Providing Guidance Information for Application-Mapping on Heterogeneous Parallel Systems

Fabian Nowak, Rainer Buchty, and Mario Kicherer

Chair for Computer Architecture,
Universität Karlsruhe (TH)
P.O.Box 76128 Karlsruhe, Germany
{nowak,buchty,kicherer}@ira.uka.de
http://ca.itec.uka.de

Abstract. With the advent of heterogeneous parallel systems, a method is required for scheduling and mapping applications on the available hardware. We propose the concept of attributes in order to allow a runtime system to decide which implementation is usable given certain constraints such as available hardware resources or power usage. The proposed approach does not break compatibility with existing source code and binary files and can be tightly integrated into a runtime system that handles these additional specifications and allows for reconfigurable hardware to be used according to application requirements and system configuration.

Key words: Multi-Core, Reconfiguration, Scheduling, Attributes

1 Introduction

Multi-core and many-core architectures are the current trend in processor design, offering performance gains without increasing the frequency too much as shown by current vendor road maps and benchmarking results. Continuing the multi-core trend, custom modules for dedicated purposes and reconfigurable systems are becoming more and more attractive and are being built into high-performance, desktop, and embedded systems.

In order to successfully establish heterogeneous environments in commodity systems, additional means have to be found that allow employing the available resources to the maximum extent that is sensible and permissible by internal and external constraints; an example is provided in [6]: for the given example, an appropriate decision whether to choose a hardware or software implementation therefore requires knowledge of the runtime behaviour associated with data size.

Most gain of such systems can be obtained when offering different function implementations for different hardware resources so that according to constraints and requirements as mentioned before, the most suitable execution unit can be chosen. We already proposed a mechanism for mapping function calls onto different implementations [1] using a light-weight runtime system that basically inserts a virtual layer between the application and the hardware resources. As of now, this runtime system does not use an automatic way of work-load assignment; the decision-making of when to execute which function implementation on which unit has to be supported explicitly.

Via so-called attributes, this paper presents an approach to support the process of scheduling and mapping the functions and their implementations for different resources on heterogeneous multi-core systems on single hosts. Attributes pose a suitable means to notify our runtime system of an application's requirements, to track the current system state that is obtainable via monitoring, and to make decisions based on threshold values and rules where to best execute a function call. They can be specified in any source code files and extracted by extensions of the respective compiler tool-chain. Function attributes are then stored within executable binary files and libraries in the Executable and Linkable Format (ELF), hence enabling convenient access by the runtime system's mapping guide and keeping binary compatibility to regular execution environments. Knowing attributes both of the

calling function and of possible target functions in different repository-like libraries enables scheduling software to chose proper implementations.

The paper is structured as follows: Section 2 discusses the related work, in Section 3 the concept of our approach is then explained together with some trade-offs and specific solutions. The implementation of our approach is presented in Section 4. Afterwards, the integration of the concept into our runtime environment is depicted in Section 5. Section 6 concludes the paper and presents our future work in this area.

2 Related Work

In the domain of heterogeneous and reconfigurable architectures, some work has been accomplished that directly addresses programmability issues: EXOCHI [14] is an environment for execution on heterogeneous multi-core systems where the executable binary is extended by accelerator-specific binary data that can be used to simultaneously execute threads both on accelerators and software; Lime [8] tries to bridge the gap between software and hardware programming by extending Java by hardware-near types. However, scheduling and mapping is precomputed using some work-estimation heuristics, which is not generally applicable to the bigger part of applications and programming flows. Furthermore, the concepts are fixed to one programming model and one language only.

Another static approach to building programs based on function calls for different platforms is meta-programming [13], where models are created and then derived for specific architectures, that can also be employed for heterogeneous systems. Although there are meta-programming approaches that do not only create static mappings but rather dynamically compile for even heterogeneous platforms [11], its focus is still on the code generation itself rather than on ease of programmability and specification of objectives.

Mapping of functions and applications onto systems is frequently achieved via flow graphs as in [9]. Possible solutions of how to map functions onto the available resources based on the evaluated application attributes and system state can be found via multi-objective optimisation, delivering a pareto-optimal schedule of both the resources and CPU time for the running attributed tasks. The scheduling problems are solved for example by Dutta et al. [3] with integer linear programming (ILP) being used as means for multi-objective optimisation. A general concept of where to store application-specific information is missing; also, extensibility by new hardware is not granted.

ASKALON [4] is a grid environment for deploying work flows of scientific applications, addressing resource allocation and reliable and fault-tolerant execution. The applications are specified in an XML-based language. Requests for allocating work flows on resources can contain high-level requirements such as clock rate or operating system. Similarly, the Arches framework focuses at bridging the gap between domain-specific scientists and HPC programmers, but does not target resource allocating and job scheduling [2].

