

An embrace-and-extend approach to managing the complexity of future heterogeneous systems

Rainer Buchty, Mario Kicherer, David Kramer, Wolfgang Karl

Universität Karlsruhe (TH)
Institut für Technische Informatik, Lehrstuhl für Rechnerarchitektur
76128 Karlsruhe, Germany
e-Mail: {buchty|kicherer|kramer|karl}@ira.uka.de

Abstract. In this paper, we present a particularly lightweight, integrative approach to programming and executing applications targeting heterogeneous, dynamically reconfigurable parallel systems. Based on an analysis of existing approaches, we strictly focused on compatibility and lightweightness. Our approach therefore follows an embrace-and-extend strategy and achieves desired functionality by adopting and augmenting existing system services, achieving the desired properties. We implemented this concept using the Linux OS and demonstrated its suitability with a heterogeneous platform comprising IA32 multicore processors and current FPGA accelerator hardware using state-of-the-art Hyper-Transport interconnection technology.

1 Introduction & Motivation

Driven by advances in fabrication technology, the computing power of individual microprocessor cores was constantly increasing over the last decades with clock rates boosted by three orders of magnitude and increasingly complex processor microarchitectures. This led to current superscalar designs exploiting ILP using sophisticated out-of-order execution and prediction techniques. With the addition of integer and floating-point vector units also data parallelism was exploited, paving the way for ubiquitous multimedia applications. The increased capabilities were implicitly usable by the programmer and, in worst case, required the use of dedicated libraries and recompilation.

Due to technological limitations, these past approaches of automatically gaining more performance by increasing single-core performance have come to a halt. Forecasts indicate a massive growth of manycore architectures and reverting the trend of most powerful individual “all-purpose” processor cores. For future multi- and many-core architectures a mix of trimmed general-purpose processors and dedicated application accelerators, e.g. cryptographic units or network processors, is envisioned: recent examples are Intel’s Atom processor (lacking costly out-of-order execution), or their re-animation of 1994’s P54C processor architecture (now offering an improved V pipeline for vector operations) for use in the upcoming Larrabee architecture. The combination of general-purpose and application-specific processing units within a current multicore processor is demonstrated by Sun’s T1 (Niagara II) processor

Improvements in FPGA technology foster the use of reconfigurable logic instead of static accelerators, enabling a more flexible use of the silicon resources by dynamically

reconfiguring the logic resources to fulfill desired application-supporting functionality. For established programming paradigms this is an even harder problem than heterogeneity, which by itself already demands for costly hardware-aware programming efforts: coping with dynamically changing platforms typically requires overloading the core program logic with associated control structures handling the heterogeneous, dynamic nature of the underlying architecture.

Unlike with previous technology improvements, the potential processing power of such future heterogeneous multicore architectures requires explicit use of parallel programming techniques, leading to a break with conventional approaches to program development and execution. Easing programmability of heterogeneous architectures and dealing with runtime reconfigurability is hence considered one of the Grand Challenges of Computer Engineering [5]. Typically, related approaches use a high level of abstraction, avoiding the need to specifically address the underlying hardware; this is left to a virtualization or runtime layer. A common drawback of most of these approaches is, however, tying the concept to a dedicated programming language, programming model, or virtual machine. In addition, none of these approaches take application requirements into consideration when performing the application-to-hardware (A2H) mapping. Disregarding these requirements is likely to break running applications dependent on the fulfilment of certain requirements such as latency, throughput, or accuracy issues.

We therefore propose a lightweight embrace-and-extend approach to the process of creating a hardware-independent application description and performing application-aware A2H mapping. Cornerstones of this approach are universal applicability, compatibility, and lightweight implementation. The approach is independent of programming language, programming model, and operating system. It provides a seamless and most compatible path for migrating from conventional programming to programming of heterogeneous and dynamic architectures without breaking existing program code, development tools, and infrastructures. This is achieved by careful extension of techniques already employed in modern OSes' runtime systems.

We like to start this paper with an overview over existing approaches for motivating our work, followed by an introduction of our concept and its exemplary implementation, demonstrating how careful embracing and extending the existing OS infrastructure leads to a lightweight, compatible approach to application programming and execution on parallel, heterogeneous systems. We prove that using such an approach achieves a seamless upgrade path from conventional to heterogeneous execution without breaking compatibility nor introducing measurable penalties in execution time.

