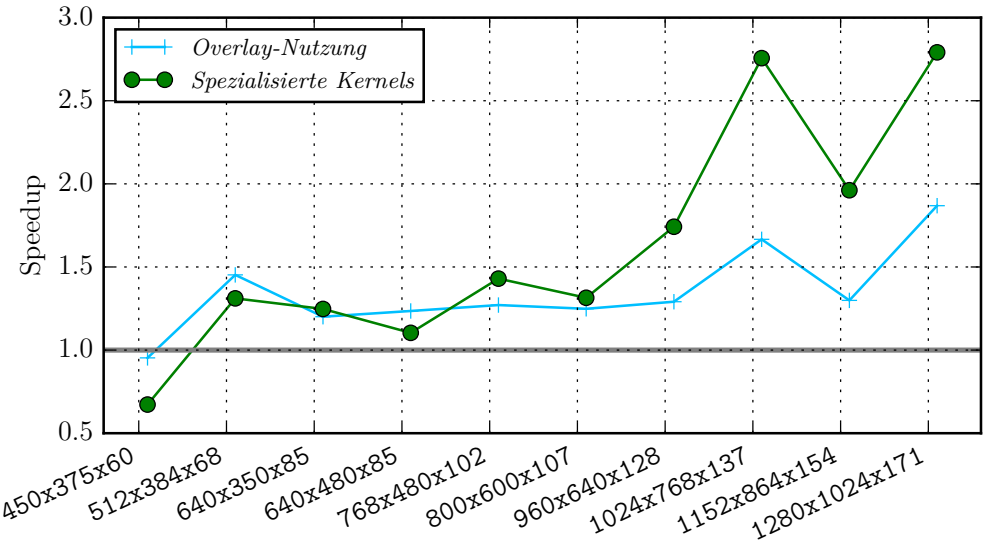

Figure 42: *Speedups of fully integrated stereo matching implementations on two systems with FPGA accelerators. These measurements include overheads for reconfiguration and data transfers, which favors the overlay architecture for small problem sizes in comparison to the specialized kernels.*

On both target platforms, the fully integrated computation of the stereo correspondence is executed by offloading the runtime-intensive kernels to the respective FPGA accelerator. Preparatory and management steps remain on the respective main processor. The runtimes for data transfers, synchronization and, in the case of the specialized kernel designs, recon-

figuration included in this execution model limit the achievable performance. Nevertheless, illustrated in Figure 42, both accelerator platforms achieve performance advantages over the powerful main processor of the Maxeler MPC-X platform for most input sizes of the application.

To quantify the conceptually driven performance differences between using an instruction-programmable overlay and fully specialized FPGA designs, we had to factor out influences of the specific platforms and their runtime environments. In [Ken16] and [KSP15], these steps are explained in detail. Subsequently, it can be shown that using the overlay yields on average about a factor of 3 less isolated kernel performance than FPGA implementations fully adapted to the task. In return, for the overlay, the runtimes of the tools used for translation or synthesis are several orders of magnitude shorter, amounting to only seconds instead of hours or even days. Overall, the high productivity required to profit from FPGA acceleration in the OTF context is not achieved by tool runtimes alone, but also depends on programming patterns (see also Section 2) or automated tools (see also Section 5) for code generation or overlay configuration generation and offloading to accelerators.

Thus, in the end, the decision between fully specialized FPGA kernels and overlay usage is similar to the decision between ASICs and FPGAs: Given sufficient development time, budget, expertise, and given a sufficiently high application demand, it will typically pay off to fully specialize. However, if any of these preconditions is not met—as can often be the case in OTF scenarios—overlays can provide an interesting alternative, at a performance cost that we now understand better.

4 AMPEHRE: An Extensible Measurement Framework for Heterogeneous Compute Nodes

Application performance profiling is a major step in software development. Based on hardware performance counters provided by the target devices and on timing information, developers gain knowledge about runtime behavior in terms of metrics such as the number of executed instructions, cache misses, page-faults, or statistics about called functions. Understanding runtime behavior is instrumental for optimizing performance. Examples for widely-used performance analysis tools are the open-source tools `Perf`, `IgProf`, and `Likwid`, and the vendor-specific tools Intel VTune Amplifier or the Nvidia GPU development IDE Nvidia Nsight and command-line tool `nvprof`. With the introduction of the Running Average Power Limit (RAPL) interface for Intel CPUs, developers are also able to perform energy measurements on CPUs.

