

# A Light-weight Approach to Dynamical Runtime Linking Supporting Heterogenous, Parallel, and Reconfigurable Architectures

Rainer Buchty, David Kramer, Mario Kicherer, Wolfgang Karl

Universität Karlsruhe (TH)

Institut für Technische Informatik, Lehrstuhl für Rechnerarchitektur

76128 Karlsruhe, Germany

e-Mail: {buchty|kramer|kicherer|karl}@ira.uka.de

**Abstract.** When targeting hardware accelerators and reconfigurable processing units, the question of programmability arises, i.e., how different implementations of individual, configuration-specific functions are provided. Conventionally, this is resolved either at compilation time when a specific hardware environment is targeted, by initialization routines at program start, or decision trees at run-time. However, this technique is hardly applicable to dynamically changing architectures; furthermore, these approaches show conceptual drawbacks such as requiring access to source code and reconfiguration, as well as overloading the code with reconfiguration-related control routines.

We therefore present a low-overhead technique enabling on-demand switching of individual functions; basically, this technique can be applied in two different manners. We will discuss the benefits of the individual implementations and show how both approaches can be used to establish code compatibility between different heterogeneous, reconfigurable, and parallel architectures. Further we will show, that both approaches are exposing an insignificant overhead.

## 1 Introduction and Motivation

While Moore's Law is still being valid, further increase in integration density due to technological limitations does not lead to faster uniprocessors, but rather multicore processors and significant advances in FPGA technology, offering the possibility to create heterogeneous architectures offering a mix of general-purpose and specialized processing units, potentially including dynamically reconfigurable hardware accelerators. A current example are so-called Platform FPGAs [17] which basically combine reconfigurable logic with conventional processor cores, dedicated communication hardware, and additional circuitry such as on-chip memory or hardware multipliers.

Such advances in technology will lead to massively parallel architectures. Following the forecasts, these will likely be heterogeneous, i.e. architectures where the individual cores might show different characteristics. Considering hybrid approaches as e.g. outlined in the context of terascale computing [6] or the AMD Fusion concept [1] it can be therefore safely predicted that future architectures will not only feature

heterogeneity but also will employ more and more reconfigurability to enable maximum use of the silicon area. Two academic research platforms illustrate such architectures: the Molen processor architecture [14] features a fixed processor core enhanced by user-defined commands executed on a reconfigurable hardware infrastructure. The Digital On-demand Computing Organism (DodOrg) [3] is an example of a dynamically reconfigurable system architecture. It specifically addresses the aspects of parallel heterogeneous systems. DodOrg comprises an array of individual processing cells, connected using peer-to-peer networking. Each processing cell is reconfigurable in terms of function and connectivity, therefore not only supporting HW-based application acceleration and memory resources, but also required grouping and communication of cooperating processing cells. As demonstrated, future platforms extend the flexibility beyond task/node assignment into the hardware layer, therefore introducing a new class of problems with regard to application programming and execution.

The key problem of such architectures is exploiting their full potential. Here, the application itself must benefit from the architectures' features, e.g. by showing significant program phase behavior and/or requiring use of dedicated application accelerators. This must be taken into account by the application programmer and being supported by the runtime environment. A unified code representation is required, supporting all present computation nodes regardless of node type. Typically, therefore only a potentially accelerable computing routine is present as multiple, node-specific instances. A corresponding representation is used depending on the assigned computing node with the selection logic, i.e. which representation to use, being typically coded into the application itself. Reconfigurable hardware further increases this problem as certain hardware configurations might not be available at program development time.

As part of our effort on programming tools and environment targeting high-performance heterogeneous and reconfigurable computing, we therefore developed a lightweight approach enabling dynamic alteration of function calls at runtime, achieving dynamic adaptation of a running program to changing processing hardware. This addresses the major issue of program development and execution on heterogeneous and reconfigurable platforms: using a runtime layer, performing appropriate function mapping according to the current system configuration and provided node-specific implementations of individual functions, re-enables the programmer to fully concentrate on the application itself, not the side effects of heterogeneity or reconfigurability issues.

Different from existing approaches, we implemented such a runtime layer as a lightweight extension to an existing OS runtime system, showing no measurable overhead during typical operation, as opposed to heavy-weight virtual machine approaches. As a beneficial side effect, this approach offers a smooth upgrade path from static to dynamic systems is provided, and even mixed execution of adaptable and static application is possible.

