A

Self-awareness as a Model for Designing and Operating Heterogeneous Multicores

ANDREAS AGNE, MARKUS HAPPE, ACHIM LÖSCH, CHRISTIAN PLESSL, and MARCO PLATZNER, University of Paderborn

Self-aware computing is a paradigm for structuring and simplifying the design and operation of computing systems that face unprecedented levels of system dynamics and thus require novel forms of adaptivity. The generality of the paradigm makes it applicable to many types of computing systems and, previously, researchers started to introduce concepts of self-awareness to multicore architectures. In our work we build on a recent reference architectural framework as a model for self-aware computing and instantiate it for an FPGA-based heterogeneous multicore running the ReconOS reconfigurable architecture and operating system. After presenting the model for self-aware computing and ReconOS, we demonstrate with a case study how a multicore application built on the principle of self-awareness, autonomously adapts to changes in the workload and system state. Our work shows that the reference architectural framework as a model for self-aware computing can be practically applied and allows us to structure and simplify the design process, which is essential for designing complex future computing systems.

Categories and Subject Descriptors: D.4 [**Operating Systems**]; C.1.3 [**Other Architectural Styles**]: heterogeneous (hybrid) systems; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: self-aware computing, adaptive system, reconfigurable computing, multi-core

ACM Reference Format:
Agne, A., Happe, M., Lösch, A., Plessl, C. and Platzner, M., Self-awareness as a Model for Managing Heterogeneous Multicores. In press. ACM Trans. Reconfig. Technol. Syst. V, N, Article A (January YYYY), 18 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Over the last decade researchers have been proposing and investigating the construction of systems with so-called self-* properties [IBM 2003; Schmeck et al. 2011]. The self in self-* refers to the capability of a system to modify its own behavior or structure without any external control in reaction to or even in anticipation of system dynamics [Serugendo et al. 2011]. System dynamics can be caused by changes in the system itself or by events external to the system. There are many instantiations of self-* such as self-adaptive, self-optimizing, self-coordinating and self-healing, and the appeal of

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901), the International Graduate School of Dynamic Intelligent Systems, and the European Union Seventh Framework Programme under grant agreement n°257906 (EPiCS).
Author's addresses: A. Agne, M. Happe, A. Lösch, C. Plessl and M. Platzner, Department of Computer Science, University of Paderborn.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Published in: *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*
© ACM 2014, DOI: 10.1145/2617596

self-* properties has fueled research fields such as self-organizing systems [Serugendo et al. 2011], multi-agent systems [Wooldridge 2009], autonomic computing [Sterritt and Hinchey 2010], and organic computing [Schmeck et al. 2011]. More recently and in a sense on top of the evolution of self-* systems, we find systems attributed with the characteristics of being *self-aware* [Agarwal et al. 2009].

The motivation for looking into self-* computing systems is two-fold: First, we can observe a rapid increase of complexity when designing and operating computing systems and applications. Nowadays, many application domains exhibit quite divergent requirements with respect to functionality and flexibility, performance, resource usage and costs, reliability and safety, and security. Distributed systems grow in the numbers and heterogeneity of nodes and must be able to cope with an increasing level of dynamics. The network topology and the collective resources of the distributed system can vary strongly during runtime since nodes may be mobile, break, run out of battery, or generally leave and enter the network. The second motivation is that compute nodes evolve towards parallel and heterogeneous architectures to realize performance gains while minimizing their power consumption. Progress in nano-electronics allows us to integrate more and more functionality on a single compute node, but at the same time requires us to deal with increasing numbers of faulty and unreliable components. Self-* approaches strive to embed higher levels of (computational) intelligence into the systems and, thus, to master complex system design and operation.

In our work we focus on heterogeneous multicores implemented with field-programmable gate arrays (FPGAs) and aim at developing models, architectures and programming environments that allow for creating self-aware compute nodes. To this end, we use a previously published reference architectural framework [Becker et al. 2011] as a model for managing heterogeneous multicores. In this paper, we discuss this model and contrast it with related and previous approaches in self-aware computing systems. Our multicore implementation is based on ReconOS [Lübbers and Platzner 2009], a heterogeneous multicore architecture and programming environment which enables the creation of compute nodes according to the discussed model of self-awareness. While many works have shown adaptivity aspects in multicores [Coskun et al. 2008; Ebi et al. 2009; Mulas et al. 2009], the novel feature of our work is that with ReconOS and an FPGA-based multicore architecture we are able to adapt the number and *type* (software, hardware) of cores involved in executing an application during runtime. In the longer term we envision future self-aware compute nodes able to autonomously adapt to external changes in the workload and quality of service requirements but also to internal changes such as thermal problems and failures, and even anticipate such changes through modeling of the system and environment and through online learning techniques.

The paper is structured as follows: Section 2 reviews related approaches in self-* systems, in particular self-aware systems, and introduces to the reference architectural framework. Section 3 gives an overview over ReconOS. In Section 4, we report on experiments with heterogeneous multicores exhibiting a rather high degree of adaptivity which is made possible by clearly structuring the system implementation along the proposed model of self-aware computing.

2. RELATED WORK IN SELF-AWARE COMPUTING

In this section, we first review important characterizations of previous self-* approaches and then turn to related work in self-aware compute nodes. Finally, we present the reference architectural framework for self-awareness used in our work.

2.1. Origins of Self-aware Computing

Self-organizing systems remained a rather broad and not that precisely defined category in literature. According to Di Marzo Serugendo et al. [Serugendo et al. 2011], a self-organizing system can change its internal structure and functionality at run-time without any explicit direction mechanism.

Multi-agent systems are one way for designing artificial self-organizing systems. Multi-agent systems employ several autonomous software agents that make local decisions and interact with each other to achieve their goals [Wooldridge 2009]. Multi-agent systems distinguish between two types of environments: physical and social environments. The environment can be characterized as accessible vs. inaccessible, deterministic vs. non-deterministic, discrete vs. continuous and static vs. dynamic.