A shortcoming of most existing tools is their lack of an easy-to-use and extensible application programmer interface (API) that allows user applications to read performance and energy data comparable across different resource types. The Performance Application Programming Interface (PAPI) project has been developed to help solve this issue. Particularly with the PAPI version 5 release, developers are able to add capabilities for power or temperature analysis by implementing so-called *PAPI components*, extending PAPI to new platforms and other sensor types. PAPI provides a unified API that hides the underlying device-specific measuring procedures when reading power, energy, and temperature sensors. But, even with PAPI, the retrieved data must be interpreted to gain semantically comparable measurements results across resource boundaries.

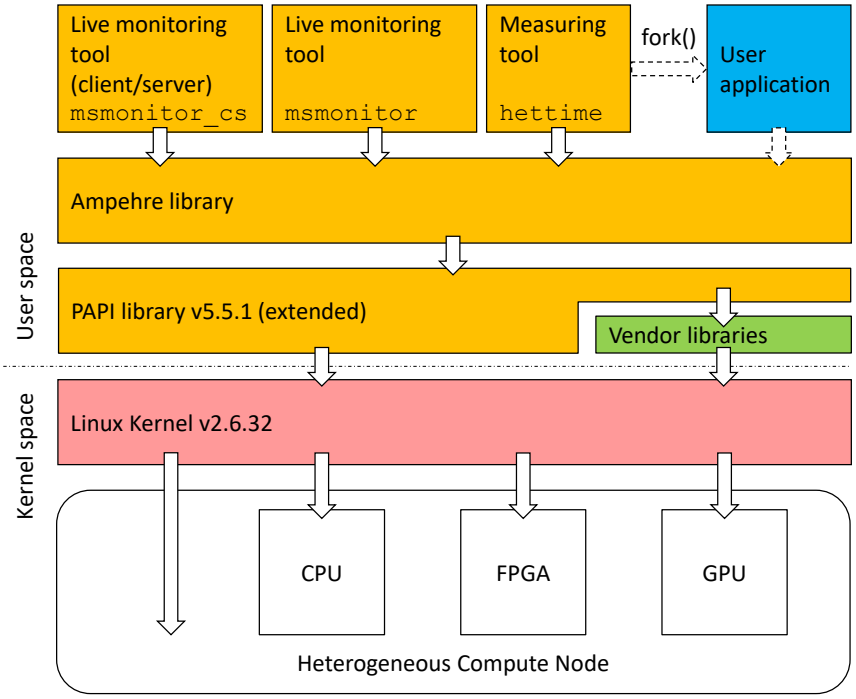

Figure 43: *Ampehre architecture (taken from [LWP18]). Blocks in orange denote components we have implemented or extended.*

To improve on this situation, we have developed the measurement framework *Ampehre*, short for *Accurately Measuring Power and Energy for Heterogeneous Resource Environments* [LKEW]. Ampehre is designed for heterogeneous high-performance compute nodes running Linux and (i) allows an easy integration into applications by providing a clear API covering all resource types, (ii) is extensible to new resources and sensors through the use of PAPI, and (iii) is available as open source.

Figure 43 presents the architecture of the Ampehre framework, which comprises three layers in user space: an extended PAPI library, the Ampehre library, and the Ampehre tools. We base the Ampehre framework on *PAPI*, which makes it inherently portable to other systems running a Linux OS distribution, and we have extended the *PAPI library* to support not only CPU and system-wide sensors but also to retrieve performance data gathered at the accelerator components GPU and FPGA.