The following paper therefore is structured as follows: first, we provide an overview over related work, highlighting differences and benefits of our approach. We will then introduce our concept in detail and discuss potential implementation alternatives, outlining the benefits of the chosen alternative. This is followed by the evaluation of the individual approaches, showing the overall feasibility and integrateability. The paper is concluded with a summary and outlook.

## 2 Related Work

One example of running applications on a heterogeneous and potentially changing hardware was recently presented by Stanford's Pervasive Parallel Lab (PPL) [11]. Basically extending the Java Virtual Machine approach [9], the PPL concept features a parallel object language to be executed on a common parallel runtime system, mapping this language onto the respective computing nodes using either interpretation or JIT techniques (including binary optimization). This approach requires applications to be rewritten from scratch to fit into the framework and its specific runtime system.

Breaking the border between hard- and software is the focus of IBM's long-term research project Lime [2, 12]. Lime uses an uniform programming language and an associated runtime system to enable that each part of the system can dynamically change between hard- and software. Similarly to the PPL concept, also Lime targets Java applications to be dynamically translated for co-execution on general-purpose processors and reconfigurable logic.

Based on the concept of cache-only memory architectures (COMAs) is the SDVM [5]. COMAs feature a distributed shared memory in which data migrates automatically to the computing nodes where it is needed. The SDVM extends this concept in a way that both, data and instructions, can transparently migrate, enabling automatic data and code migration within an SDVM-equipped parallel system. The SDVM<sup>R</sup> [7] furthermore adds so-called configware migration to this concept targeting transparent automatic hardware reconfiguration. Both SDVM incarnations require special program treatment and, moreover, a dedicated runtime layer.

The programming of heterogeneous parallel system is the focus of EXOCHI [16]. EXOCHI consists of the *Exoskeleton Sequencer* EXO and the C/C++-programming environment *C for Heterogeneous Integration* CHI. CHI extends the OpenMP pragma regarding heterogeneity. After compilation, a so called *fat binary* will be created, containing the whole program code for the given heterogeneous system, which can be mapped to current hardware configuration via the CHI-own runtime system. EXO permits this mapping by managing a heterogeneous system as ISA-based MIMD-resources.

A similar approach is presented by the Merge-Framework [8]. Merge is based upon the EXO representation used in EXOCHI and aims through a runtime system in combination with a library model at a complete dynamization of the execution on a heterogeneous system. To permit the proper use of available parallelism, Merge uses *map-reduce* as its input language.

Program generation and application execution for heterogeneous architectures consisting of CPUs and GPUs is addressed by several projects. Within the scope of this paper, a most prominent example is the Khronos OpenCL approach featuring program generation for computation offloading to either CPU or GPU [4]. OpenCL defines a programming model supporting data- and thread-parallelism with the according runtime layer dealing with thread management and deploying in GPU-accelerated systems.

From the cited work our approach differs in several ways: it was designed in a most compatible way, requiring minimal to none changes on application level. It offers a versatile control interface, enabling the combination with existing programming models, resource and configuration managers, and is therefore applicable to a wide range of heterogeneous and reconfigurable management and programming approaches. It is

particularly light-weight, not requiring an additional virtualization or abstraction layer, but rather being an extension of the OS's runtime system, making it possible to perform dynamic runtime function resolution, i.e. map individual functions of an application to one of several provided implementation alternatives. Targeting parallel systems, our approach also supports the OpenMP programming model. It therefore provides smooth upgrade path from conventional to reconfigurable systems.

To address the different viewpoints, the following section contains an introduction to our basic approach and discusses possible implementation methods, their specific properties, and use within an OpenMP environment.

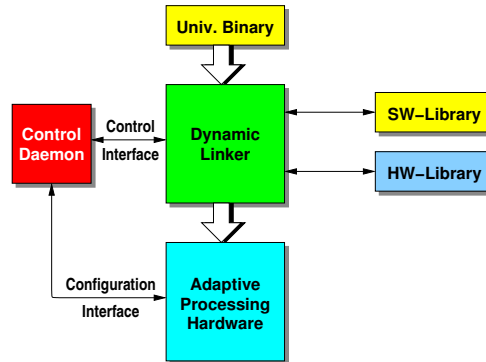
### **3 Runtime-reconfigurable function calls**

Our approach was explicitly designed with easy integrateability into existing systems in mind. Therefore, compatibility and lightweightness were topmost design criteria. No extra runtime layer shall be required, no dedicated programming language or special binary format mandatory. In contrast to existing approaches, we therefore chose to realize our approach as an extension of the existing OS runtime system. As our implementation example, we chose Linux, but our approach is generally applicable to any Unix- or Unix-like system.