Autonomic computing aims at solving the emerging complexity crisis in software engineering. Since humans are no longer able to deal with the rising complexity, dynamics, heterogeneity and uncertainty of future systems, these systems should be enabled to (autonomically) manage themselves. Self-management includes self-configuration, self-optimization, self-healing, and self-protection. Four self-* attributes have been proposed for autonomic computing systems: self-awareness, environment-awareness, self-monitoring, and self-adjustment [Sterritt and Hinchey 2010]. In an autonomous computing system, all system components are autonomic themselves and an autonomic manager monitors the components and the environment and develops and executes plans based on the analysis of this information. This approach is closely related to a hierarchical multi-agent system. IBM, who as a main driving force behind the autonomic computing idea, proposed a reference architecture named *MAPE-k* for the autonomic manager, which executes the monitor, analyze, plan and execute (MAPE) control loop and maintains a knowledge base [IBM 2003]. The architecture uses sensors to collect information about the environment and the system itself.

Organic computing [Schmeck et al. 2011] is a related concept that on one hand extends autonomic computing by the properties of self-organization and self-explanation, but on the other hand does not require systems to be fully autonomous. The approach introduces an observer and a controller component on top of the adaptive system. External users provide goals to the controller and only in case an adaptation violates these goals or any other given constraints, the controller interferes. This has been denoted as controlled autonomy.

Self-awareness appeared as key attribute in both autonomic and organic computing and, subsequently, research on self-aware computing systems has been fueled by DARPA [Paulson 2003] and EC [European Commission 2013] funding lines.

2.2. Self-aware Computing Nodes

Agarwal et al. [Agarwal et al. 2009] defined a self-aware computer to be introspective, adaptive, self-healing, goal-oriented, and approximate. The authors implement self-awareness through an observe-decide-act control loop similar to the MAPE-k model. Such a control loop found use in various prototypes [Santambrogio et al. 2010], including a heterogeneous system comprising a workstation and an FPGA accelerator [Sironi et al. 2011], a multi-processor inside a workstation [Sironi et al. 2012; Bartolini et al. 2012] and novel computer architectures for exascale computing [Hoffmann et al. 2012]. The observation phase employs monitors to measure metrics such as performance and power consumption. Performance monitoring mostly relies on the Heartbeats framework [Hoffmann et al. 2010], where a running application announces the completion of an application-specific amount of processing by issuing a heart beat. Simple performance goals can then be specified in terms of required heart beat rates. The decision phase allows for using performance models and learning components to deter-

mine a suitable adaptation whenever the goals are not met. The action phase actually performs the adaptation by turning "switches" or "knobs". For example, the switches might represent different implementation variants of an application's tasks and the knobs might represent parameters such as the clock rate or supply voltage of the cores.

Closely related to our work, Sironi et al. [Sironi et al. 2010] presented an FPGA-based self-aware adaptive computing system based on heterogeneous multi-cores. Their system supports performance monitoring through the Heartbeats framework, decision making and self-adaption. Performance adaptation is enabled by mapping an application to a CPU, a reconfigurable hardware core, or both. The authors used an encryption algorithm as application and implemented their system on a Xilinx Virtex-II Pro FPGA. The experiments covered only static measurements of the application's performance without swapping between implementations at run-time and provided self-adaptation only at the conceptual level. More recently, Sironi et al. [Sironi et al. 2011] discussed a heterogeneous system that consists of a multi-core processor and a reconfigurable device. Experiments were done on a platform with an Intel Core i7 and a Xilinx Virtex-5 FPGA. An application that hashes data blocks was instantiated four times with certain performance goals. Using a hot-swap mechanism that switches between a software and a hardware implementation, all performance goals were met. In contrast to [Sironi et al. 2010], we provide actual measurements of a system that dynamically adapts the number of used hardware cores in order to meet performance constraints. Unlike [Sironi et al. 2011], we target an embedded architecture where the entire system is implemented on a single chip and, in addition to respecting performance constraints, our system can also respect thermal constraints.

Further related work focuses on general methodologies to allow for autonomous self-adaptation on embedded multicore systems. For instance, [Zeppenfeld et al. 2011] have developed an autonomic homogeneous multi-processor system-on-chip architecture, where each processor is connected to several monitors, actuators and a single learning classifier table evaluator. The table stores condition-action rules where the fitness is learned at run-time using reinforcement learning. Multiple optimization goals were combined in a single objective function. Whenever the system performs a self-adaptation the used strategy receives a reward or a penalty, which depends on how much the system state has changed according to the objective function. The authors applied their general approach on a networking scenario with two levels of adaptation. On the one hand, the frequencies of each processor could be altered and, on the other hand, tasks could be migrated between the processors. The paper shows that it is beneficial when the autonomic network processors share workload information, defined as frequency times core utilization. [Diguet et al. 2011] proposed a generic self-adaptation methodology for heterogeneous multicore architectures, which distinguishes between *algorithmic* and *architectural* adaptations. A global configuration manager controls the architectural adaptations, which optimize the HW/SW partitioning of the task set in order to deal with trade-offs at the system-level, e.g. minimizing the overall power consumption versus maximizing the performance of the applications. Furthermore, each application is continuously optimized by a local manager, which can choose between different application specific algorithmic adaptations. The parameters for the algorithmic adaptations have to be defined by the application developer at design-time, for instance by using simulations. In contrast to [Zeppenfeld et al. 2011] our self-expression strategies do not yet include on-line learning. Moreover, we focus on heterogeneous multicore systems with a CPU and multiple reconfigurable hardware slots. In contrast to [Diguet et al. 2011] our approach does not differentiate between application-specific and system-level adaptations, although both forms of adaptation could be easily modeled and integrated into our system.