Figure 44 denotes the main PAPI components with their interfaces utilized by Ampehre to obtain measurements from the heterogeneous computing resources and the main board of our server node: The PAPI component `rapl` supports CPU measurements, including the cores, last-level cache, memory controller and DRAMs. Modern Intel CPUs provide several so-called Model Specific Registers (MSR) to retrieve data related to energy consumption, temperature, etc. The PAPI component `ipmi` is necessary to retrieve system-wide measurements such as the system-wide power dissipation measured at the power supply. For this, the component communicates with the *Baseboard Management Controller* (BMC) by means of the Linux OpenIPMI library. IMPI is a standard to unify server platform management. The *Nvidia GPU* is supported if PAPI is compiled with the `nvml` component. This component includes the Nvidia Management Library (NVML), which is used to obtain the current power dissipation and die temperature. Finally, Am-

pehre is enabled to gather measurement data on the *Maxeler Vectis* by linking against the MaxelerOS library if the `maxeler` component is enabled in PAPI. From the overall four described PAPI components, we have implemented `maxeler` and `ipmi` from scratch and extended `rapl` and `nvml` in order to support the sensors of interest on our heterogeneous compute node. The node employs a *Dell PowerEdge T620* with two *Intel Xeon E5-2609 v2* CPUs as host processors running CentOS 6.8 Linux with kernel v2.6.32, a PCIe-connected *Nvidia Tesla K20c* GPGPU based on the Kepler microarchitecture, and a PCIe-connected *Maxeler Vectis* FPGA board based on Xilinx Virtex 6 (xc6vsx475t).

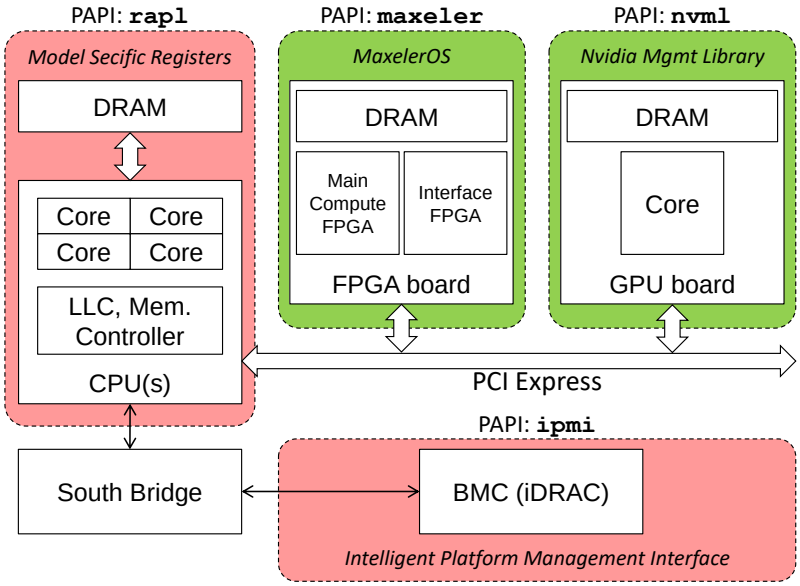

Figure 44: *PAPI components required to retrieve energy and temperature measurements (taken from [LWP18]). We use Linux OS kernel interfaces to sample CPU and BMC sensors (red blocks), and vendor libraries to retrieve measurements from the FPGA and GPU boards (green blocks).*

The *Ampehre library* extends PAPI functionality with the goal to hide all computations and data interpretations from the application developer. The Ampehre library unifies the meaning of gained data across resource boundaries and provides the developer with a set of functions having the same semantics for all resource types. Table 1 gives an overview of the metrics that can be reported by the Ampehre framework for each of the four PAPI components. The measured energy is by definition a value accumulated over the measurement period. For the other quantities, which are power, temperature, utilization, frequency, and amount of allocated memory, Ampehre reports the current (latest) value and the minimum, maximum, and average over the measurement period.

Developers can instantiate the Ampehre library in their applications to use our measurement framework, or they can use one of the following *Ampehre tools*:

`hettime` extends the well-known Linux utility *time* by reporting comprehensive measurements for an executed binary, i.e., also the energy consumed by the overall system, the average power dissipation and maximum temperature for each component, etc. The results can be stored in JSON files, CSV tables, or simply printed to the shell. `hettime` is highly configurable through command line parameters.