The concept to extend a program with additional information has proven to be successful by OpenMP, where so-called pragmas are used to direct the compiler to parallelise the following code region. Also, annotating program code on a semantic level, indicating for example the role of a class instance, is done by Mügge et al. [7].

In contrast to the afore-mentioned works, our concept aims at supporting scheduling and mapping of applications for heterogeneous systems while keeping source code compatibility and not introducing new languages or programming models. It furthermore enables us to execute both regular executables and binary files extended with attributes.

3 Concept

The overall problem under investigation is how to execute programs with regard to assertions and constraints when a specific, but not fixed, perhaps even adapting and changing hardware/environmental setup is given. In such a setup, scheduling has to be achieved considering fixed constraints such as maximum power usage or minimum runtime in order to adhere to power limitations or deadlines. The problem addressed in this paper is how to support a runtime system in finding suitable schedules while integrating smoothly into today's systems and extending existing concepts and programming flows.

As solution to this problem, we suggest to attribute both programs and libraries and to evaluate these attributes in a mapping guide. This is shown in Fig. 1: An attributed executable binary file is started in the runtime environment, providing its attributes to a mapping guide and having its function calls be directed to suitable implementations in software or hardware. The implementations might refer to hardware frameworks [6] that provide implementations in so-called slots with status interfaces via a file system interface called ProcFS. The mapping guide directs the runtime system which function calls to redirect to which different implementations in order to fulfil requirements and features as specified by the attributes and given by the internal system state and external constraints.

It also controls the general system setup, such as controlling reconfiguration of FPGAs or granting exclusive access onto ASIC resources. Inside the virtualization layer of the Dynamic Linking System (DLS), function calls are then routed to different implementations in software or on available hardware according to the commands of the mapping guide.

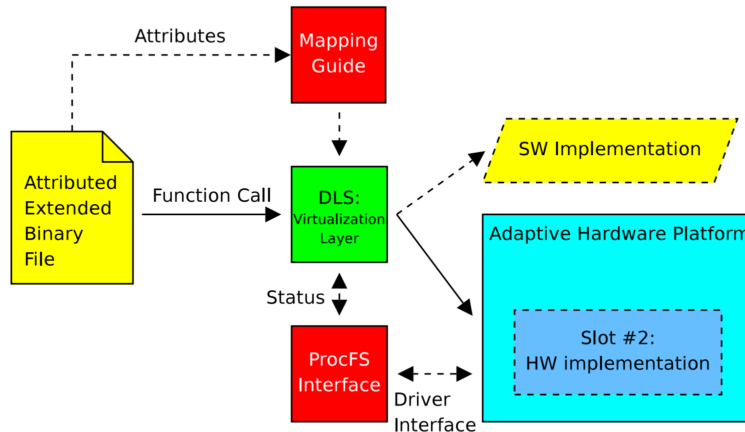

Fig. 1. Overview of the general attribute concept

This is elaborated more precisely in Fig. 2: both the application with the function calls inside and the available software implementations are to be attributed, alleviating the task of deciding which implementation to use.

As already mentioned before, the attributes need to be extracted from the program code via a preprocessor and inserted into the binaries. The flow of attribution, extraction and extension is depicted in Fig. 3 with both the source file and the binary file. First, the program sources need to be extended by pragmas with the statically known information of program requirements and implementation capabilities. Then, an extended GCC-based toolchain extracts the pragmas, compiles the source code and links the objects, and finally attaches the extracted attributes to the executable. This information is stored in a new DLRS segment, which contains an arbitrary attributes segment among others, as can be seen on the right.