Several related works focused on balancing performance with power/energy consumption and thermal constraints in compute nodes. Although these works have not been using or stressing the term self-aware computing, they share the same scenarios, objectives and often also the methods. For example, Niu et al. [Niu et al. 2011] combined processors with reconfigurable hardware cores by equipping nodes of a compute cluster with FPGA accelerator cards. The computational intensive part of an N-Body simulation was mapped to the FPGA accelerator cards using multiple instances of a hardware core on a single FPGA accelerator. A central controller receives status information of all compute nodes over a wireless network, including temperature and power consumption readings for the FPGAs. Based on this information, the controller can enable/disable hardware cores on the FPGA accelerator cards and re-distribute workload inside the cluster to maximize the performance at given thermal and power budgets.

Jones et al. [Jones et al. 2007] proposed an adaptive FPGA-based architecture that switches between a low and a high clock frequency in order to decrease the latency of an application while maintaining a given thermal budget compared to a thermally-save static solution. The system is using a high clock frequency when the measured temperature is inside the thermal budget and the application generates load; otherwise, the clock frequency is lowered. The authors demonstrated their approach on a Field Programmable Extender (FPX) platform, where they compared their adaptive strategy switching between 25 MHz and 100 MHz with a thermally-save static solution at 50 MHz. For longer workload bursts they reduced power consumption by 30% and doubled the performance while maintaining a given temperature threshold of 70°C.

Chen and John [Chen and John 2009] designed a scheduling technique for heterogeneous multicore processors where the cores differ in instruction-level parallelism, branch predictor size, and data cache size. The scheduling technique profiles an application to find the best mapping with respect to high throughput and reduced energy consumption. Compared to a naïve scheduling technique, they could reduce the energy delay product by 24.5% on a 64 core system.

2.3. Reference Architectural Framework for Self-aware Compute Nodes

We base our work on the reference architectural framework proposed by Becker et al. [Becker et al. 2012], which is depicted in Figure 1. While in its very basic functionality this framework bears similarity with IBM's MAPE-k model, Agrawal et al.'s observe-decide-act control loop or the observer-controller structure of organic computing, it is more elaborate and draws inspiration from the notions of self-awareness and self-expression in neurocognitive sciences [Duval and Wicklund 1972; Goukens et al. 2007]. For example, neurocognitive sciences distinguish between private and public self-awareness and introduce several levels of self-awareness.

According to [Lewis et al. 2011], self-awareness requires the compute node to possess knowledge of and based on phenomena internal and external to itself. The internal phenomena are captured by sensors, e.g., utilization counters or thermal sensors, and handled by a private self-awareness engine. The external phenomena are recognized through the environment, e.g., workload and varying quality of service requirements, and handled by a public self-awareness engine. Both engines interact with models of the system and the environment which are optionally learned through this interaction, and altogether they provide the system with state and context (dark-gray boxes in Figure 1).

Self-expression (medium-gray boxes in Figure 1) is the ability of a node to adapt to changes in the system or the environment. To that end, the framework foresees a self-expression engine containing either a single adaption strategy or multiple ones. The

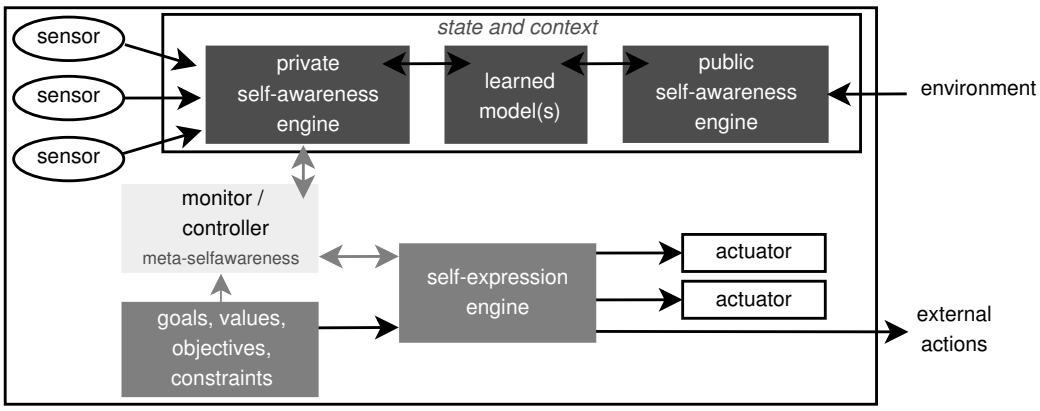

Fig. 1. Reference architectural framework for a self-aware compute node [Becker et al. 2012].

self-expression engine takes the system state and context as input in order to decide on an adaptation action. The adaptation itself is done by internal actuators, e.g., power management or thermal management by migrating threads in a heterogeneous multicore, or by taking external actions, e.g., communicating with other compute nodes or a user. Self-expression is driven by a system's goals, values, objectives and constraints which might be given at design-time or dynamically updated at run-time.

Referring to neurocognitive models, a compute node can optionally implement meta-self-awareness which can be seen as a higher level of self-awareness where a node is aware of its own awareness. In terms of the reference architectural framework meta-self-awareness maps to a monitor/controller component (light-gray box in Figure 1). Meta-self-awareness is required, for example, to select between different sensors, actors, learning techniques or between multiple adaptation strategies, the latter of which proved especially useful in presence of conflicting or rapidly changing objectives [Esterle et al. 2011]. While the framework implements the basic feedback loop of the MAPE-k model or Agarwal et al.'s observe-decide-act control loop with its self-awareness and self-expression components (from sensors and external environment to actuators and external actions), the monitor/controller component relates to the observer/controller of organic computing.