Component	Energy	Power	Temp.	Utilization	Frequency	Alloc. Memory
	Accumulated	*Current, Minimum, Average, Maximum*				
`rapl`	✓	✓	✓	✓	✓	✓
`nvml`	✓	✓	✓	✓	✓	✓
`maxeler`	✓	✓	✓	✓	✗	✗
`ipmi`	✓	✓	✓	✗	✗	✗

Table 1: *Quantities that can be measured or computed with Ampehre (taken from [LWP18]).*

`msmonitor` is a Qt-based live monitoring tool plotting the most recent measurements. `msmonitor` can display the measurement data in the form of an array of curves or as heat maps. These features are exemplary illustrated in Figure 45. The screenshot displays data taken while an arbitrary set of 15 tasks is concurrently executed on CPU, GPU, and FPGA. The array of curves on the left side of Figure 45 represent the current power dissipation of the three computing resources, while the heat maps on the right side of Figure 45 show device utilizations.

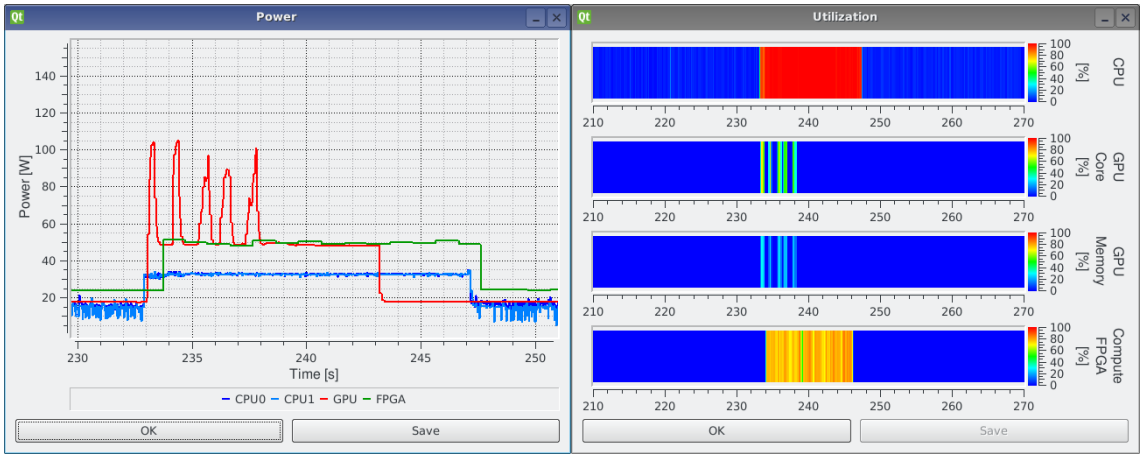

Figure 45: *Power dissipation and utilization plotted by* `msmonitor` *while an arbitrary set of 15 tasks are executed on CPU, GPU, and FPGA (taken from [LWP18]).*

`msmonitor_cs` is a server-client implementation of `msmonitor` for reducing probing effects on the measured server by transferring the GUI rendering to a client connected via TCP/IP.

5 Transparent Acceleration for Heterogeneous Platforms with Compilation to OpenCL

Hardware accelerators, such as GPUs or FPGAs, can offer exceptional performance and energy advantages compared to CPU-only systems. Services that use accelerators are especially interesting in an on-the-fly scenario because they offer higher degrees of freedom in the configuration process, can result in different quality of service aspects,

and finally improve the overall execution. Service providers, however, need to spend considerable efforts on application acceleration without knowing how sustainable the employed programming models, languages and tools are. To tackle this challenge, we developed and demonstrated a new runtime system called HTroP [RVKP19; RVKP18] that is able to automatically generate and execute parallel accelerator code (OpenCL) from sequential CPU code. HTroP transforms suitable data-parallel loops into independent OpenCL-typical work items and offloads the execution of these work items to the hardware accelerators through a mix of library components and application-specific OpenCL host code. Computational hotspots that are likely to profit from parallelization are identified and can be offloaded to different resources (CPU, GPGPU and Xeon Phi) at runtime. We demonstrated the potential of HTroP on a broad set of applications and are able to improve performance and energy efficiency.

