Journal of Systems Architecture 61 (2015) 628–638

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Automatic task mapping and heterogeneity-aware fault tolerance: The benefits for runtime optimization and application development

Mario Kicherer*, Wolfgang Karl

Karlsruhe Institute of Technology (KIT), Institute of Computer Engineering (ITEC), Kaiserstraße 12, 76128 Karlsruhe, Germany

ARTICLE INFO

Article history:
Received 17 October 2014
Revised 5 October 2015
Accepted 6 October 2015
Available online 23 October 2015

Keywords:
Heterogeneous computing
Accelerators
Fault-tolerant systems
Runtime systems

ABSTRACT

The best mapping of a task to one or more processing units in a heterogeneous system depends on multiple variables. Several approaches based on runtime systems have been proposed that determine the best mapping under given circumstances automatically. Some of them also consider dynamic events like varying problem sizes or resource competition that may change the best mapping during application runtime but only a few even consider that task execution may fail. While aging or overheating are well-known causes for sudden faults, the ongoing miniaturization and the growing complexity of heterogeneous computing are expected to create further threats for successful application execution. However, if properly incorporated, heterogeneous systems also offer the opportunity to recover from different types of faults in hardware as well as in software. In this work, we propose a combination of both topics, dynamic performance-oriented task mapping and dependability, to leverage this opportunity. As we will show, this combination not only enables tolerating faults in hardware and software with minor assistance of the developer, it also provides benefits for application development itself and for application performance in case of faults due to a new metric and automatic data management.

1. Introduction

Accelerators like GPUs promise a significant performance improvement for certain problems compared to the calculation on a general-purpose CPU. However, their actual benefit depends on multiple variables that may even change during application execution like, e.g., the problem size. Therefore, runtime systems have been introduced that dynamically evaluate these variables and determine the best mapping of task to processing unit. Most of them follow an empiric approach: they measure the execution time on the different processing units and choose the one with the shortest time for further executions of the task. During application execution, some of them also consider changing problem sizes or resource competition that may change the best processing unit for a task. However, except of dealing with unresponsive processing units, such runtime systems for task mapping in heterogeneous systems do not consider a failure during task execution. The probability for faults raises, though. Individual programming models and software stacks for the accelerators, e.g., just-in-time compilers and runtime libraries, hardware-specific optimizations as well as necessary data transfers to and from device memory make the source code and the resulting application more complex. To make things worse, the susceptibility of the hardware to faults is expected to increase as well: aging effects and charged particles hitting conductor paths could become a considerable threat for calculations due to shrinking feature sizes [12,25]. For example, Haque and Pande created a test application for GPUs and tested over 50,000 systems. They discovered that *"two-thirds of tested GPUs exhibit a detectable, pattern-sensitive rate of memory soft errors"* [10]. While a fault during the rendering of a frame in a game will be hardly noticed, a fault during a timestep of a simulation might lead to unusable results.

Consequently, application execution will depend on an increasing number of processing units with decreasing reliability and on growing middleware layers that introduce further complexity on their own. However, due to the available redundancy in terms of processing units as well as task implementations for them, heterogeneous systems also offer the opportunity to tolerate different kinds of faults in hardware and in software. In order to leverage this opportunity, different mechanisms are necessary, though.

In this work, we combine an existing runtime system for performance-oriented task mapping with methods for fault tolerant execution of tasks and show how both topics can benefit from each other. For example, with the dynamic task mapping, malfunctioning processing units can be easily avoided. However, if they only suffer from a low rate of transient faults, their benefit for the application compared to other processing units might still exceed the costs for

* Corresponding author. Tel.: +4972160846048.
E-mail addresses: kicherer@kit.edu (M. Kicherer), karl@kit.edu (W. Karl).

http://dx.doi.org/10.1016/j.sysarc.2015.10.001
1383-7621/© 2015 Elsevier B.V. All rights reserved.

recovering from a fault. As the runtime system already maintains a profile for each processing unit, the fault rate can be stored as well and used to create a new metric that determines their remaining benefit. With this profile, also extraordinary long or short task executions can be detected which can indicate an abort or a unit trapped in an endless loop. Furthermore, due to the disjoint memory hierarchies and the resulting expensive data transfers between host and accelerator memories, automatic task mappers are often aware of the necessary data for each task and transfer the data on demand according to the chosen mapping. With a mechanism for automatic data management and the data copies in the different memories, the need to create additional copies for a checkpoint can be avoided in certain cases.

If dual-modular redundancy is used to detect corrupted results, a following question with a considerable impact on performance is where the results of the redundant calculations are compared. With the approach in this work, the comparison can be handled like just another regular task. The runtime system determines necessary data transfers and the execution time for each mapping and chooses the fastest one.