2 Program Execution on Heterogeneous Parallel Systems

In order to harnessing the power of heterogeneous parallel architectures, infrastructures supporting program development for and execution on such systems must comply with the specific requirements of both aspects, parallelism and heterogeneity: they need to enable exploitation of thread-level parallelism as well as assignment of compute-intensive kernels to dedicated accelerator hardware, including configuration of this hardware if reconfigurable technology is used. This field is actively researched with a number of already existing and currently developed approaches.

A recent example of a heterogeneous system is the MOLEN reconfigurable processor [15]. It employs function-level granularity and offers dynamically alterable co-processor instructions. Its according development framework focuses on the generation of integrated applications consisting of a software description, design and integration of required accelerator hardware and its integration into the instruction set, as well as generating the according software tools and synthesis scripts. It is therefore plain application-centric and no specific runtime aspects and/or compatibility aspects are considered; ongoing work, however, targets OpenMP interoperability.

Several approaches ease application description for heterogeneous and parallel systems. They typically operate on function-level granularity rather than instruction level and typically feature an associated runtime system. The currently most prominent example is CUDA [6], targeting heterogeneous platforms consisting of a host machine running the basic control flow, and GPUs used as highly parallel FP accelerator units. CUDA features an extension to the C language dealing with partitioning and offloading compute kernels to the GPU. These GPUs employ a dedicated memory hierarchy and a hardware thread manager to cope with the high level of parallelism involved. CUDA requires only minimal changes on software level, basically rewriting accelerable functions using CUDA primitives, therefore providing a smooth upgrade-path leading to quick adoption by programmers. This concept is extended by the so-called OpenCU [11] approach, targeting a more flexible distribution of the workload between CPU and GPU, so that individual functions may run on either hardware. The commercial RapidMind [13] platform follows a similar concept, consisting of a gradual language extension (mainly providing a uniform datatype declaration) and an according runtime system. An application is written in a way that enables the framework to create binary representations to be executed on various types of computing nodes, including GPUs, IA32 CPUs, and the Cell BE architecture. A dedicated runtime layer ensures the application task distribution and interplay of the individual application tasks. The last three approaches do not specifically take application requirements into account, but solely rely on implicit declarations like providing correct data types or following a programmer-defined partitioning. However, the approaches demonstrate how a smooth upgrade-path may lead to quick adoption by programmers and integration into operating systems.

Focusing on dynamic exploitation of parallelism for improved use of parallel systems is Intel's *C for Throughput Computing* (Ct) [7] focuses on dynamic exploitation of parallelism by improving the use of parallel system through an automated, dynamic partitioning and orchestration of a running application. To enable such, Ct requires adopting a dedicated C++ library and adjusting the program code accordingly. This approach is a strong point for intelligent, self-partitioning runtime systems; however, in its current form it is not suitable for hardware-constraint systems. Neither does it address the fulfilment of application requirements.

The EXOCHI project [16] directly approaches the programming of heterogeneous platforms, featuring a C/C++ environment supporting creating a unified application description. This description is compiled into a so-called fat binary containing individual binary representations for a given heterogeneous system. Using a dedicated, OpenMP-derived runtime system this binary is then mapped to the individual cores. The Merge framework [9] extends the EXOCHI concept towards dynamic mapping of individual

application parts to the underlying heterogeneous architecture; it uses the *map-reduce* parallel programming language. EXOCHI/Merge provide an interesting way to achieve unified binaries and exploit parallelism, but again do not address specific application requirements. The approach is furthermore tied to a specific VM and, in case of Merge, requires the use of a dedicated programming language.

The recent Lime project [4, 1, 14] aims at a unified application description by extending the Java framework and focuses on removing the border between soft- and hardware: an application written in Java may be either executed in software on a JVM or transformed into a dedicated hardware description.. This most interesting approach again is based on a careful extension of an existing programming and execution infrastructure. It therefore is tied to Java and the JVM, which restricts the use of legacy code and also might impose a problem for certain hardware-constraint systems.

Of the existing approaches, only very few – typically vendor-provided and platform-restricted – approaches are so far used by a significant amount of programmers. What these approaches share is a moderate extension of existing programming methods e.g. by introducing certain keywords and data types. However, even with such moderate extensions the compatibility with existing code is broken. Other approaches even require adoption of unusual programming models and/or programming languages.