OpenCL provides an open standard interface for parallel computing using task- and data-based parallelism, which can be executed across different devices. This means that by generating OpenCL kernel code (once), one can target multiple accelerators. OpenCL, not only poses the challenge of extracting hotspots into kernels and optimizing them for the target accelerator architecture but also involves many tedious adjustments to the remaining host code. Given these challenges, there is a considerable gap between the architectural potential of highly heterogeneous multi-accelerator architectures and their actual adoption and utilization that we aim to overcome with HTroP.

Our approach builds upon and integrates results from different open-source projects: We consider LLVM bitcode as the input format to HTroP, on which all optimization, transformation and acceleration steps are performed. The detection of data-parallel loops is based on LLVM's Polly project [GH16]. Polly uses an abstract mathematical description to detect and model static control flow regions (so-called SCoPs). And finally, we use LLVM's Axtor backend [Mol11] to translate LLVM bitcode into OpenCL kernel code. In our own previous work [DRVP15], we used OpenMP and vectorization to offload hotspots from a low-power client to a remote server with an Intel Xeon PHI accelerator. Related work has researched SCoP-based hotspot detection and acceleration but with other programming models and fewer and different devices in the backend. With Polly-ACC, Grosser et al. [GH16] target Nvidia GPUs using CUDA calls from the host CPU and a PTX backend. Compared to our approach, LLVM bitcode can be generated for a wide range of applications without requiring the source code to be available. Additionally, by using OpenCL as kernel code, various services can be generated on-the-fly, targeting a range of accelerators.

Figure 46 gives an overview of our approach. Our tool flow receives the legacy application in LLVM bitcode and detects computational hotspots as SCoPs. These get parallelized and offloaded using three subsequent optimization passes. In the first transformation step, the *Work-item Parallelizer* uses the dependence analysis information (from Polly) to determine how a loop can be transformed to expose parallelism suitable for OpenCL. For example, Listing 10.1 shows a simple 2D convolution in pseudo code. The outer two **for** loops (line 2 and 4) iterate over the entire input `in`. The inner two **for** loops (line 7-8) perform the convolution for each entry. The dependence info reveals that the innermost loops are data dependent. Hence, only the outer two loops are parallelized.

```
1 heavyConv2D(int *in, int *out, int rows, int cols) {
2   for (int r = 0; r < rows; r++) {
3   // AFTER Work-item Parallelizer for-loop replaced by: int r = get_global_id(0);
```

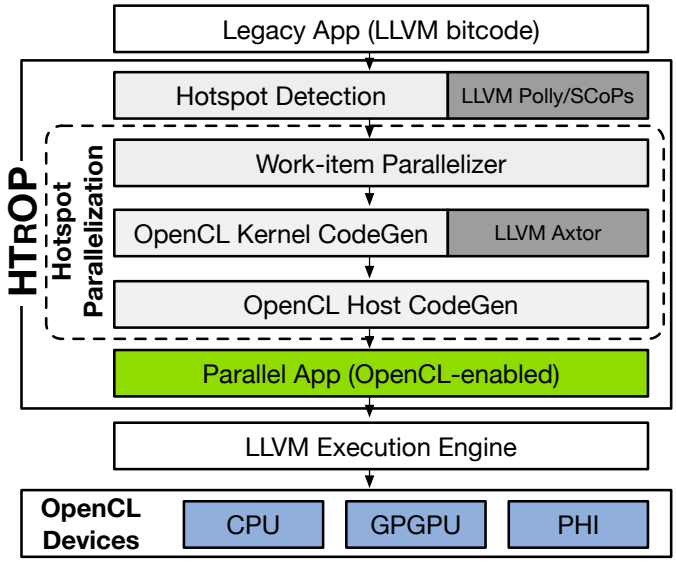

Figure 46: *Architecture of the runtime system. The sequential application is analyzed and parallel OpenCL code is generated on-the-fly.*

```
4     for (int c = 0; c < cols; c++) {
5     // AFTER Work-item Parallelizer for-loop replaced by: int c = get_global_id(1);
6       int |sum| = 0;
7       for (int i = 0; i < 5; i++) {
8         for (int j = 0; j < 5; j++) {
9           // ...
10          |sum| += in[r + i][c + j] * COEFFS[i][j];
11      // ...
12      out[r][c] = |sum|;
```

Listing 10.1: *Nested loops performing a 2D convolution. The two highlighted lines show the modifications performed by the* Work-item Parallelizer *to expose parallelism.*

The following steps are performed to expose work-item parallelism in each loop that has no dependencies:

1. Determine the loop induction variable.
2. Remove the loop control flow.
3. Replace the induction variable with a call to the `get_global_id` OpenCL API call.