3. RECONOS

ReconOS is an architecture, programming model, and execution environment for FPGA-based heterogeneous multicores [Lübbers and Platzner 2009; Agne et al. 2011]. ReconOS integrates reconfigurable logic cores with CPU cores and runs hardware/software multithreaded applications on top of a common host operating system (see Figure 3) . Originally developed as extension layer on top of the eCos operating system, ReconOS was later ported to MMU-less Linux running on PowerPC and Microblaze cores. Over the years, ReconOS has been extended to support current FPGA architectures and design tool flows. The current version of ReconOS (v3) supports Xilinx Virtex-6 FPGAs and as host operating systems both Linux with full virtual memory support and the Xilkernel, a small lightweight operating system supplied by Xilinx. ReconOS is available as open source [ReconOS 2013].

In this section, we provide an overview over the ReconOS v3 architecture and programming and then discuss the suitability of ReconOS as architectural basis for implementing self-aware compute nodes according to the reference architectural framework of Section 2.

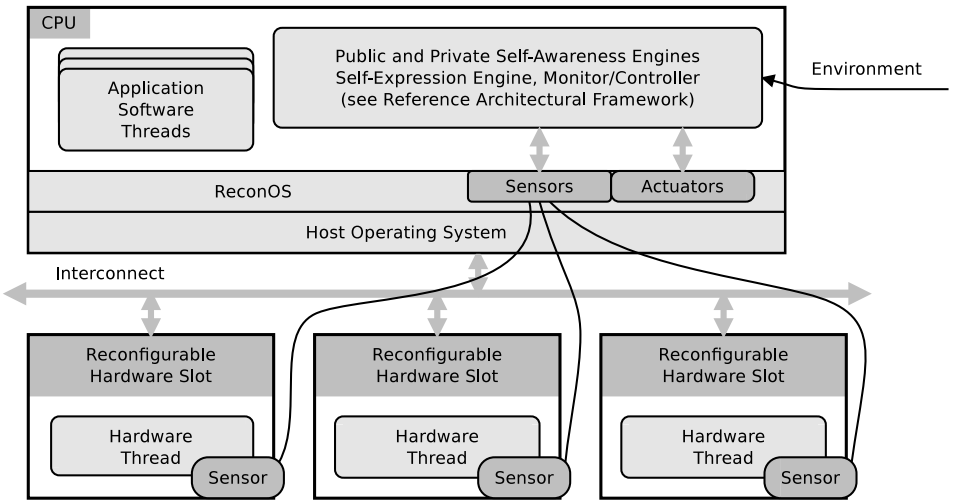

Fig. 2. A heterogeneous multicore architecture.

3.1. Architecture and Programming

ReconOS extends the well understood and widespread multithreaded programming and execution model from software threads to hardware threads. To this end, classic hardware accelerators or coprocessor cores are turned into hardware threads which can execute concurrently to other software and hardware threads. A ReconOS system contains reconfigurable regions on the FPGA, called reconfigurable hardware slots. Hardware threads can either be mapped to reconfigurable hardware slots statically at design time, or dynamically at runtime through partial reconfiguration. In ReconOS, all threads share the same address space and may synchronize and communicate with other threads through the use of operating system services, such as mutexes, semaphores, message queues, etc. Whether a specific thread resides in hardware or software is fully transparent to other threads.

While software threads have sequential execution semantics, typical hardware accelerators extensively use fine and medium-grained parallelism. As shown in Figure 3, we split hardware threads into two three main blocks. The user logic comprises the accelerator datapaths of the thread and implements the main computations. The operating system synchronization state machine (OSFSM) controls the datapaths via a set of handshake signals and interacts with the host operating system in a sequential manner. Finally, hardware threads may contain local memory to buffer data blocks for processing. Access to the memory hierarchy is also controlled by the OSFSM.

ReconOS hardware threads are coded in VHDL. We provide a VHDL function library with OS calls that is used to program the OSFSM. Table I gives an overview of the functions currently available with ReconOS v3. Each hardware thread is assigned a corresponding light-weight software thread, the so-called delegate thread. Whenever a hardware thread's OSFSM issues an OS call, the delegate thread becomes active and performs the OS call to the host operating system on behalf of the hardware thread. Through this concept neither the host OS kernel nor the user threads have to know whether other threads run in software or hardware. This transparency with respect to thread implementation greatly eases design space exploration and is key to adapting the system by changing thread mappings across the hardware/software boundary at runtime.

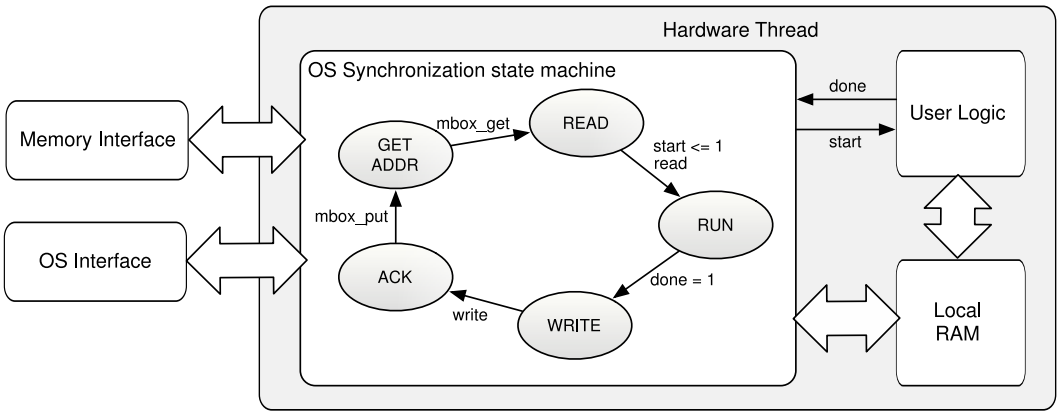

Fig. 3. An exemplary hardware thread with OSFSM, user logic, an internal RAM as well as the hardware thread's interfaces to memory and the OS.