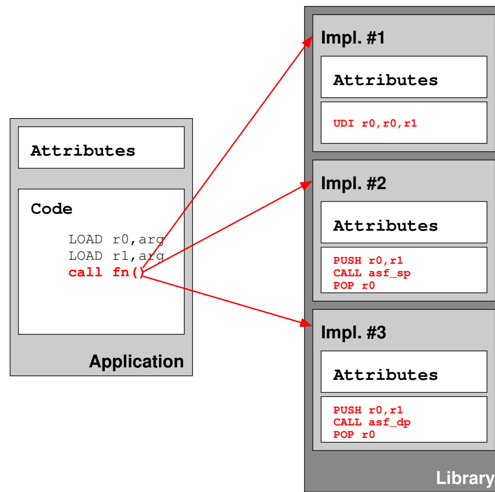

Fig. 2. Mapping of a function onto suitable implementation

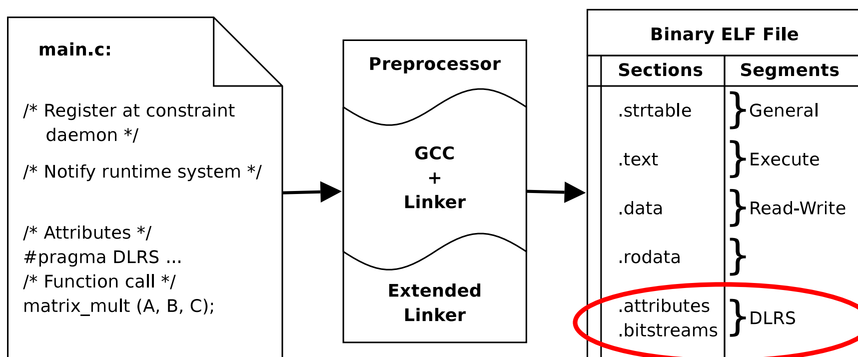

Fig. 3. Attributed Program and Compiled Binary File

With attributes such as bandwidth, maximum memory consumption, preferable input data size and normalised execution times, it is possible to automatically call a suitable implementation because specific demands or properties of software or hardware implementations are known from the attributes of their containing libraries. In the long run, the attributes will help close the feedback loop: applications and monitoring systems can adjust attributes such as the reliability to the measured values, derive new attributes or even indicate incorrect implementations and broken hardware.

In our twofold approach, we propose the use of pragmas to augment the program code with attributes achieving that source code remains compatible with existing programming environments and compilers, and the binary extension is in such a way that annotated binaries can be executed in legacy environments and regular binaries can still be executed in our environment. Also, the general approach is completely language-agnostic. The following subsections explain the extensions in more detail.

3.1 Code Extension

Attributing program code works as follows: both function implementations and calls within the Dynamic Linking and Runtime System (DLRS) are prefixed with `#pragma DLRS` and an arbitrary number of `key=value` pairs that describe the features or requirements of the implementation. In order to have a usable parameter set, the programmer and the surrounding tools are required to use orthogonal parameters only, i.e. they do not depend on or influence each other, and to use the same attribute names for all functions in a project. By integrating these parameter names into existing programming environments, this does not pose any severe restrictions. Secondly, function calls are attributed in the same way, hence specifying the requirements if available. To give a better understanding, Fig. 4 contains two annotated function implementations of the *3DES* function, while in the *main* method the requirements are given.

```
#pragma DLRS name=calculate_3DES data_size_min=10KB target=FPGA
double * calculate_3DES_hw (double* x) {
  return hmol_accelerate_direct (accel_slot, ...);
}

#pragma DLRS name=calculate_3DES data_size_max=50KB target=CPU
double * calculate_3DES_sw (double* x) {
  double *encrypted;
  ...
  return encrypted;
}

int main (int argc, char **argv) {
  double *input, *encrypted;
  input = ...
  #pragma DLRS prefer_hardware=1 data_size=30KB
  encrypted = calculate_3DES (input);
  ...
}
```

Fig. 4. Attributed Caller and Callee Implementation

As can be seen from this example where both implementations seem to equally suit the attributed function call with regard to sensitive data sizes, it is necessary to have additional high-level constraints such as power-efficient execution in order to facilitate the decision which implementation to choose. For example, this can be achieved by classifying hardware implementations as generally more power-efficient, so `calculate_3DES_hw` would be the choice for power-efficient execution.

3.2 Binary Extension

For the attributes to be used by the runtime system and constraint daemons, we suggest to extend the basic executable file by the attributes. A different approach would be to deliver additional information files together with the executable, but this would harm the basic concept of how programs are delivered and executed.

Executable files for modern UNIX-like systems are stored in the Executable and Linkable Format [10]. Basically, such an executable consists of several headers that specify the number of segments and their offsets in the file. A segment may comprise several sections in the file. Sections contain program code, constants etc.

We propose to add a new segment for the Dynamic Linking Runtime System. As of now, it consists of two sections: attributes and hardware bitstreams. In the first section, all the specified attributes have to be stored for each attributed function of any source file. To access the names, either the regular string section can be used or a new string section within the DLRS segment. The second section for the bitstream is not addressed in this paper.

4 Implementation and Results

4.1 Toolchain

The attributes from the extended code need to be acquired by parts of the programming tool-chain as has already been indicated in Figure 3. We developed a separate preprocessor in Python that is able to extract the DLRS-specific pragmas. It is designed in such a way that pragmas such as these of OpenMP can still be used.

For extending the ELF binary, we developed an ELF editor as depicted in Fig. 5.