For our universally applicable approach, we therefore define the following cornerstones: compatibility and fulfilment of application requirements. Maintaining source-code and binary compatibility ensures a smooth upgrade path from conventional systems. Hence, the approach must neither rely on a dedicated programming language nor model. With respect to the specific requirements of parallel systems, compatibility and interoperability with commonly used parallel programming models such as OpenMP is mandatory. Also runtime compatibility must be ensured, i.e. using legacy binaries in new hardware-aware environments and vice versa. Such can be achieved by using a lightweight embrace-and-extend approach, i.e. the extension of already present system layers and clever exploitation of their properties. Strict fulfilment of specific requirements is mandatory for certain applications. disregarding these will lead to noticeable effects ranging from simple slowdown to breaking the application: this is most notable for all real-time applications such as real-time media streaming, transcoding, or securing, but also vital for certain numerical computations where e.g. certain minimum computing accuracy is required for individual program phases. We therefore require compatible ways to express application requirements and ensure their fulfilment. For legacy applications a mechanism is required ensuring same performance as experienced on these application's native systems. Existing approaches typically either completely disregard such requirements or are based on a worst-case scenario, the latter degrading the system's flexibility in task/thread-mapping and overall efficiency.

Our approach, to be presented in the following section, is specifically designed for fulfilling the above cornerstones. It puts strong focus on universal applicability, ranging from high-performance computing to embedded systems. It neither breaks existing code nor disables mixed-code execution and enables declaration and fulfilment of application requirements. As a result of the overall lightweight design the approach does not impose a execution time penalty compared to native execution.

3 A lightweight and universal approach to parallel and heterogeneous program execution

In order to achieve the required interoperability, compatibility, and lightweightness, we carefully extended and augmented existing concepts employed in modern operating systems in order to fulfill the specific requirements of describing and executing programs on a heterogeneous, dynamically configurable parallel system. Figure 1 shows the basic architecture of our concept spanning compiler, runtime and hardware domain.

Key part of this architecture is a guided function mapping process taking place during runtime: an application typically dissects into a control thread and potentially accelerable computing kernels. These kernels are mapped to suitable SW and HW implementation alternatives during runtime with according hardware reconfiguration taking place where required. This level of granularity is common to almost all existing approaches. In contrast to these, we however do not employ an additional runtime layer but rather embrace and extend the OS's own runtime system.

Modern OSes typically employ so-called runtime (or dynamic) linking, performing function resolution if non-statically linked library routines are called. The used binary formats support this dynamic linking process by dedicated data structures; an example