The underlying concept, depicted in Figure 1, consists of an adaptive processing hardware, on which a universal, i.e. configuration-independent, binary is executed containing the control thread and computing functions to be dynamically mapped to the computing hardware. Depending on the current system configuration this may result in pure software, hardware, or hybrid implementations of the affected functions.

Such function resolution can be achieved by substituting the actual function call by a function pointer with the same name, arguments, and return value which during runtime gets resolved to the desired implementation. This function resolution ideally is individually controllable per thread and offers sufficient security against potentially malicious interference. That way, application code and reconfigurability issues can be strictly separated. A dynamic linker therefore is the central element of this function resolution. It is responsible for resolving configuration-specific functions and mapping them to the desired implementation during runtime, including on-demand retriggering of the dynamic linking process due to hardware changes. This is different to conventional approaches for dynamic runtime linking, where a function resolution takes only place once, or compiler-based decision trees which include the control logic into the application program.

To accomplish this, we introduce two different approaches to runtime reconfigurable function resolution. Our first approach is using so-called proxy functions in which any reconfigurable function is represented by its proxy; during runtime, the proxy is resolved to the desired implementation. The second approach performs dynamic changes within the Global Object Table (GOT) using the ELF binary format's lazy linking technique [13]. Both approaches have different capabilities, benefits, and drawbacks. In the following we will therefore present alternatives and discuss their individual advantages and disadvantages.



**Fig. 1.** Application Execution on Dynamically Reconfigurable Systems

Both approaches require steering the dynamic linking process, controlling which functions may be mapped to what implementations without breaking an application’s requirements. This is typically the task of a dedicated control daemon which can be considered an additional system service, triggering hardware configurations through a configuration interface and communicating with the linker through a dedicated control interface. Via this interface, specific function implementations may be added, removed, or selected during runtime. For maximal flexibility and interfaceability with a wide variety of control instances, we implemented this control interface using the process file system or *procfs* [10]; *procfs* is a pseudo file-system used to access process information from the kernel and is available on most Unix and Linux environments.

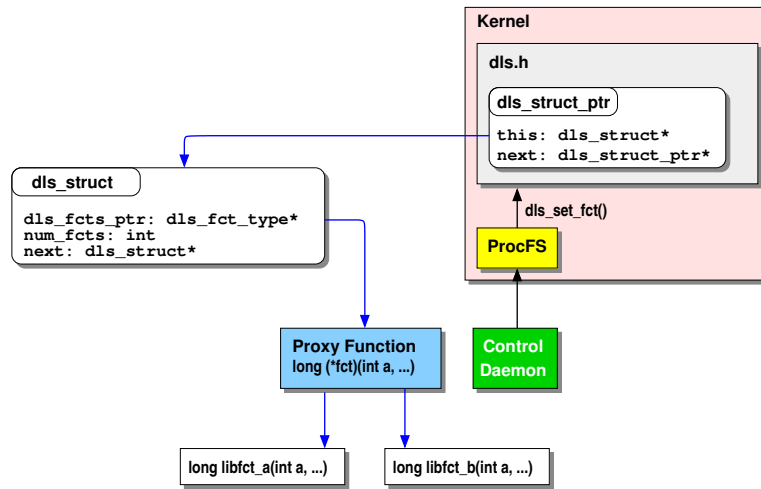
### 3.1 TSS-based Dynamic Linking System (DLS)

Our first approach, called DLS (Dynamic Linking System), uses a proxy function concept depicted in Figure 2 implemented by enriching a task-management kernel structure called *task-state segment* (TSS) [15]<sup>1</sup>. The benefit of this approach is fine-grain control on a per-thread base. In theory, an unlimited amount of function pointer substitutions is possible, even with an overlapping set of functions and libraries.