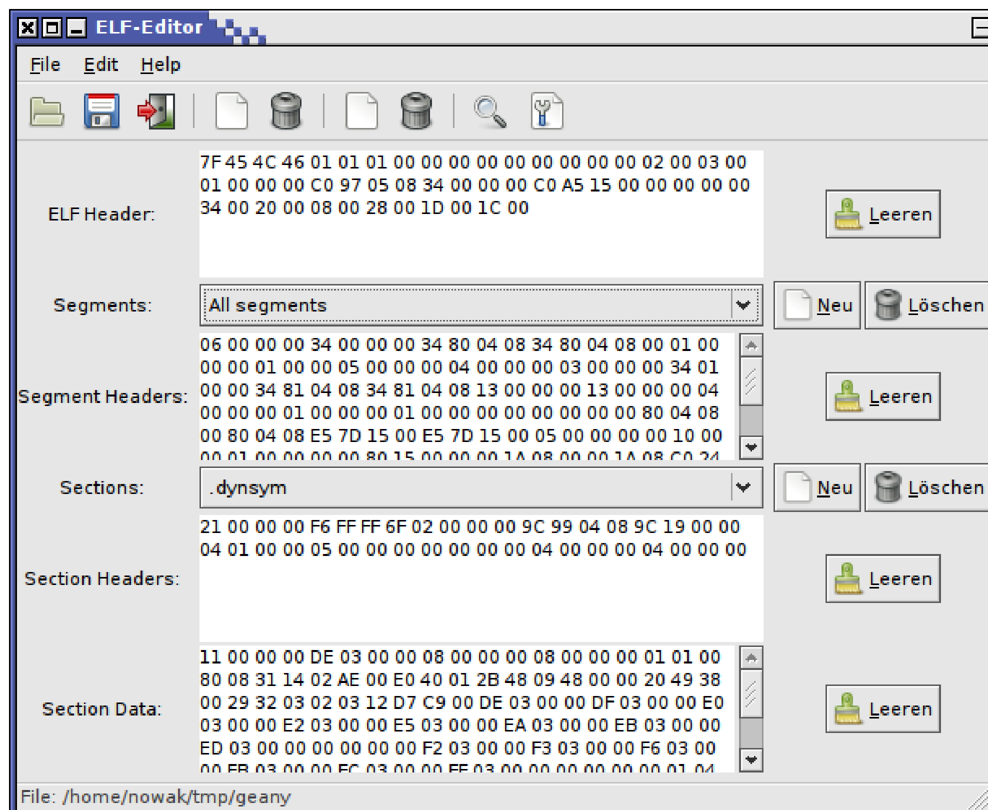

Fig. 5. ELF Editor for working with headers, segments and sections of ELF files

4.2 Evaluating Attributes at Runtime

With respect to the already existing runtime system, the question occurs where and how to evaluate attributes.

The evaluation could occur within the application itself, which is not a preferred approach as applications cannot be aware of other programs' resource demands and of the overall system status. Hence, there has to be a central instance that is aware of the available computing and memory resources in the system and of the remotely accessible resources as well. The application attributes are stored within the binary and hence can be retrieved via ELF access libraries such as libelf [5] from both the application itself and from the central instance. For the central approach, this instance has to be notified of a new "client". This can be achieved if the client program registers at the instance so that the instance obtains the name of the program's executable and can extract the attributes. The system attributes are on the one hand obtained from monitoring instances in the system and from those attached to the hardware directly, and on the other hand from the amount of occupied and free resources.

5 Integration into the Runtime System

In this section, we will show how the attribute concept is integrated into our runtime environment [1]. This runtime system [1] leverages switching between different implementations of the same functionality such as executing in hardware or using additional error analysis in contrast to the regular software implementation. The requirement is to allow switching both from within the application and from system level in order to allow more sophisticated schedulers to maximise system usage in terms of number of finished jobs or processed data by employing additional hardware resources such as FPGA boards and graphics adaptors.

There are two possible ways of achieving that: the ELF concept allows to extract function names from an application and upon that, the referenced function implementation can be switched to an alternative one that is already present in the address space. This is called the GOT-based Linking System (GLS). Furthermore, it is possible to reroute the function calls from another application via the Linux kernel. The set of possible function implementations for the function calls can be set up and even extended at runtime, and hence must not be known at compile time except for one default target. This is our Dynamic Linking System (DLS).

After obtaining the respective notification from a program before the function call finally happens, the mapping guide directs the runtime system to switch the target of a function call to a different implementation that provides additional benefit over the regular one with regard to certain optimisation criteria such as power, performance, or correctness of the result. At this point, the attributes provide the necessary information for deciding which implementation to finally use, adhering to global system constraints.