The induction variable of a loop represents the variable that is incremented/decremented for each iteration (e.g., `r` and `c` in Listing 10.1). The induction variable can be obtained from the loop header. Once the induction variable is found, we find the corresponding compare instruction that checks the loop exit condition. The compare and branch instructions associated with the loop control flow are removed. This effectively removes the loop structure with all the code previously inside the loop being executed exactly once. The final step is to replace the induction variable with a call to the `get_global_id` OpenCL API call. The lines without line numbers in Listing 10.1 replace Lines 2 and 4 (with Lines 3 and 5) after the *Work-item Parallelizer* is done.

In the second optimization pass, this modified LLVM bitcode is fed into the Axtor-based *OpenCL Kernel CodeGen* to produce corresponding OpenCL kernel code. Since the legacy

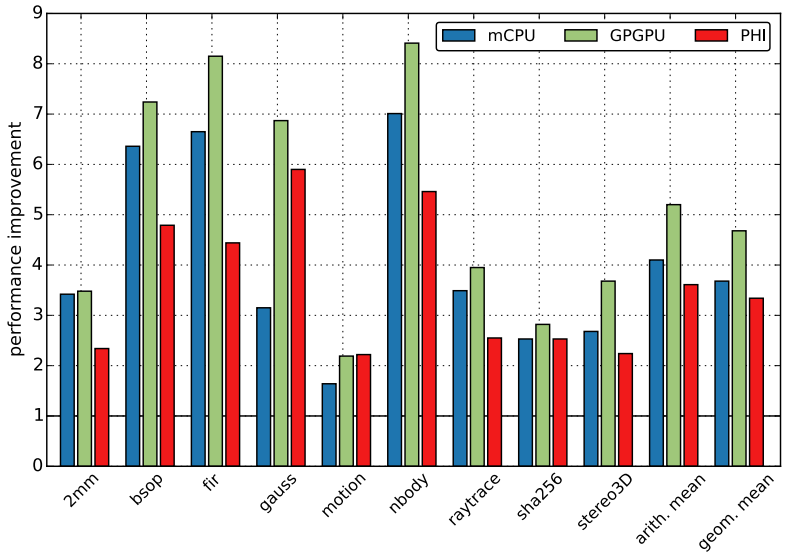

Figure 47: *Performance (speedup) of our runtime system (including all overheads) compared to the normalized CPU baseline (= 1).*

application does not originally support OpenCL, the *OpenCL Host CodeGen* updates the application to support all the devices along with the corresponding OpenCL host code to invoke the kernel. We have implemented a wrapper library that creates and exposes device handles for all appropriate OpenCL devices of our evaluation platform to the global scope of the application. The result is an OpenCL-enabled parallel application that is executed through the LLVM Execution Engine and can offload hotspots to the appropriate OpenCL device.

In order to evaluate our approach, we used the multi-accelerator that we have described in Section 4 and Figure 43. We use a set of benchmark applications extracted from scientific computing, financial, signal- and image processing, and security domains. The baseline is single-threaded CPU code compiled with `gcc` v4.8.2 using the highest optimization level `-O3`. The performance evaluation in Figure 47 reveals speedups for all measured applications with considerable differences between applications and with visible, but small differences among the target devices.

OpenCL turned out to be an effective vehicle for targeting multiple architectures, allowing us to generate the mechanical parts of the host code and to use the same parallelism pattern for the transformation of computationally intensive regions of the application into accelerator code. Service providers can use HTroP in order to generate different variants of services or optimize the execution of services on-the-fly.