This approach requires minor changes to the kernel source code, basically to support task-associated management structures for the dynamic functions, as well as modifications within the program source code: using this approach, an application program must declare affected functions as runtime reconfigurable and perform necessary bookkeeping operations. To accomplish this, 3 dedicated functions are provided by the runtime system which the programmer needs to use to initialize and register required data structures and to provide pointers to function implementations.

This registration process is illustrated by Figure 3 where a function is first defined as runtime reconfigurable and an according management structure `dls_struct` is pro-

<sup>1</sup> Please note that several operating systems, including Linux, do rely on software task switching as opposed to hardware task switching mechanisms provided by some CPUs. The TSS is therefore not bound to a specific hardware-dependent layout such as e.g. the IA32 TSS.



**Fig. 2.** DLS: TSS-based Proxy System

```

dls.define(testfct, int, char *str, int len);

dls_init(testfct);
dls_add(testfct, "libs/liba.so", testfcta);
dls_add(testfct, "libs/libb.so", testfctb);

```

**Fig. 3.** Initialization and Registration Example (see text)

vided. In further steps this structure is initialized and registered to the kernel, and individual implementations of this very functions are added.

Upon program execution, the proxy function pointer is then resolved to the currently selected implementation; this selection can be changed on-demand via *procfs* by triggering a dedicated switch logic. If the switch logic receives the command to change a function pointer, it will search for the corresponding `dls_struct` in the affected thread's linked list and change the function pointer to the desired function to be called. The previous value of this function pointer is stored separately for safe library unloading.

This concept aims at ease of implementation from a programmer's and runtime perspective. However, it does not come free of charge but requires certain kernel modifications: the task management structure `task_struct` needs to be expanded by a pointer to a linked list of function management structures `dls_struct_ptr`, which holds a pointer to the corresponding function resolving structure `dls_struct`. Furthermore, an exit function for freeing `dls_struct_ptr` is added.

Considering only single-threaded tasks, the unloading of libraries in this approach is safe. One possibility would be the counting of how many function pointers are connected with a library. So every library can be unloaded from address space if its usage counter equals zero. But even forced or speculative unloading of libraries is safe. If a function pointer still has a reference to a function in such a library and this current

function is called, the fix-up function gets activated and reloads the library. In worst case therefore just an unnecessary time overhead is created.

In the multi-threaded case, safe unloading can only be performed automatically under certain circumstances. Simple counting of enter and return events is not always sufficient – a thread being inside a function can be destroyed before it returns, so the programmer has to take care of decreasing the counter.

This approach may be implemented in two ways, Static Linking (DLS-SL) and Dynamic Linking (DLS-DL). If every function implementation is known at compile-time, the static DLS-SL approach can be used. Because of the known function addresses, there is no need for a dynamic loader and proper steering functions, therefore reducing the switching overhead. If function implementations should be added or removed dynamically during program execution, the dynamic linking approach DLS-DL must be used. Using this implementation, a dynamic loader and a so-called fix-up function are necessary: the fix-up function uses the dynamic loader to load and unload function implementations and performs function resolution by changing the function pointer to the appropriate function symbol.

Without any switching, both implementation alternatives show a minimal runtime overhead at worst (see Section 4) as also the standard function resolution taking place during execution of dynamically linked functions requires an indirection: the according function is searched in an object table and then executed. The only difference in this approach is the use of a different search function taking place on an own, task-dependent management list. Due to the fix-up function and the dynamic loader, the switching overhead for DLS-DL is about 100% higher compared to DLS-SL.

While function switching is ideally steered from outside for proper separation of program execution and configuration control, the DLS approach is also capable of triggering the switching logic from inside the application, further decreasing the switching overhead as no *procfs* call to the kernel is required.

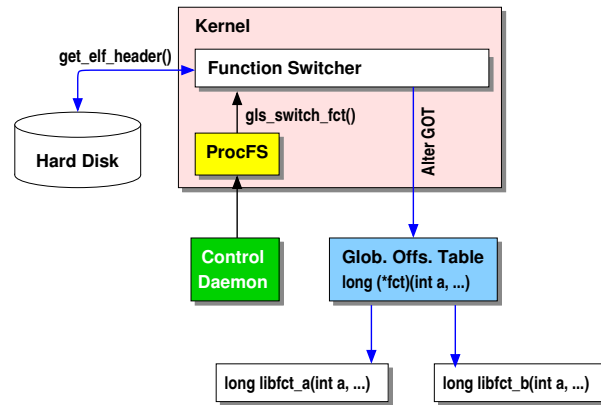
In our current implementation, dedicated initialization and registering routines are required. We are currently evaluating possibilities to also do this from outside the program code using *procfs* calls.

