

# Delivering Guidance Information in Heterogeneous Systems

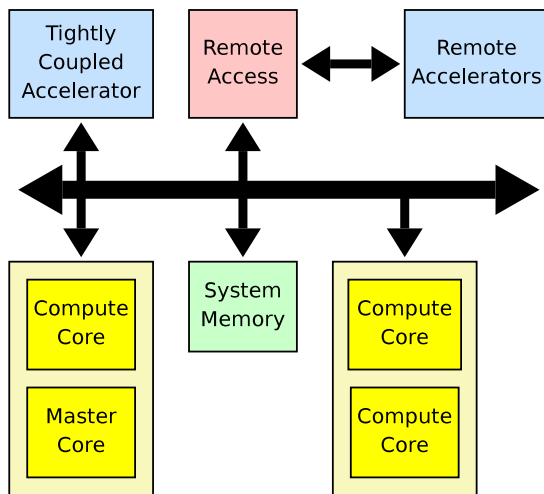
Fabian Nowak, Chair for Computer Architecture, Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany  
Mario Kicherer, Chair for Computer Architecture, Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany  
Rainer Buchty, Chair for Computer Architecture, Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany  
Wolfgang Karl, Chair for Computer Architecture, Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany

## Abstract

With the advent of reconfigurable platforms and GPUs, we need means to successfully support the programmer and the system scheduler in efficiently exploiting the resources of the system. We present an approach of how the programmer can annotate the code so that a hardware abstraction layer can choose which resource to use for a given function call in a user-transparent manner. We evaluate the introduced toolchain and the time required to obtain these code annotations.

## 1 Introduction

Due to the rise in heterogeneity in today's systems, the question comes up how to support the programmer in writing programs for such systems. These programs have to be capable of exploiting a majority or even all of the system's resources while also running at an optimum with regard to one or more constraints. Possible resources include GPGPUs, reconfigurable coprocessors, and extension boards among others as sketched in **Fig. 1**.



**Figure 1:** Simplified sketch of possible heterogeneous target environments

Constraints may be performance, energy-awareness, or combinations such as efficiency. The frequently employed hardware-aware computing that adheres to such constraints leads to statically bound program code that in return is complex to create, extend and maintain.

A supporting runtime system for hardware abstraction in heterogeneous parallel systems that allows the adaptation of an application to the current environment at runtime has been proposed in [3]. In detail, the runtime system is capable of rerouting a function call to a different implementation in the executable itself or in an external library. It provides an API for control daemons and the program itself to request the rerouting and, additionally, a user-accessible

interface. The idea behind is to have functions with given semantics separated from implementations that allow calculation on a specific system resource or in a specific manner.

In order to calculate an efficient configuration of function mappings and required resources for a given environment at program start or even at runtime, the control unit needs information about the application's requirements and the offered qualities of the available implementations. In this work, we present a mechanism to store these requirements and qualities as so-called attributes directly in the executable and library files in a backward-compatible fashion. The mechanism is evaluated with regard to costs in toolchain extension and scalability in requesting the attribute information from the binary file.

This paper is structured as follows: First, we present the related work in Section 2. We then introduce our concept of delivering requirements and properties of implementations via attributes in Section 3. The implementation of this concept is laid out in Section 4, its interface is detailed in Section 5. Section 6 presents the evaluation of the proposed system. Our paper is summarized in Section 7, where an outlook on future work is also given.

## 2 Related Work

Compute resources in a single system might be different with regard to their vendors, instruction sets, and interfacing, hence requiring multiple-target binaries or separate bitstreams for individual configuration. In [14], an approach for assigning functions to the heterogeneous cores is presented.

OpenCL [5] constitutes an approach to foster development of a single programming model for FPGA platforms and GPGPUs that both until recently came with their own subsystems [6, 1].

There also exist approaches [2] to precompute static configurations of function call schedules for a given target architecture when employing the Bulk-Synchronous Parallel programming model (BSP) for execution on parallel architectures. However, the presented work is statically bound to one homogeneous architecture with similar input sizes

known in advance while in contrast we target runtime support for hybrid and heterogeneous systems.

Considering reconfigurability, [4] gives a DSP capable of exploiting this great feature for embedded systems. Also in the domain of controlling program execution inside FPGA SoCs, the SDVM<sup>R</sup> constitutes an interesting approach [7].