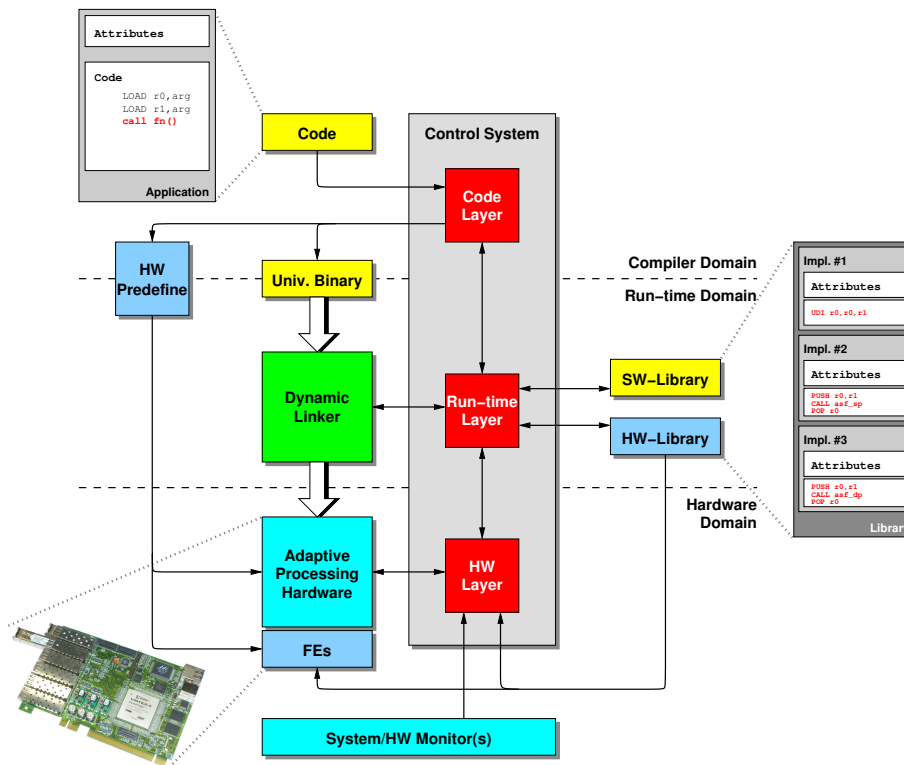


Fig. 1. Concept Outline

for this is the Executable and Linking Format (ELF): ELF executables employ several sections, some of which are dedicated to resolving of function symbols, most notably the so-called Global Offset Table (GOT). During runtime, these sections are searched for functions to be called and resolved to the resulting address. For later reference, this address is stored in the GOT to speed up later function calls. This process is called lazy linking. An obvious approach therefore is to extend this technique in order to enable dynamic re-linking of already resolved functions upon demand.

This approach, depicted in Figure 2 is ideal with respect to compatibility and interoperability as only the linking process is altered. Only the required switching logic needs to be added to the OS which, with modularized kernels, can be done using a loadable kernel module. This process is completely transparent to the programmer and the remaining parts of the OS. Being an extension to the existing linking process, the approach enables compatibility with existing legacy code. By design it is furthermore completely independent of compiler and programming language.

However, the GOT-based approach is not thread-safe: being part of the binary, the maintenance structures used for dynamic function resolution are only present per task, i.e. per running application instance. Hence, they offer no possibility to increase the resolution from application/task to thread level.

We therefore designed a second, thread-safe approach supporting dynamic function resolution by embedding the according maintenance information into the Task State Segment (TSS). This is possible, because the TSS – despite the fact that certain CPUs offer hardware support for TSS management – is entirely managed and maintained by the OS. This approach, dubbed Dynamic Linking System (DLS) and illustrated by Figure 3, employs so-called proxy functions in place of function calls. During runtime, this proxy is dynamically adjusted to the desired function implementation. The number of function pointer substitutions is only limited by system memory and address space, therefore providing a theoretically unlimited amount of function pointer substitutions, even with an overlapping set of functions and libraries.

Depending on whether runtime addition and removal of further dynamically mappable functions is required, this approach can be realized as two different implemen-

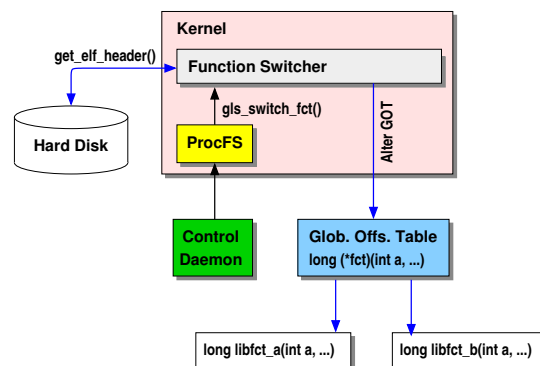


Fig. 2. Function-mapping using GOT manipulation

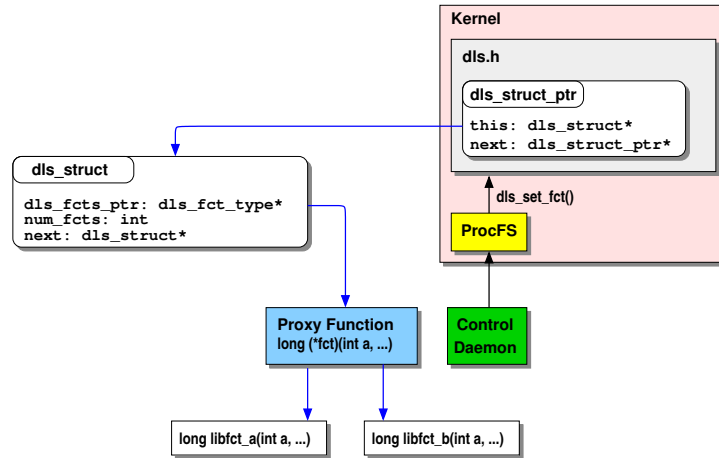


Fig. 3. Function-mapping using a TSS-based Proxy System

tations, resembling static (DLS-SL) and dynamic linking (DLS-DL) of conventional binaries. In the static approach, DLS employs only the dynamic mapping required for thread-safe function mapping, otherwise also a dynamic linking mechanism is used for function resolution. Because of the added functionality, the changes to the kernel are slightly more complex, therefore no easy module-based approach is possible, but the changes have to be made in the kernel source code.