### 3.2 GOT-based Linking System (GLS)

The GLS approach utilizes the lazy-linking technique used in the ELF binary format. The advantage of this approach is its independence of both, compiler and used programming language, as it purely operates on the ELF file format. Also, no source code changes are required, making it ideal for binary-only distributions and approved legacy code where code changes would lead to costly re-approval.

This approach is therefore tied to the ELF format by design. Porting to other file formats with similar features is potentially feasible, though. Compared with DLS, GLS needs just smallest changes to kernel source code: only the support for accessing the virtual memory of other processes needs to be activated. The switching logic itself can be realized as a kernel module which can be loaded at runtime.

Furthermore, this method only allows coarse-grained decisions, i.e. the redirection of function calls are effective for the whole program instance, therefore providing no way to individually adjust functions for different threads like in DLS.



**Fig. 4.** GLS: GOT-Manipulation

Lazy linking means that a symbol will be resolved when it is first used and not at the beginning of the execution when the libraries are loaded. To use this method, we alter the name of the symbol at runtime and reset according data structures in a way that the next time the symbol is used, the dynamic linker is called and resolves the symbol with the new name.

To explain the work mechanism, we will first present the anatomy of an ELF executable. These files contain several sections, of which the following are of special interest: the *.got* section contains the address of the symbol in user space or the associated *.plt* entry pointing towards the procedure look-up table (PLT), which in term contains code that calls the dynamic linker. The *.dynsym* section contains some details about the symbols, e.g. the address of a symbol name. These symbol names are provided by the *.dynstr* section. Finally, the *.rel.plt* section among other information contains the address of the corresponding GOT entry of the symbol.

During runtime, the kernel has to take the following steps: it will first iterate through the *.rel.plt* table and calculate the address of a symbol name based on associated entries in *.dynsym* and *.dynstr*. When a match is found, then the symbol in the *.dynstr* table is changed. For current limitations of our implementation, the new name must not be longer than the existing name. Finally, the entry in *.got* is set to the associated *.plt* entry. This is illustrated by Figure 4.

### 3.3 Multi-threading Considerations

In order to be suitable for multi- and many-core architectures, the support of the OpenMP programming model is mandatory.

DLS is an ideal candidate for use in combination with OpenMP as it supports function redirection on a per-thread base, i.e. allows that each thread can execute a different implementation of the same function. However, in addition to the modifications mentioned in Section 3.1, some further but minor modifications to an application's source code are required: in order to properly identify the individual threads, the function call

itself must be extended by the ID of the calling thread so that each thread is provided a private function pointer.

As mentioned above, GLS only allows function redirection on a per-task base, therefore only OpenMP applications with different workloads for each thread, e.g. using the `sections-construct`, can take advantage of GLS. In this case no additional modifications are necessary, neither to kernel nor application source code.

## 4 Evaluation

Our approach follows a low-overhead design. In this section we present measurement numbers proving that during normal operation our approach does not implicate any additional overhead. Such does only occur for function switching, where numbers are presented showing how this overhead distributes on the function switching itself and the overhead induced by the control interface.

For our evaluation we used an AMD Opteron 870 system equipped with 2 cores and 2 GB main memory, running Ubuntu Linux with a patched 2.6.20-kernel, providing the required alterations described in Section 3.

Aim of this evaluation is to determine the overhead of our runtime system and how the control interface and function switching contributes to the measured overhead. We therefore explicitly chose a minimal application calling a simple function `return a+b;` provided by two different libraries. That way, no side effects from configuring hardware accelerators, peripheral communication or function workload interfere with the measurements, exposing the runtime system's overhead for control, function switching, and function resolution. The measurements were conducted as follows: for each configuration, individual batches of 10,000,000 function calls per batch were performed. The batch runtime was measured using the Unix `time` command and the average runtime was calculated based on the measured times of 20 runs.

### 4.1 Single-threaded Execution

Without any switching, the runtime of GLS is identical to normal program execution as depicted in Table 1(a), because no symbol needs to be changed and both binaries are identical. This is confirmed by our measurements. It is also shown that no measurable overhead exists for the DLS-SL and DLS-DL approaches during normal execution without function switching, i.e. with no function resolution occurring.