In [8], we developed a framework for partitioning FPGAs to allow concurrent access onto an FPGA for multiple users with each running potentially different hardware acceleration engines. The system showed to already be successful when running a single 3DES core, outperforming software execution starting at input data sizes of 10KB.

This framework was motivated by Molen [11] that allows user-defined instructions to execute accelerated function versions in reconfigurable silicon. However, there is no clear programming paradigm and no holistic, transparent approach to deliver the FPGA bitstreams for the functions to be accelerated in the binary itself.

Finally, in [3] we presented a lightweight system allowing switching function calls to desired libraries and implementations at runtime. This can be steered both from within the program itself and from outside, i.e. an extended scheduler being aware of the overall system status and the program's requirements.

The preceding work is illustrated in [10], explaining the overall concept and possible approaches for implementation, including the toolchain and further tools.

### 3 Concept

Heterogeneous systems are architectures as depicted in Fig. 1, where tightly coupled accelerators frequently are GPGPUs or in-socket FPGAs, while remote accelerators can be attached via Ethernet or PCI-like buses. Most programs are started on one random core that acts as a master core for split and gather operations, result printing, status information and so on, while the others are frequently employed for computation only. The key for efficient usage of these resources is to offer a mechanism to indicate in advance the required features of the implementation that is going to be used. Hence, in this section we present our concept of how to add attributes for requirements and properties to both programs and libraries. This information can then be used in control daemons as in Fig. 2 to direct the central component of the Dynamic Linking System (DLS), the Dynamic Linker itself, of which library or specific function implementation is most appropriate with respect of the current resource usage and accessibility.

We already suggested extending the ELF binary file of an application by its a-priori known requirements and desires. The second part of the concept is to also annotate existing library implementations via the same means, which is possible because libraries are also stored in ELF. When linking to dynamic libraries, relocation information for the symbols needs to be added to both executable files and library files; in the same way, attributes and additional data can be attached as is depicted in Fig. 3.

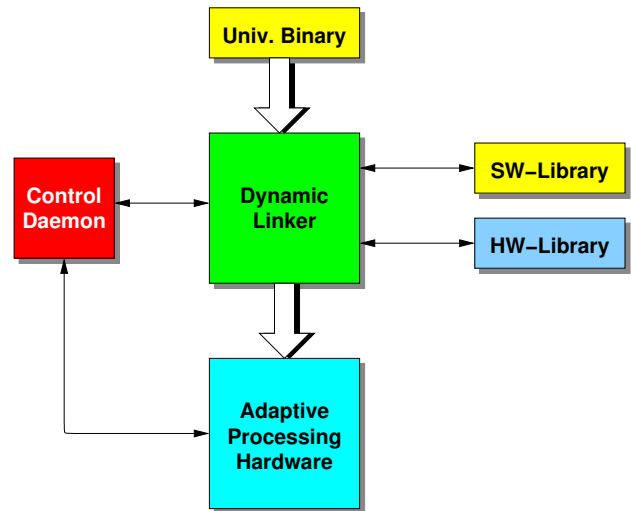


Figure 2: Dynamic Linking System (DLS): The control daemon directs the linker of which library to choose