6 Conclusion and Outlook

Over the three funding periods of CRC 901, the topic of the use of heterogeneous computing resources in data centers has developed strongly not only in research but especially in practical, economic applications. The ongoing shift of computation from end-user devices to cloud data centers opens up cloud resource providers to leverage heterogeneous compute

resources and benefit from their advantages while keeping the programming interfaces unchanged for service users. This enables faster technological innovation at the hardware level without requiring radical changes in programming models and tooflows on the user side. While the tools presented in this chapter have not been directly taken up on a large scale, the concepts and methods have certainly found their way into practice. For example, FPGA-based overlay architectures are used in Microsoft Bing to implement scoring methods on search results. Heterogeneous programming models with support for CPUs, GPUs and FPGAs as well as runtime systems for the dynamic allocation of tasks to resources are also in widespread use today, e.g. in the SYCL standard which is the basis for Intel's development environments under the name oneAPI.

Last but not least, the extensive experience with FPGA accelerators, programming models and runtime systems has also been incorporated into the design of the FPGA partition of the Noctua 1 and Noctua 2 supercomputers at the Paderborn Center for Parallel Computing. A unique platform has been created that provides a stable production environment for the use of FPGAs in HPC and data-center applications. At the same time, the partition is an ideal testbed for testing communication mechanisms in multi-FPGA applications due to a worldwide unique architecture with an optical L1 network switch, which was developed at the CRC. Thus, a basis for the continuation of this research line exists far beyond the end of CRC 901.

We are pleased to note that the topic examined in CRC 901 has not become stale, even after 12 years of funding. Quite the contrary: although the advantages of highly specialized domain-specific architectures are generally recognized, no other architectures have yet been able to establish themselves apart from GPUs in the data center. One reason for this is certainly that generating efficient code for specialized architectures from abstract specifications remains a major challenge. To have the potential of Domain-Specific Computing, therefore, new approaches are necessary. We also see a high potential for research with impact for methods that globally optimize the operation of a data center, acting across layers. The current approach of increasingly dynamic but still local optimizations of the operating state does not lead to globally optimal operating states. Especially in times of increased volatility of energy price, energy availability, and load from user requirements, feasible methods are needed to cope with the complexity of systems and requirements.

Bibliography

[DRVP15] Damschen, M.; Riebler, H.; Vaz, G.; Plessl, C.: Transparent offloading of computational hotspots from binary code to Xeon Phi. In: *Proc. Design, Automation and Test in Europe Conf. (DATE)*. EDA Consortium, Mar. 2015, pp. 1078–1083

[GH16] Grosser, T.; Hoefler, T.: Polly-ACC Transparent compilation to heterogeneous hardware. In: *Proceedings of the 2016 International Conference on Supercomputing*. ACM. 2016, p. 1

[Han21] Hansmeier, T.: Self-aware Operation of Heterogeneous Compute Nodes using the Learning Classifier System XCS. In: *HEART '21: Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*. Association for Computing Machinery (ACM), 2021

[HBP22] Hansmeier, T.; Brede, M.; Platzner, M.: XCS on Embedded Systems: An Analysis of Execution Profiles and Accelerated Classifier Deletion. In: *GECCO '22: Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery (ACM), 2022, pp. 2071–2079

[HKP20a] Hansmeier, T.; Kaufmann, P.; Platzner, M.: An Adaption Mechanism for the Error Threshold of XCSF. In: *GECCO '20: Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery (ACM), 2020, pp. 1756–1764

[HKP20b] Hansmeier, T.; Kaufmann, P.; Platzner, M.: Enabling XCSF to Cope with Dynamic Environments via an Adaptive Error Threshold. In: *GECCO '20: Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery (ACM), 2020, pp. 125–126

[HP21] Hansmeier, T.; Platzner, M.: An Experimental Comparison of Explore/Exploit Strategies for the Learning Classifier System XCS. In: *GECCO '21: Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery (ACM), 2021, pp. 1639–1647

[HP22] Hansmeier, T.; Platzner, M.: Integrating Safety Guarantees into the Learning Classifier System XCS. In: *Applications of Evolutionary Computation, EvoApplications 2022, Proceedings*. Vol. 13224. Lecture Notes in Computer Science. Springer International Publishing, 2022, pp. 386–401

[JM14] Jin, M.; Maruyama, T.: Fast and Accurate Stereo Vision System on FPGA. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7 (Feb. 2014), no. 1, 3:1–3:24.