Table I. Supported VHDL procedures for ReconOS hardware threads

Function	Description
`osif_sem_post()`	POSIX semaphores (counting and binary)
`osif_sem_wait()`	
`osif_mutex_lock()`	POSIX mutexes
`osif_mutex_trylock()`	
`osif_mutex_unlock()`	
`osif_cond_wait()`	POSIX condition variables
`osif_cond_signal()`	
`osif_cond_broadcast()`	
`osif_mbox_get()`	message boxes that allow for monitoring of fill count and fill rate
`osif_mbox_put()`	
`osif_rq_send()`	message queues
`osif_rq_receive()`	
`memif_read()`	(virtual) memory access
`memif_write()`	

3.2. Sensors and Actuators

Under ReconOS we use a thread-level implementation of the reference architectural framework (cmp. Figure 1), where the self-awareness and self-expression engines are wrapped into software threads. ReconOS itself enables the creation of self-aware compute nodes mainly through sensors and actuators.

In the current version we provide sensors for capturing i) performance and utilization as well as ii) temperature data. Sensing utilization is straight-forward and supported by corresponding OS services, monitoring software threads and extensions to hardware threads, e.g., activity counters that measure times where a hardware thread is idle waiting for data or synchronization. Performance is typically defined and captured in an application-specific way. We augment user level threads with calls to message boxes to send events or performance data to the thread implementing the private self-awareness engine. Being somewhat similar to the HeartBeats approach, our approach is more flexible. Many services, for example FIFOs (message queues) in ReconOS, are provided by operating system objects implemented in software. Data such as current fill levels can be retrieved via the ReconOS API at any time by the self-awareness engines. Periodic measurements of fill levels can be used to derive related utilization and performance data, such as fill rates.

Sensing temperature data is far more involved. In case only one temperature reading per time is required without any need for capturing the spatial temperature distribution, the FPGA's built-in thermal diode can be used. The diode is pre-calibrated

and can be read out using the system monitor core provided by Xilinx. In addition, ReconOS allows the designer to instantiate a thermal sensor array for finer-grained temperature measurements, in particular to measure the core temperatures in a multicore design. The sensor array spans over the total reconfigurable area of the chip and - after a calibration phase - delivers highly accurate, high frequency temperature readings [Happe et al. 2011]. ReconOS includes device drivers for both ways of sensing the temperature.

In general, the OS layer of a heterogeneous multicore can react to changes in the system state and environment by power and thermal management including, for example, starting, stopping and migrating threads. ReconOS currently supports actuators that drive thread scheduling operations such as creating and killing threads. Besides the POSIX-specified functions for software threads, there are currently two hardware thread related API functions for that purpose: The function `reconos_hwt_create()` is called to create a new hardware thread in a specific reconfigurable slot while `osif_thread_exit()` exits the calling hardware thread, freeing the operating system resources as well as the reconfigurable area associated with that thread.

4. CASE STUDY

The concept of self-awareness in compute nodes as described in [Lewis et al. 2011] provides a way to structure applications on heterogeneous multicores that have to deal with unpredictable system dynamics at runtime. The resulting implementation is then a (partial) instantiation of the architectural reference framework (cmp. Section 2). In this section we report on a case study involving two applications running on a single compute node at the same time, sorting and matrix multiplication. We first present the applications and the workload generated by them and provide implementation details. Then, in Section 4.1 we discuss strategies for dealing with performance constraints and in Section 4.2 we examine temperature constraints together with performance constraints. Finally, Section 4.3 compares the presented strategies.

For the sorting application we generate 8 kilobyte blocks of 32 bit integers at a varying rate and insert them into the application's input FIFO. We vary the rate to mimic a fractal workload W_s exhibiting a degree of self-similarity that is commonly observed in a number of application domains, such as networking [Leland et al. 1994]. The matrix multiplication operates on matrices of size $2^7 \times 2^7$. Using Strassen's algorithm [Strassen 1969] for matrix multiplication, larger matrices of size $2^n \times 2^n$ with $n >= 7$ can be handled by performing 7^{n-7} multiplications of matrices of size $2^7 \times 2^7$. This reduces the total number of scalar multiplications at the cost of additional memory. We assume the workload W_m for the matrix multiplication to be infinite, i.e., there will always be matrices for the system to multiply.

We have implemented both applications on a ReconOS system running on a Virtex-6 FPGA (XC6VLX240T). The FPGA area is divided into 13 regions. One large region contains the static system, including the Microblaze CPU, the memory controller and a UART device used to transfer data to a workstation. Additionally, 12 partially reconfigurable regions (slots) are allocated for the hardware threads. Any slot may contain either one sorting thread or one matrix multiplication thread at a time. We use the internal configuration access port (ICAP) of the Xilinx FPGA to partially reconfigure these regions, which enables us to time-share FPGA resources between the two applications. The system is highly heterogeneous because it contains two fundamentally different kinds of compuntational cores: The instruction set based main CPU runs the operating system and scheduling components, as well as the matrix subdivision step of the Strassen algorithm while dedicated hardware cores perform integer sorting and fixed sized matrix multiplication.

The resource requirement (post-synthesis data) of a sorting hardware thread amounts to 1452 6-LUTS, 424 flip-flops, and two block RAMs for local memory. A matrix multiplication hardware thread uses 1406 6-LUTS, 754 flip-flops, three multiply-accumulate blocks (DSP blocks), and 17 block RAMs for local storage. The complete system, including CPU, hardware threads, and memory bus, runs at 100 MHz clock frequency.

The typical execution time for sorting a block of integers is around 82 ms, and around 108 ms for a matrix multiplication. However, due to limited shared memory bandwidth these numbers may vary during runtime depending on the system's load. The best achievable reconfiguration delay D_{min} for a hardware thread is also determined by the available bandwidth to system memory. While in an otherwise idle system we have measured a D_{min} of 42 ms, the reconfiguration delay increased by a factor of more than 9$\times$ for heavily loaded systems.