For executing the program and mapping the program calls, there two alternatives that both use a central instance as described in 4.2. The first one is calling and executing the program by means of the runtime environment, while the second is introducing a middle layer implemented as a shared library that communicates with the DLS via IPC before each function call to be executed.

The latter is preferable for manually guiding the program. However, we prefer the first alternative because it allows to statically calculate a first mapping of all the program calls and to later on adjust the mapping dynamically.

6 Conclusions and Future Work

In this paper, we presented attributes as an approach for executing different function implementations subject to system and application constraints. The approach requires a preprocessor extracting pragmas that are then inserted by an additional backend into regular executable binary files and into libraries that contain different function implementations. With the introduction of one additional specification layer in addition to the program code itself, the programmer is free to exploit a correspondingly extended system to the maximum extent allowed by the system with as many hardware accelerators in use as possible and sensible and also supporting reconfiguration of reconfigurable resources; but he is still free to use additional programming models such as OpenMP or even other programming languages than C. Attributes proved to be reasonable for making decisions on which implementation to use based on input data size, but can also be used to care for tracking the status of dynamically reconfigurable hardware.

Ongoing work consists of translating high-level user-specific and user-specified attributes into low-level attributes that can be interpreted by the mapping daemon, and of creating metrics for comparing the low-level attributes in such a way that the underlying architecture does not matter. Of special interest are dedicated algorithms for evaluating the attributes.

For the future, we plan to target automatic hardware generation via such pragma attributes ("hardware-synthesizable") by the preprocessor calling additional tools such as the Trident compiler [12] and other C-to-VHDL tools, aiming at automatic creation of hardware implementation repositories.

References

1. Rainer Buchty, David Kramer, Mario Kicherer, and Wolfgang Karl. A light-weight approach to dynamical runtime linking supporting heterogenous, parallel, and reconfigurable architectures. In *Architecture of Computing Systems – ARCS 2009*, volume 5455 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin / Heidelberg, February 2009.
2. Nathan DeBardeleben, Ron Sass, Daniel Stanzione, Jr., and Walter B. Ligon, III. Building problem-solving environments with the arches framework. *J. Syst. Softw.*, 82(7):1137–1151, 2009.
3. Hritam Dutta, Frank Hannig, and Jürgen Teich. Performance Matching of Hardware Acceleration Engines for Heterogeneous MPSoC Using Modular Performance Analysis. In *Architecture of Computing Systems – ARCS 2009*, volume 5455 of *Lecture Notes in Computer Science*, pages 233–245. Springer Berlin / Heidelberg, February 2009.
4. T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. In *6th International Workshop on Grid Computing (Grid 2005)*, Seattle, USA, November 2005. IEEE Computer Society Press.
5. Joseph Koshy. libelf by Example. `http://people.freebsd.org/~jkoshy/download/libelf/article.html`.
6. David Kramer, Thorsten Vogel, Rainer Buchty, Fabian Nowak, and Wolfgang Karl. A general purpose HyperTransport-based Application Accelerator Framework. In *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA2009)*, pages 20–29. Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg, February 2009. ISBN 978-3-00-027249-3.
7. Holger Mügge, Tobias Rho, and Armin B. Cremers. Integrating aspect-orientation and structural annotations to support adaptive middleware. In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 9–14, New York, NY, USA, 2007. ACM.
8. Shan Huang and Amir Hormati and David Bacon and Rodric Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *Proceedings of the 2008 European Conference on Object-Oriented Programming (ECOOP)*, July 2008.
9. Lodewijk T. Smit, Gerard J.M. Smit, Johann L. Hurink, Hajo Broersma, Daniël Paulusma, and Pascal T. Wolkotte. Run-time assignment of tasks to multiple heterogeneous processors. In *5TH PROGRESS Symposium on Embedded Systems*, pages 185–192. STW Technology Foundation, 2004.
10. The Santa Cruz Operation, Inc. System V Application Binary Interface (Edition 4.1), 1997. `http://www.caldera.com/developers/devspecs/gabi41.pdf`.
11. Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):1–40, 2008.
12. J.L. Tripp, K.D. Peterson, C. Ahrens, J.D. Poznanovic, and M.B. Gokhale. Trident: an FPGA compiler framework for floating-point algorithms. *Field Programmable Logic and Applications, 2005. International Conference on*, pages 317–322, August 2005.

13. Salvador Trujillo, Maider Azanza, and Oscar Diaz. Generative Metaprogramming. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 105–114, New York, NY, USA, 2007. ACM.
14. Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. *SIGPLAN Not.*, 42(6):156–166, 2007.