[Ken16] Kenter, T.: Reconfigurable Accelerators in the World of General-Purpose Computing. PhD thesis. Paderborn University, 2016.

[KSP15] Kenter, T.; Schmitz, H.; Plessl, C.: Exploring Trade-Offs between Specialized Dataflow Kernels and a Reusable Overlay in a Stereo Matching Case Study. In: *Int. Journal of Reconfigurable Computing (IJRC)* (2015), pp. 1–24

[LKEW] Lösch, A.; Knorr, C.; El-Ali, A.; Wiens, A.: *Ampehre: Accurately Measuring Power and Energy for Heterogeneous Resource Environments*. http://ampehre.uni-paderborn.de/. Last accessed on Jan 3, 2023

[LP17] Lösch, A.; Platzner, M.: reMinMin: A Novel Static Energy-Centric List Scheduling Approach Based on Real Measurements. In: *Proceedings of the 28th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2017

[LP18] Lösch, A.; Platzner, M.: A Highly Accurate Energy Model for Task Execution on Heterogeneous Compute Nodes. In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018

[LP20] Lösch, A.; Platzner, M.: MigHEFT: DAG-based Scheduling of Migratable Tasks on Heterogeneous Compute Nodes. In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020

[LWP18] Lösch, A.; Wiens, A.; Platzner, M.: Ampehre: An Open Source Measurement Framework for Heterogeneous Compute Nodes. In: *Proceedings of the International Conference on Architecture of Computing Systems (ARCS)*. Vol. 10793. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 73–84

[MKP20] Meyer, M.; Kenter, T.; Plessl, C.: Evaluating FPGA Accelerator Performance with a Parameterized OpenCL Adaptation of Selected Benchmarks of the HPCChallenge Benchmark Suite. In: *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE. 2020, pp. 10–18

[MKP22] Meyer, M.; Kenter, T.; Plessl, C.: In-depth FPGA accelerator performance evaluation with single node benchmarks from the HPC challenge benchmark suite for Intel and Xilinx FPGAs using OpenCL. In: *Journal of Parallel and Distributed Computing* 160 (2022), pp. 79–89.

[MKP23] Meyer, M.; Kenter, T.; Plessl, C.: Multi-FPGA Designs and Scaling of HPC Challenge Benchmarks via MPI and Circuit-Switched Inter-FPGA Networks. In: (2023).

[Mol11] Moll, S.: Decompilation of LLVM IR. In: *Master's thesis* (2011)

[MSZ+11] Mei, X.; Sun, X.; Zhou, M.; Jiao, S.; Wang, H.; Zhang, X.: On Building an Accurate Stereo Matching System on Graphics Hardware. In: *Proc. ICCV Workshop on GPU in Computer Vision Applications (GPUCV)*. IEEE, 2011

[RVKP18] Riebler, H.; Vaz, G.; Kenter, T.; Plessl, C.: POSTER: Automated Code Acceleration Targeting Heterogeneous OpenCL Devices. In: (2018)

[RVKP19] Riebler, H.; Vaz, G.; Kenter, T.; Plessl, C.: Transparent Acceleration for Heterogeneous Platforms With Compilation to OpenCL. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 16 (2019), no. 2, pp. 1–26

[SHW+14] Shan, Y.; Hao, Y.; Wang, W.; Wang, Y.; Chen, X.; Yang, H.; Luk, W.: Hardware Acceleration for an Accurate Stereo Vision System Using Mini-Census Adaptive Support Region. In: *ACM Transactions on Embedded Computing Systems (TECS)* 13 (Apr. 2014), no. 4s, 132:1–132:24

[TLLA14] Tippetts, B.; Lee, D. J.; Lillywhite, K.; Archibald, J. K.: Hardware-Efficient Design of Real-Time Profile Shape Matching Stereo Vision Algorithm on FPGA. In: *Int. Journal of Reconfigurable Computing (IJRC)* (2014)