4.1. Self-expression under Performance Constraints

In this scenario, we combine a performance constraint for the sorting application with the objective to maximize the number of matrix multiplications. With L_{max} as the capacity of the sorting application's input FIFO, we want the FIFO's fill level $L_s(k)$ at any time step k not to exceed the maximum level $\forall k : \; L_s(k) \leq L_{max}$, i.e., the FIFO should not overflow. Blocks that would lead to a FIFO overflow are discarded and counted as constraint violations.

The challenge for finding a good self-expression strategy, that is an assignment of hardware threads to slots and a corresponding schedule, meeting the constraint and optimizing the objective arises from i) the workload imposed on the system, which is not known in advance and ii) the internal dynamics of the system caused by mutual interference between memory accesses for reconfiguration and processing. Since the reconfiguration interface and the hardware threads share a single bus to main memory, bandwidth used for transferring reconfiguration data is not available for transferring processing data and vice versa. Moreover, when several hardware slots undergo reconfiguration some of the hardware threads will be delayed because there is only one ICAP reconfiguration interface. The resulting system dynamics are hard to model analytically, which motivates the self-aware computing approach.

We generate the sorting workload W_s deliberately to temporarily exceed the system's maximum sorting rate R_{max}, i.e., the maximum continuous rate at which blocks can be inserted without overflowing the input FIFO. Our implementation would achieve R_{max} if all 12 hardware slots were configured with sorting threads. Workloads temporarily exceeding R_{max} stress the system's self-expression strategy, because spikes in the workload must be compensated for in order to meet the performance constraint.

According to the architectural reference framework, we implement a thread containing the private self-awareness engine. This engine collects and maintains information about the system state, such as the FIFO fill level $L_s(k)$, the current FIFO in-rate, and the last measured reconfiguration delay D_i for each reconfigurable hardware slot i. The collected information is then used by the self-expression engine implementing the self-expression strategy. We invoke the self-expression engine at discrete time steps with the interval Δt. The engine decides on the number of threads required for sorting and for matrix multiplication and runs the scheduler that stops threads and triggers reconfiguration if necessary. In terms of the reference architectural framework, the scheduler acts as an actuator. The choice of Δt has as significant impact on the behavior of the self-awareness engine. A lower Δt enables the system to respond faster to workload changes at an increased computational overhead. In our experiments we use a Δt of one second which reduces the load caused by the self-awareness engine to a

negligible level while it still allows the system to react to workload changes sufficiently quick.

In the following, we discuss two fundamental, complementary self-expression strategies and one meta-strategy that decides on which fundamental strategy will be used. The meta-strategy tries to incorporate the strengths of both sub-strategies while avoiding their weaknesses.

Proportional strategy:. This strategy sets the number of required sorting threads $N_s(k)$ in time step k proportional to the current fill level of the FIFO $L_s(k)$:

$$N_s(k) := k_p \cdot L_s(k)$$

with the proportional factor

$$k_p := N_{\max}/L_{\max}$$

where $N_{\max}$ is the number of available hardware slots and $L_{\max}$ is the capacity of the input FIFO. The proportional strategy handles most workloads well that do not surpass the maximum continuous sorting rate R_{max}. It will however quickly violate the performance constraint in the case the workload peaks greater than R_{max}. Figure 4 summarizes the results for this strategy. Figure 4(a) displays the modulated fractal sorting workload W_s in blocks per second (BPS). Figure 4(b) depicts the number of currently configured sorting threads, ranging from 0 to 12, and Figures 4(c)-(e) present the fill level of the input FIFO, $L_s(k)$, the rate $R_s(k)$ at which blocks are sorted in blocks per second, as well as the rate $R_m(k)$ at which matrix multiplications are performed in multiplications per second (MPS). All measurements were taken on a single run of the application over a period of 200 seconds. After 200 seconds, 126 blocks were dropped by the proportional strategy in violation of the performance constraint.

All-or-nothing strategy:. We designed the all-or-noting strategy to better handle workload peaks in excess of R_{max}. Once engaged, the strategy tries to empty the input FIFO completely and as quickly as possible, using all available hardware slots for sorting. When $L_s(k)$ drops to zero, all hardware slots are reconfigured to multiply matrices. By monitoring the current fill rate of the input FIFO as well as recent reconfiguration delays, the strategy calculates in each time step the fill level at which it must start reconfiguring all hardware slots with sorting threads to avoid an overflow of the input FIFO. In detail, the strategy decides on the number of required sorting threads $N_s(k)$, given the current input FIFO fill rate $L'_s(k) = (L_s(k) - L_s(k-1))/\Delta t$, and the recently measured reconfiguration delays D_i of the slots i, such that

$$N_s(k) := \begin{cases} 0 & \text{if } L_s(k) = 0 \\ N_{\max} & \text{if } L_s(k) > L_{\text{trigger}} \quad \text{or} \quad L'_s(k) > \alpha \\ N_s(k-1) & \text{else} \end{cases}$$

with α being the critical fill rate given by

$$\alpha := \frac{L_{\max}}{\sum_{i=1}^{N_{\max}} D_{i-1}}$$

The purpose of the condition $L_s(k) > L_{\text{trigger}}$ is to avoid the FIFO slowly becoming full without $L'_s(k)$ ever surpassing α. In our experiments we found it sufficient to set L_{trigger} to $L_{\max}/4$. Figure 5 presents the experimental results for the all-or-nothing strategy. While this strategy handles the sorting workload without constraint viola-