3.1 Providing and Achieving Mapping Guidance

In order to not breaking application constraints, e.g. throughput or computation accuracy, the aforementioned mapping process *must* strictly adhere to application requirements. This requires mechanisms for providing these requirements on source-code and binary level, and a control interface by which the runtime mapping process is steered.

For the control interface we again embraced already existing techniques: targeting a Unix-based infrastructure for our test implementation, we integrated this control interface using the so-called proc filesystem [10] or procfs; procfs is a pseudo file-system used to access process information from the kernel and is available on most Unix and Linux environments. Through this interface not only guidance information can be provided for steering the mapping process in order to fulfill application requirements, but also specific function implementations may be added, removed, or selected during runtime. This interface gives the flexibility to either control the steering process manually, script-based, or through a dedicated control daemon.

These requirements can either be provided as independent control data, or, preferably be included into the corresponding application and library binaries by employing capabilities of current file formats. The aforementioned ELF format, for instance, enables transparent augmentation by additional sections. This ensures compatible execution, i.e. augmented binaries may be executed on standard systems and vice versa.

During runtime, the function resolver evaluates function call requirements against implementation capabilities and chooses from an (ideally) Pareto-optimal selection based on guidance information stored in the application binary and/or individual libraries.

Typically, a programmer would like to provide such guidance information during programming time. This is possible code using so-called pragmas. Pragmas are a method to provide additional information to an according compiler infrastructure, which is only extracted and processed if these compilers know about specific pragmas; otherwise, they do not affect the program generation process and therefore provide a compatible way of augmenting the application description with guidance information.

4 Implementation and Evaluation

Key parts of the described approach were implemented and evaluated on a Linux-based system, demonstrating the applicability and suitability of this approach. Our test setup comprises of a general-purpose multicore PC platform enhanced by a dedicated FPGA accelerator card comprising a Xilinx Virtex4-FX100, using state-of-the-art HyperTransport interconnection technology.

The software stack is depicted in Figure 4. Here, we see the interplay of the various software layers from application to kernel space down to hardware access. To properly separate these layers, we employ two OS-provided control interfaces which are inter-process communication (IPC) based on the aforementioned procs interface between the runtime part and the control daemon, and hardware device drivers to access the accelerator hardware by this daemon. For our test implementation we partitioned the accelerator’s resources in order to achieve a heterogeneous, parallel application accelerator. Hence, the logic resources are provided as 6 individually configurable and accessible slots attached to a static HT-based interface infrastructure [8]. Enabling a uniform, configuration independent interface, dedicated abstraction layers are provided in hardware.

With this platform, we could demonstrate that our solution does not show negative impact on application runtime for standard operation, i.e. when no function mapping occurs [2]. Extending these measurements, we furthermore quantified the latency involved with function switching as presented in Table 1. In comparison with set-up latencies for CUDA compute kernels and FPGA hardware reconfiguration as shown in Table 2 we can safely state that not only is our approach some orders of magnitudes, but also the cost for re-resolving a function pointer becomes invisible as function change time is dominated by the accelerator hardware setup costs.

Using dedicated test applications, we ensured proper functioning of our infrastructure [12, 3]. One test application for this setup was hardware-accelerated 3DES encryption [2]. Depending on the data payload size, either the software or hardware imple-

| Mechanism | Latency | Switching | Interface |
|-----------|-------------|-------------|-------------|
| DLS-SL | 1.6 μ s | 0.5 μ s | 1.1 μ s |
| DLS-DL | 4.8 μ s | 3.3 μ s | 1.5 μ s |

Table 1. Function switching latencies

| Setup Type | Latency | Factor |
|-------------------|--------------|---------|
| CUDA kernel init. | \sim 0.6 s | 100,000 |
| FPGA config. | \sim 10 ms | 2,000 |