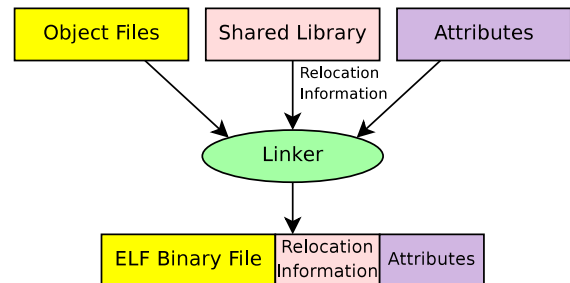


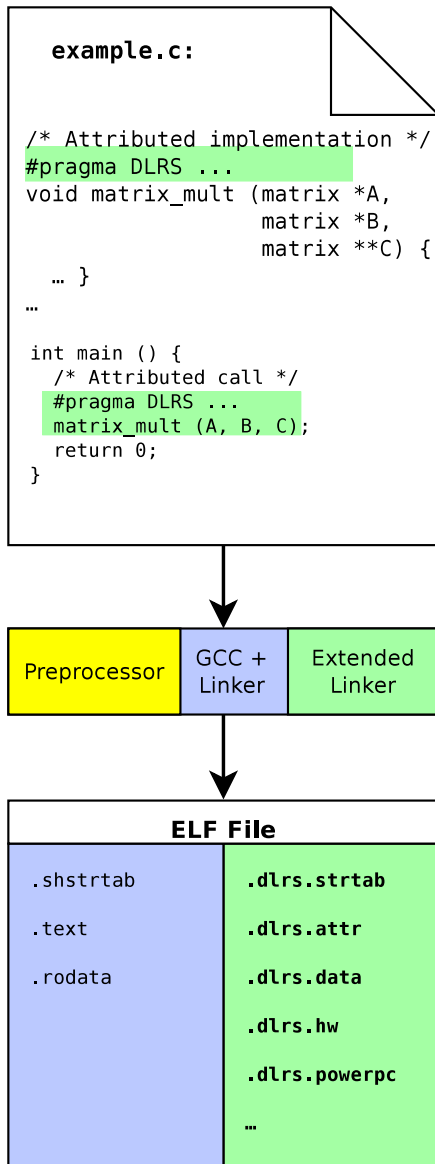
Figure 3: Storing relocation information for dynamic libraries and attributes altogether in a single ELF file

We name the combination of the Dynamic Linking System (DLS) and the attribute extensions for runtime evaluation *Dynamic Linking and Runtime System (DLRS)*.

### 4 The DLRS Sections

When extending regular binaries that are stored as ELF files, it is important to remain compatible with legacy systems and, of course, legacy programs themselves. So we are not allowed to alter any content directly, instead, we need to add new and distinct sections for all the information required.

These new sections are indicated in Fig. 4 where extensions to the regular GNU toolchain extract the attributes from the source code and create additional, non-conflicting additional sections for organizational data, attributes, additional data, and even program code for multi-target binaries.



**Figure 4:** Generation of attributed ELF files from annotated source code by toolchain extensions

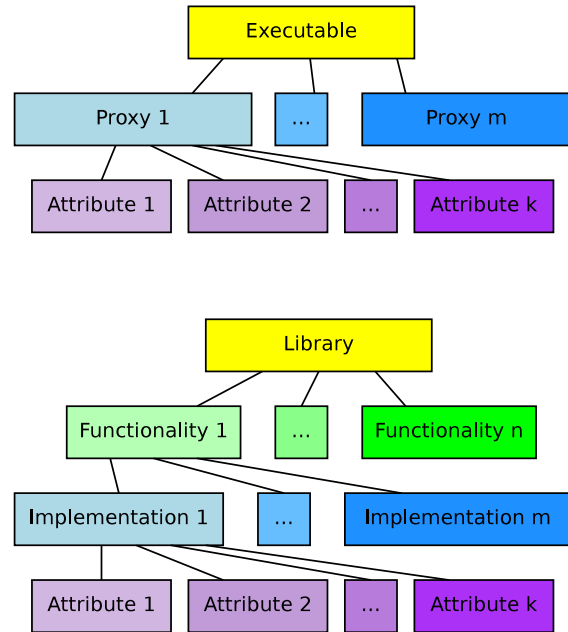
## 4.1 String Tables

Speaking of information, there are two aspects to be considered: first of all, information is mainly represented in string form, i.e. function names, attribute keys, attribute values, and program names. This information should be stored in a string table in order to allow reuse of individual strings, especially because many of the keys will be the same. However, using the regular string table is not recommended as it would alter and influence the regular program and break the afore-mentioned compatibility.

## 4.2 Attribute Tree

The other aspect with regard to information is organization: As a program is structured by its main routine, the function calls inside, and the function implementations, storing data in a tree-like structure is desirable in order to

allow easy and convenient access within logical order. **Figure 5** illustrates the tree-like organization of the attributes: in the executables, every proxy (that represents a function pointer) has a set of attributes describing its requirements. In the libraries, the implementations are organized in so-called functionality groups. Each implementation in such a group does the same job, but in a different way. To distinguish between them, every implementation has also a set of attributes describing their qualities.



**Figure 5:** Tree-like organization of proxies in executable files and functionalities in libraries with their respective attributes

## 4.3 Implementation

As already mentioned, each proxy can refer to multiple implementations. Normally, there is only one implementation per library/executable available. Such a proxy requires a string table section for the attribute keys and values, the proxy names and some more strings. Another section contains the “pairing” of keys and values for each proxy, and additional sections may appear for hardware descriptions, executable code for different instruction sets, or acceleration kernels. When several implementations are put together in one file, they are grouped by their functionality, but separate sections are used for each implementation in the ELF file, which are then labeled more accurately with the function names instead of the functionality.