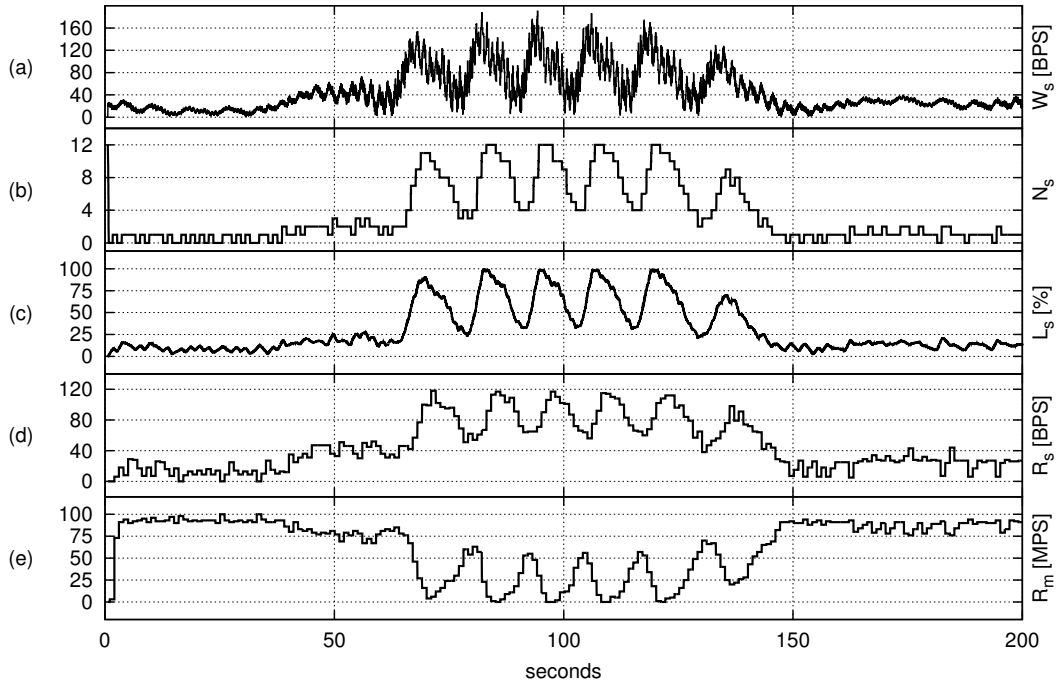

Fig. 4. Experimental evaluation of the *proportional* strategy. The figure is structured as follows: (a) workload, in blocks/s; (b) number of sorting threads; (c) input FIFO fill level, in percent; (d) the rate at which blocks are sorted, in blocks/s; (e) the rate at which matrices are multiplied, in multiplications/s.

tions, it also performs less multiplications than the proportional strategy as the comparison in Table II shows.

Meta strategy:. We have developed a meta-strategy that tries to leverage the advantages of both fundamental strategies while avoiding their weaknesses. Both, the proportional and the all-or-nothing strategy prefer certain workloads over others. The first strategy leads to high multiplication performance over a wide range of workloads, but handles sorting workload spikes rather poorly. The second strategy copes with workload spikes without constraint violations at the price of an overall decreased number of matrix multiplications. The definition of the meta strategy follows straight from an examination of the experimental results. The key insight is that the proportional strategy will meet the performance constraint as long as the input FIFOs fill rate is at most R_{max}. For higher fill rates, the more FIFO space conservative strategy all-or-nothing will minimize constraint violations.

Self-expression strategy *meta*:

(rule #1)proportional: $(\forall k - p \leq j \leq k : \quad L'_s(j) \leq R_s(j))$
(rule #2)all_or_nothing: (else)

The purpose of the parameter p in the meta strategy is to reduce oscillations between the two single strategies in the face of a noisy workload. We found a value of 20 to work well in our experiments.

Figure 6 shows the experimental evaluation of the meta strategy. The switch from the proportional to the all-or-nothing strategy happens at the beginning of the first workload spike surpassing R_{max} which is also indicated by the abrupt changes in the

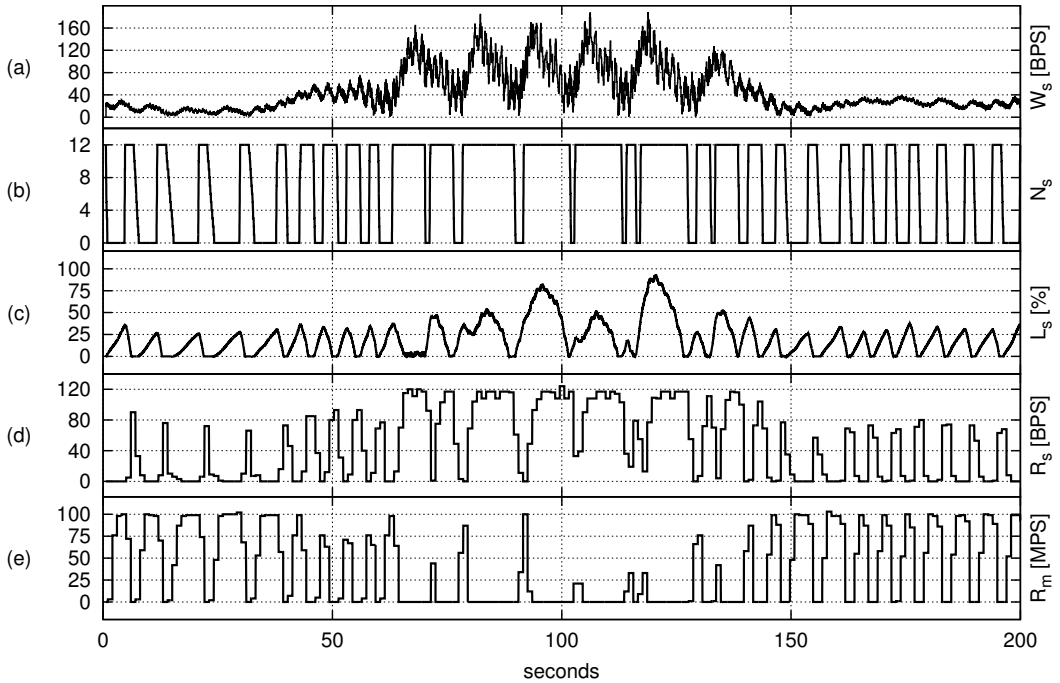

Fig. 5. Experimental evaluation of the *all-or-nothing* strategy. The figure is structured as follows: (a) workload, in blocks/s; (b) number of sorting threads; (c) input FIFO fill level, in percent; (d) the rate at which blocks are sorted in blocks/s; (e) the rate at which matrices are multiplied, in multiplications/s.

number of sorting threads. After the workload smoothes, the meta strategy switches back to the proportional mode.