To measure the theoretical maximum overhead, we used the same test application, but this time we forced a function switch each function call. It must be noted that this is a pure stress-test example and does in no way reflect ordinary use: typically, function switching will occur at most every few seconds as dictated by the hardware reconfiguration time. Under such conditions, function switching will have no measurable impact to the overall execution time and our approach will show the same performance as the OS's native, unaltered runtime system.

For the stress test, we see an overhead of 1.72 to 3.22 for the individual implementations as shown in Table 1(b). These numbers include both, overhead for function

switching and communication through *procs*. Naturally, DLS-SL shows better performance due to lower overhead compared to the dynamic approaches DLS-DL and GLS. The difference between GLS and DLS-DL results from the dynamic (un-)loading of the libraries after each function call: for GLS, every library function remains in the memory while for DLS-DL the libraries may be evicted from main memory.

The results in Table 1(c) show the overhead purely related to function resolution, excluding communication overhead. The overhead of the DLS-SL approach is therefore, as expected, negligible, ranging in the area of about 1%. The overhead of DLS-DL decreases by approx. 70% from 3.22 (Table 1(b)) to about 2.24, clearly exposing the influence of the external control interface and the pure switching and resolution overhead.

## 4.2 Multi-threaded Execution (OpenMP)

To measure the overhead for OpenMP applications, we used the same test application, but this time the reconfigurable function was called from inside a parallel OpenMP region. The iterations were equally distributed among each thread. Unless marked otherwise, we measured the overhead running 10 threads, using the DLS-SL approach.

We first determined the general overhead related to thread-dependent function resolution and switching. Tables 2 and 3 basically show the same behavior as in the single-threaded case. Without any switching occurring, the overhead is negligible. The switching overhead in the OpenMP case is in the same range as for the single-threaded case.

To demonstrate that, as expected from the implementation, the overhead of our solution is independent of thread number and number of function alternatives per thread,

(a) w/o Switching					(b) Switching Stress Test (controlled externally)				(c) Switching Stress Test (in-application)					
	min	avg.	max	Ovhd.		min	avg.	max	Ovhd.		min	avg.	max	Ovhd.
(1) <b>native</b>	21.26s	21.60s	21.91s	–	(1)	21.26s	21.60s	21.91s	–	(1)	21.26s	21.60s	21.91s	–
(2) <b>GLS</b>	21.26s	21.60s	21.91s	0	(2)	60.86s	63.22s	65.58s	2.93	(2)	n/a	n/a	n/a	n/a
(3) <b>DLS-DL</b>	21.08s	21.54s	21.88s	~0	(3)	66.88s	69.60s	72.41s	3.22	(3)	47.33s	48.41s	49.35s	2.24
(4) <b>DLS-SL</b>	21.06s	21.57s	21.94s	~0	(4)	35.20s	37.20s	39.40s	1.72	(4)	21.03s	21.85s	22.66s	1.01

**Table 1.** Runtime System Overhead Measurements (see text for interpretation)

	min	avg.	max	Ovhd.
<b>w/o</b>	24.09s	24.88s	25.84s	–
<b>DLS-SL</b>	25.88s	26.53s	28.26s	1.06

**Table 2.** OpenMP runtimes w/o switching

	min	avg.	max	Ovhd.
<b>w/o</b>	24.09s	24.88s	25.84s	–
<b>DLS-SL</b>	36.32s	38.36s	41.87s	1.54

**Table 3.** OpenMP Switching Stress Test

#Threads	1	5	10	20
<b>DLS-SL</b>	35.93s	39.38s	38.36s	38.29s

**Table 4.** Thread-related Overhead

#Functions	2	4	8	16
<b>DLS-SL</b>	37.28s	38.10s	37.46s	37.76s

**Table 5.** Functions-related Overhead

we tested the influence on runtime of both alternatives. According to the results shown in Table 4 the caused overhead is independent of the number of threads; the deviations in the presented numbers are solely related to OpenMP scheduling overhead and not caused by our implementation. Table 5 proves that the overhead is independent of the number of function implementations and therefore fixed within measuring accuracy.