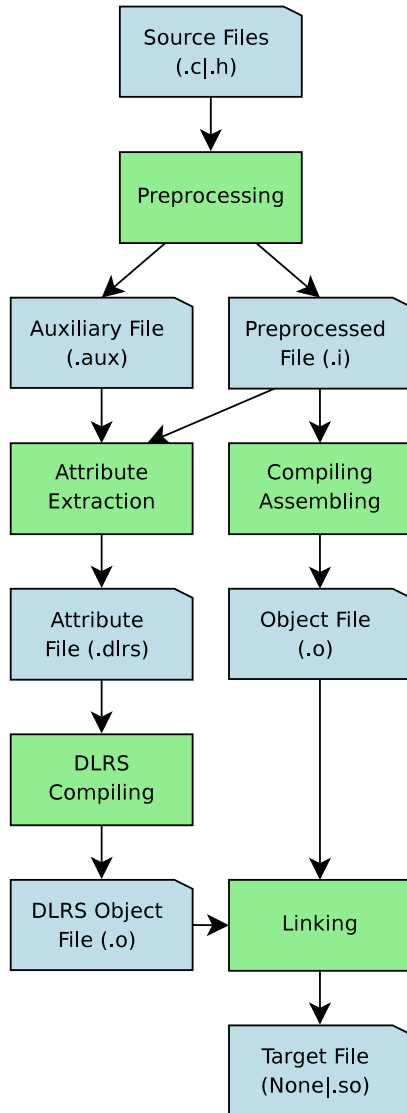
## 5 Interface

With the general concept and the DLRS sections in mind, we now present the interface to these sections, which is used by the toolchain itself and allows convenient access for the afore-mentioned control daemon as of Fig. 2.

## 5.1 DLRS Library

The DLRS library `libdlrs` fulfills two purposes: on the one hand, it allows our tool such as `dlrsobj` to write the DLRS object file that is later on linked to the remaining object files in order to create an attributed binary file. On the other hand, control software is able to read the attributes from the extended target files and to then further direct the dynamic linker. Of course, the interface for reading is invaluable for debug tools such as `dlrsdump` as well.

## 5.2 Toolchain Extension



**Figure 6:** Toolchain extracting the attributes, creating intermediate files and new object files, and finally linking all objects into attributed ELF files

In order to extract the attributes and link a separate attribute object file to the other object files, the toolchain has been extended as in **Fig. 6**: first, the source files are preprocessed by GCC, resulting in a `.i` file. Then, `.aux` files are generated while compiling the source code to regular object files. These `.aux` files allow to easily extract the remaining, unprocessed pragmas with our attributes

by `dlrsextract`, and to generate the DLRS object file with a dedicated compiler `dlrsobj`. The code is compiled regularly, leading to a bunch of object files that are finally linked together with the DLRS object figureherefigureherefile, producing our attributed binary target file.

## 6 Evaluation

We evaluate our implementation by carrying out the following measurements: increase in the executable file's size due to overhead for the general organization, increase in data size with increase in number of attributes and implementations, and execution time of the extended toolchain, and time consumed for acquiring the attributes from the section representations. Evidently, there is no real impact caused by our program extensions apart from the desired support of programs for heterogeneous systems.

The following tests were run on an Intel Core2 Quad 2.4GHz.

### 6.1 Increase in File Size

For our attributed files, we can already estimate the overhead based on data management in the additional sections. However, having the same attributes multiple times will even save file size due to multiple reuse of the strings by means of the proxy-specific string section. Hence, we present the file sizes resulting from the regular toolchain and from our toolchain for several different numbers of proxy implementations and attributes both for reused and unique attributes in **Tables 1, 2**. In our case, the attribute tuples consist of 5 bytes for key and value each, therefore file size will vary when using longer key names or longer values, respectively. Similarly, the implementation names themselves are stored inside which further impacts file size.

When putting more than one proxy/functionality into source files, the file size simply grows by the same amount when adding attributes to plain source files (given that the number and size of the attributes are the same), and hence is not shown in this section.