Besides efficient fault tolerance on end-user systems, the combination has also benefits for application development. When writing a task implementation for an accelerator, developers usually have a reference implementation for the CPU that they use to compare and verify the results. Instead of manually comparing the results after a complete application run, our combined approach can do this automatically and for each intermediate result which reduces the time until a developer becomes aware of errors. In case the results differ, additional information about the difference can give valuable hints for debugging the application. Therefore, we also propose a graphical user interface that is called by the runtime system to visualize the differences between the results and to enable further analysis of the results.

Parts of this work are based on a prior publication [15]. Besides a general revision, we contribute additional measurements and the concept of the graphical user interface in this work.

The remainder of this paper is structured as follows: we will first give an overview of related work and state of the art in Section 2. In Section 3, we present the preliminary work that constitutes the basis of our approach. Afterwards, we describe the major techniques of our contribution. Performance and feasibility of this approach are evaluated and compared in Section 5. Finally, Section 6 concludes the paper giving further outlook.

2. Related work

Dependability is a wide research topic with a long history. In this paper, we focus on work related to fault detection and tolerance in modern systems. In the following, we start with the related work focusing on reliability for CPU computations.

Many research projects propose fine-grained on-chip redundancy to decrease the costs for rollbacks and to benefit from underutilized resources. Targeting general-purpose CPUs, several projects utilize the features of modern processors, e.g., multiple cores and superscalar out-of-order pipelines [9,21,23].

Vera et al. [29] also propose a fine-grained redundancy approach for CPUs. They argue that only 20 % of the instructions of a modern architecture are responsible for more than 60 % of the total vulnerability. They introduce so-called selective replication of only certain instructions and achieve a considerable fault coverage while introducing only minor overhead. A similar approach based on VLIW architectures is introduced by Lee et al. [17] that exploits empty slots for dynamic duplication.

As a software-based solution, Rebaudengo et al. present a source-to-source compiler creating redundancy on the source-code level [22]. Their efforts aim to detect transient faults causing data and program-flow corruption.

Besides reducing the overhead of redundant execution, other approaches try to avoid redundancy at all by detecting faults by other light-weight indicators, such as symptoms like anomalous application behavior detected by segmentation faults or an unusual rate of branch mispredicts or cache misses [8]. Such detection mechanisms save time, but come at the price of mispredictions or lower fault coverage.

Besides symptom-based fault detection, arithmetic codes can be used for validation [30]. Here, input values for calculations are modified in a way that the results can be validated using a checksum-like mechanism.

In heterogeneous systems, work-intensive tasks of the application are migrated to accelerators and only protecting the computations on the CPU is not sufficient. Therefore, other projects present their efforts to increase reliability of heterogeneous computing.

Takizawa et al. introduce CheCUDA that enables a checkpoint and restart mechanism for CUDA kernels [27]. In combination with a tool for CPU-bound application checkpointing, applications with CUDA kernels can be restarted after a fault or even be migrated to another host.

Kawai et al. introduced DS-CUDA that allows a normal CUDA application to exploit accelerators on different nodes [13]. In addition, they enable a similar transparent fault tolerant mechanism that executes a task on two GPUs, automatically compares the results and restarts the task if necessary. In contrast to the contribution in this work, they only focus on CUDA applications and devices and neither consider other (fallback) devices nor do they keep statistics of the devices in order to avoid failing devices.

For redundancy-based fault detection on GPUs, Dimitrov et al. [6] introduce and evaluate three possible methods to efficiently execute kernel code multiple times: simple duplication of kernel computations, interleaved kernel instructions, and exploiting unused thread-level parallelism. A similar approach has been presented by Sabena et al. where they compare different methods for redundant execution on GPUs [24]. Like the CPU mechanisms, these efforts concentrate on a single type of accelerator. However, our approach can be used with arbitrary types of accelerators.

Another work targeting GPUs is from Fang et al. [7]. They introduce their debugger-based fault injector GPU-Qin that enables injections on instruction level. In their evaluation, they show that there are different classes of applications that exhibit a similar low or high susceptibility for corrupted results or abortion of execution.

Lee et al. present their extension of an OpenACC-compatible compiler that enables automatic comparison of results from CPU and GPU calculations [18] and determines redundant or missing data transfers for GPU calculations. Similar to this approach, they try to simplify the development of applications for heterogeneous systems with automatic fault detection. However, this work introduces generic mechanisms that are not bound to specific hardware, programming models or compilers and a graphical user interface that visualizes differences to provide additional information for debugging.

Boyer et al. presented a dynamic load-balancing mechanism for systems with non-uniform processing units [3]. In their work, they profile the processing units with small chunks of the total work load during application execution and thereby also detect unresponsive units which are avoided in further runs.

Solutions for efficient task mapping is also of interest in multi- or manycore systems [5,26]. For such systems, Zhang et al. presented a mechanism for efficiently hiding faulty cores in a manycore processor [31]. Their solution maintains a sane view of a logical topology that does not only hide faulty cores but also improves the alignment of the cores for minimal communication costs.