Table 2. Accelerator setup latencies

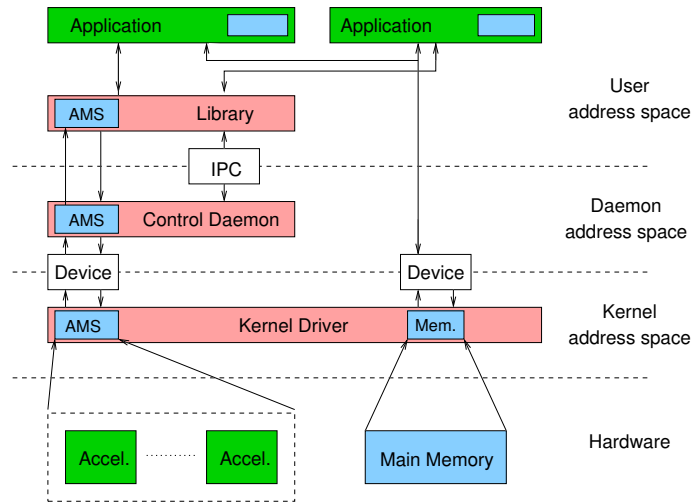


Fig. 4. Software Component Interplay

mentation was selected in order to achieve maximum performance. With this very test application, we furthermore successfully demonstrated compatibility with the OpenMP programming model.

5 Conclusion and Outlook

Starting with a requirements analysis and discussion of related approaches, we motivated an alternative, lightweight and most compatible concept addressing program generation and execution on heterogeneous, dynamically changing parallel systems.

By carefully extending existing concepts and techniques employed in modern OSes, our concept enables full compatibility with existing application binaries and libraries, showing no performance degradation compared to native execution. Furthermore, our approach enables full compatibility and interoperability with existing parallel programming models such as OpenMP or MPI. Exploiting the capabilities of modern binary formats, we ensured backwards compatibility with conventional runtime systems not performing any mapping process. We provide implicit and mechanisms to apply guidance information for steering the mapping process. If no such information is applied, a best-effort fall-back strategy is executed. The presented approach is by design language-agnostic and not tied to a specific infrastructure. The applicability and suitability of the concept was demonstrated by a test implementation based on the Linux operating system, using the ELF file format. This solution can easily be transferred to other Unix-based systems and other OSes employing similar runtime systems and binary formats.

As of now, the prototype implementation offers all basic building-blocks required for dynamic function mapping. Currently under development is an autonomous control daemon which automatically guides the mapping process with respect to present hardware resources and application requirements.

References

1. Amir Hormati and Manjunath Kudlur and David Bacon and Scott Mahlke and Rodric Rabbah. Optimus: Efficient Realization of Streaming Applications on FPGAs. In *Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Oct 2008.
2. 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, 22nd International Conference (LNCS 5455)*. Springer, March 2009.
3. Rainer Buchty, David Kramer, Fabian Nowak, and Wolfgang Karl. A Seamless Virtualization Approach for Transparent Dynamical Function Mapping targeting Heterogeneous and Reconfigurable Systems. In *ARC2009 – Proceedings of the 5th International Workshop on Applied Reconfigurable Computing (LNCS 5453)*. Springer, March 2009.
4. David F. Bacon and Rodric Rabbah. Liquid Metal (Lime). Aug 2008. http://domino.research.ibm.com/comm/research_projects.nsf/pages/liquidmetal.index.html.
5. Theo Ungerer et. al. *Grand Challenges der Technischen Informatik*. VDE Verband der Elektrotechnik, Elektronik, Informationstechnik e.V., 2008.
6. Tom R. Halfhill. Parallel processing with CUDA. In *Microprocessor Report*, Jan 2008.
7. Intel Corp. Ct: C for Throughput Computing. 2007-2009. <http://techresearch.intel.com/articles/Tera-Scale/1514.htm>.
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 (WHTRA 2009)*. Universitätsbibliothek Heidelberg, February 2009. ISBN 978-3-00-027249-3.
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. M. Tim Jones. Access the Linux kernel using the /proc filesystem. In *IBM developerWorks*, 2006. <http://www.ibm.com/developerworks/library/l-proc.html>.
11. Aaftab Munshi and Jeremy Sandmel. Data Parallel Computing on Multiple Processors. April 2008. Patent Publication No. US2008004648, <http://www.wipo.int/>.
12. Fabian Nowak, Rainer Buchty, David Kramer, and Wolfgang Karl. Exploiting the HTX-Board as a Coprocessor for Exact Arithmetics. In *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA 2009)*. Universitätsbibliothek Heidelberg, February 2009. ISBN 978-3-00-027249-3.
13. RapidMind, Inc. RapidMind Multi-Core Development Platform. 2008. <http://www.rapidmind.net/>.
14. Shan 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)*, Jul 2008.
15. S. Vassiliadis, S. Wong, and S. D. Cotozana. The MOLEN μ -coded Processor. In *11th International Conference on Field-Programmable Logic and Applications (FPL)*, Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147, August 2001.
16. 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.