### 6.2 Acquiring Attributes

More important than the increase in file size is the time required to obtain the attribute information. Thus, we measured the time taken to obtain proxy information and the according attributes from the binary files. All measurements have been executed without any optimization. **Table 3** summarizes the execution times for requesting attribute data for 1, 10, and 100 proxy implementations with 1, 10, and 100 attributes per implementation.

# implementations	Regular	DLRS				
		0 attribs	1 attrib	10 attribs	50 attribs	100 attribs
1	10,952B	11,048B	11,336B	11,593B	12,713B	14,106B
2	10,985B	11,081B	11,433B	11,818B	13,578B	15,787B
10	11,242B	11,338B	12,138B	13,675B	20,571B	29,164B
50	16,618B	16,714B	19,754B	27,051B	59,547B	100,140B
100	18,219B	18,315B	24,155B	38,668B	103,148B	183,741B

**Table 1:** File sizes in Bytes of resulting binaries when reusing the same attributes

# implementations	Regular	DLRS				
		0 attribs	1 attrib	10 attribs	50 attribs	100 attribs
2	10,987B	11,083	11,435	11,932B	14,172B	16,973B
10	11,244B	11,340B	12,140B	14,653B	25,853B	39,854B
50	16,620B	16,716B	19,756B	33,757B	89,757B	159,758B
100	18,221B	18,317B	24,157B	52,158B	152,958B	304,159B

**Table 2:** File sizes in Bytes of resulting binaries when using unique attributes

# attributes	# of implementations			
	1	10	50	100
1	2.7 $\mu$ s	2.2 $\mu$ s	2.9 $\mu$ s	3.0 $\mu$ s
10	2.5 $\mu$ s	2.6 $\mu$ s	2.9 $\mu$ s	3.0 $\mu$ s
50	2.3 $\mu$ s	2.7 $\mu$ s	3.0 $\mu$ s	3.1 $\mu$ s
100	2.7 $\mu$ s	2.9 $\mu$ s	3.0 $\mu$ s	4.0 $\mu$ s

**Table 3:** Time required to obtain the complete attribute data

attribute	implementation		
	first	last	mean
first	4.703ms	4.734ms	4.718ms
last	4.718ms	4.750ms	4.734ms
mean	4.710ms	4.742ms	

**Table 4:** Time required to obtain attribute information of first and last attributes and proxy implementations when considering the first proxy

We also measured the time for acquiring the first and last attributes of the first and last proxy/functionality from a binary file with 100 implementations and 100 attributes each with 10 proxies in total. As the sequential search in the string tables might deliver varying request times, we also calculated the mean values in **Table 4**, which indicate that access time is considerably stable and fairly low in general even when iterating through the data structures.

**Table 5** presents the results when accessing the first/last functionality implementation, which isn't necessarily the 10th implementations but depends on the order in which the implementations are stored by the DLRS library.

attribute	implementation		
	first	last	mean
first	4.718ms	4.763ms	4.740ms
last	4.728ms	4.742ms	4.735ms
mean	4.723ms	4.753ms	

**Table 5:** Time required to obtain attribute information of first and last attributes and proxy implementations, respectively, when considering the last proxy only

All these results clearly show that operating systems effects still massively influence the execution times, and hence no conclusion can be drawn with regard to decrease in speed for larger data amounts. This is very good as our

system performance won't be affected or degraded at all by these extensions.

### 6.3 Execution Time of the Toolchain

We measured the build times of the `make` processes both with the regular GNU toolchain and the attribute-extended toolchain; additionally, we also took the individual processing times of `dlrsextract` and `dlrsobj`, respectively, over an average of 10 runs. To get an understanding of scalability and applicability for large projects, we ran several tests with increasing numbers of attributes. For each proxy, its name is the first attribute, and the remaining attributes are a dummy enumeration. Note that for the function call, there is one additional attribute stored except for the case with zero attributes. From the results presented in **Table 6**, we can see that from a toolchain point of view, the attributes and extensions account for nearly a third of the build time, but at least scale very well with the number of attributes.

Similar to the previous test, we also tested the influence of the number of proxy implementations when not sharing the same attribute names. **Table 7** clearly shows that build time is strongly influenced by linking the additional sections together because the DLRS tools have only a minor impact on the overall build time, but the build time grows dramatically for a high number of implementations in comparison to the regular toolchain.