## 5 Conclusion and Outlook

Heterogeneous, dynamic systems call for methods reflecting the required dynamics also in software. Conventional techniques such as on-demand compilation or explicit code switches are not sufficient for dynamically changing systems, especially when reconfigurable resources are shared among several individual tasks. In addition, such solutions require source code access to adjust code generation to a given configuration, which especially in commercial environments is not always the case.

We therefore proposed light-weight methods for on-demand change of function during runtime. Ideally, these methods do not require application source code access and only minimal changes on OS level. Additionally, they adhere to requirements of multi- and manycore architectures supporting existing parallel programming models. For this, we presented two implementation alternatives, DLS and GLS, based on the task-state segment (TSS) using so-called proxy functions and the ELF binary format's object table (GOT). The latter does not require any changes on application and can be implemented as a kernel module. By design, it however is tied to the ELF format and does not enable easy per-thread but rather per-instance control. DLS, in term, allows per-thread control but – in the current implementation – requires slight code changes to register and initialize reconfigurable functions. Both approaches require changes at kernel level for enabling reconfigurability and according control interfaces.

With our test case we were able to prove the low-overhead claim, showing that overhead only takes place in case of function switching, and how this overhead is dividing into the control part and the actual function switching and resolving. A measurable amount of overhead is only generated for purely synthetic tests consisting of an empty function body and constant change of that function pointer; for real-world application examples it can be therefore safely assumed that our approach does only produce negligible (if at all measurable) overhead during program execution. We furthermore demonstrated the applicability of our approach to OpenMP, making our approach suitable for both, conventional multitasking and multi-threaded parallel systems.

We are currently refining this runtime system with respect to increased transparency and security aspects especially suiting the needs of multithreaded applications. We recently demonstrated the general usability with a heterogeneous processing case study employing hardware acceleration. As of now, a case study from the field of numerical mathematics is being performed, exhibiting distinct program phases, therefore requiring and benefitting from dynamic function resolution.

## References

1. Advanced Micro Devices. AMD Fusion Whitepaper. [http://www.amd.com/us/MarketingDownloads/AMD\\_fusion\\_Whitepaper.pdf](http://www.amd.com/us/MarketingDownloads/AMD_fusion_Whitepaper.pdf).

2. 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. to appear.
3. Jürgen Becker, Kurt Brändle, Uwe Brinkschulte, Jörg Henkel, Wolfgang Karl, Thorsten Köster, Michael Wenz, and Heinz Wörn. Digital On-Demand Computing Organism for Real-Time Systems. In Wolfgang Karl, Jügen Becker, Karl-Erwin Großpietsch, Christian Hochberger, and Erik Maehle, editors, *Workshop Proceedings of the 19th International Conference on Architecture of Computing Systems (LNI P81)*, pages 230–245, March 2006.
4. Khronos Group. Khronos OpenCL API Registry. December 2008. <http://www.khronos.org/registry/cl/>.
5. Jan Haase, Frank Eschmann, Bernd Klauer, and Klaus Waldschmidt. The SDVM: A Self Distributing Virtual Machine for computer clusters. In *Organic and Pervasive Computing – ARCS 2004, International Conference on Architecture of Computing Systems*, volume 2981 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.
6. Jim Held, Jerry Bautista, and Sean Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. *Research at Intel Whitepaper*, 2006. [http://download.intel.com/research/platform/terascale/terascale\\_overview\\_paper.pdf](http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf).
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. 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.
9. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Sun Microsystems, 1999. ISBN 978-0201432947, <http://java.sun.com/docs/books/jvms/>.
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. Kunle Olukotun et al. Towards Pervasive Parallelism. In *Barcelona Multicore Workshop (BMW2008)*, Jun 2008. <http://ppl.stanford.edu/wiki/images/9/93/PPL.pdf>.
12. 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.
13. The Santa Cruz Operation, Inc. System V Application Binary Interface (Edition 4.1). 1997. <http://www.caldera.com/developers/devspecs/gabi41.pdf>.
14. S. Vassiliadis, 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.
15. Vikram Shukla. Explore the Linux memory model. In *IBM developerWorks*, 2006. <http://www.ibm.com/developerworks/linux/library/l-memmod/>.
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.
17. Xilinx, Inc. Virtex<sup>TM</sup> Family FPGAs. 2008. [http://www.xilinx.com/products/silicon\\_solutions/fpgas/virtex/index.htm](http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/index.htm).