From these results we can conclude that for large-scale programs, the overhead becomes even less significant as a major part of the build time is required for compiling the source code to object files in contrast to the current empty function implementations that are compiled very fast. Hence, our evaluation only gives the worst-case impact.

## 7 Conclusions and Future Work

Today's heterogeneous systems require high-level but transparent support for specifying additional information including an implementation's properties and a program's

# of attributes	Regular Toolchain	Extended Toolchain	dlrsextract	dlrsobj
0	74.3ms	108.2ms	23.2ms	5.2ms
1	74.4ms	113.6ms	23.3ms	5.0ms
10	73.9ms	111.8ms	23.4ms	5.0ms
50	74.5ms	111.1ms	23.2ms	5.1ms
100	74.7ms	112.7ms	23.6ms	6.0ms

**Table 6:** Execution times of the regular and extended toolchain for one proxy implementation and varying number of attributes

# implementations	Regular Toolchain	Extended Toolchain	dlrsextract	dlrsobj
2	77.0ms	113.8ms	23.1ms	5.1ms
10	80.8ms	118.7ms	23.2ms	5.0ms
50	100.0ms	144.3ms	23.5ms	6.2ms
100	122.1ms	175.8ms	23.5ms	7.7ms

**Table 7:** Execution times of the regular and extended toolchain for varying number of proxy implementations with each 10 attributes

expectations. We presented our implementation to achieve this goal, which consists of adding additional sections to the binary file of a program stored in the executable and linkable format (ELF). This ensures that the implementation is usable for both dynamic libraries and programs, so that both a program’s requirements and a library’s implementation capabilities can be specified. The additional attribute information of function implementations is stored in further sections of the ELF binary file that are transparent to regular UNIX-like systems. Information is accessed in a tree with the appropriate pointers to the data in a separate strings section. Besides, binary data such as FPGA bitstreams can also be stored and managed.

Further work consists of arranging and merging requirements and properties in such a way that subject to an overall goal, e.g. power consumption, optimal execution is guaranteed.

## 8 Acknowledgments

For his tremendous work concerning the implementation of the attribute concept and the extensions to the toolchain, we sincerely thank Jiefei Chen.

## References

- [1] Advanced Micro Devices, Inc. Brook+ SC07 BOF Session. Nov 2007. <http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf>.
- [2] Olaf Bonorden, Friedhelm Meyer auf der Heide, and Rolf Wanka. Composition of efficient nested BSP algorithms: Minimum spanning tree computation as an instructive example. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 2202–2208, June 2002.
- [3] 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.
- [4] Fabio Campi, Antonio Deledda, Matteo Pizzotti, Luca Ciccarelli, Pier Luigi Rolandi, Claudio Mucci, Andrea Lodi 0002, Arseni Vitkovski, and Luca Vanzolini. A dynamically adaptive DSP for heterogeneous reconfigurable platforms. In Rudy Lauwereins and Jan Madsen, editors, *DATE*, pages 9–14. ACM, 2007.
- [5] Khronos Group. OpenCL – Parallel Computing for Heterogeneous Devices, August 2009. [http://www.khronos.org/developers/library/overview/opencl\\_overview.pdf](http://www.khronos.org/developers/library/overview/opencl_overview.pdf).
- [6] Tom R. Halfhill. Parallel Processing with CUDA. In *Microprocessor Report*, volume Jan 28, 2008. <http://www.mpronline.com/>.
- [7] Andreas Hofmann and Klaus Waldschmidt. SDVM<sup>R</sup>: A Scalable Firmware for FPGA-based Multi-Core Systems-on-Chip. In *9th Workshop on Parallel Systems and Algorithms (PASA 2008)*, volume LNI P-124, pages 59–68, Dresden, Germany, January 2008. GI e.V.
- [8] 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 30–38. Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg, February 2009.

- [9] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 287–296, New York, NY, USA, 2008. ACM.
- [10] Fabian Nowak, Rainer Buchty, and Mario Kicherer. Providing guidance information for application-mapping on heterogeneous parallel systems. In *PARS Newsletter #26, GI/ITG*, December 2009. to appear.
- [11] S. Vassiliadis and S. Wong and S. D. Cotofana. The MOLEN  $\mu$ -coded Processor. In *11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147*, pages 275–285, August 2001.
- [12] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming model for a heterogeneous x86 platform. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 431–440, New York, NY, USA, 2009. ACM.
- [13] 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.
- [14] 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.
- [15] 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.