4.2. Self-expression under Conflicting Constraints

Often, computing system have to deal with conflicting objectives or constraints. In case of our heterogeneous multicore, a typical conflict exists between performance and temperature management. To exemplify a situation with conflicting constraints, we introduce thermal constraints to our case study. We extend the private self-awareness engine to continuously monitor the chip temperature using the Xilinx system monitor. When the temperature exceeds a threshold θ_{low} the self-expression strategy stops hardware threads of applications that do not have any performance constraints, i.e., the matrix multiplication in our case study. When the temperature rises above the threshold θ_{high}, all remaining hardware threads are stopped for a quick temperature reduction. In case that no thermal constraints are violated, the system applies the meta strategy for self-adaptation.

In order to decrease the chip temperature the application might have to disable hardware threads which, potentially, leads to a violation of the performance constraint. As a consequence, our strategy favors thermal constraints over performance constraints.

While thermal management of FPGA-based systems will become increasingly important in the foreseeable future due to shrinking device structures and increasing densities [Borkar 2005], our sorting and matrix multiplication applications on today's FPGA technology, however, do not generate significant heat. In order to emulate the thermal situation of a future FPGA multicore we have integrated dedicated logic into

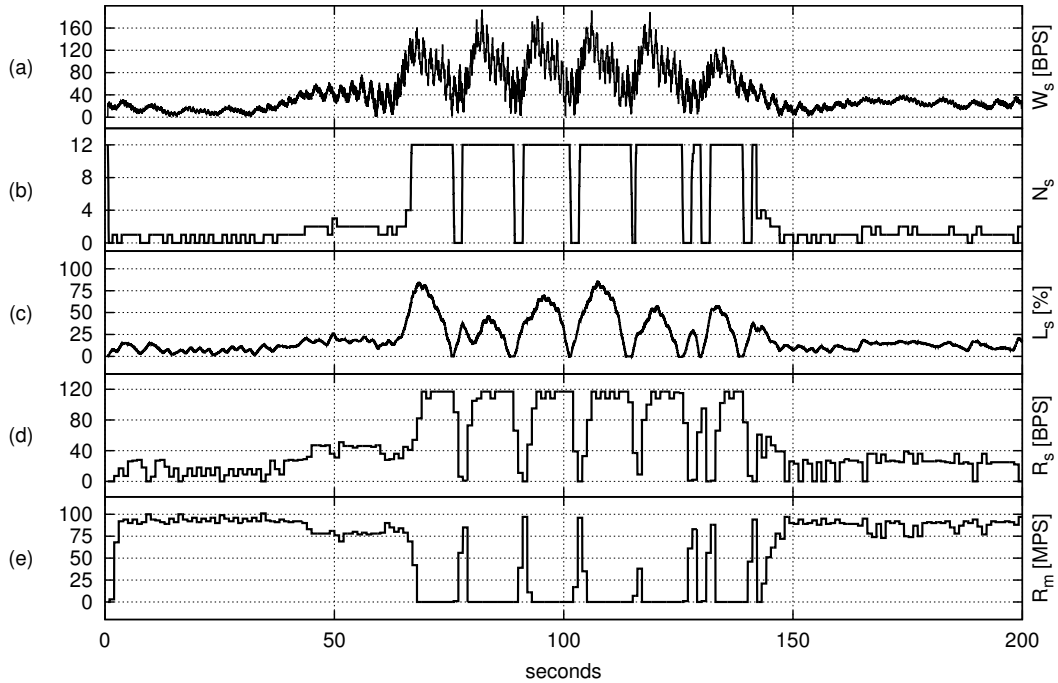

Fig. 6. Experimental evaluation of the *meta* strategy. The figure is structured as follows: (a) workload, in blocks/s; (b) number of sorting threads; (c) input FIFO fill level, in percent; (d) the rate at which blocks are sorted, in blocks/s; (e) the rate at which matrices are multiplied, in multiplications/s.

the hardware threads that creates heat whenever the thread is active. This dedicated heat-generating logic comprises a number of ring oscillators, each implemented in a single LUT [Happe et al. 2012]. For the sorting hardware thread, we have integrated 150 ring oscillators and for the matrix multiplication hardware threads 50 ring oscillators.

Figure 7 displays the experimental results for the thermally-aware strategy. The thermal thresholds were set to $[\theta_{\text{low}}, \theta_{\text{high}}] = [46.5°C, 47°C]$, indicated by the gray band in Figure 7(f). Since the thermal thresholds are quite tight, the system often violates the performance constraint of the sorting application and drops 309 data blocks in 200 seconds. Furthermore, compared to the results achieved by the meta strategy without thermal constraints the matrix multiplication performance decreases from 75 to about 25 multiplied matrices per second once the chip temperature exceeds θ_{low}, Figure 6(e) and Figure 7(e).

4.3. Comparison of Self-expression Strategies

Table II compares for all self-expression strategies the performance constraint violations in number of dropped sort blocks, the matrix multiplication performance, and the maximum measured temperature. Among the thermally-unaware strategies, the meta strategy clearly excels by respecting the performance constraint while processing almost as many matrix multiplications as the proportional strategy. The thermally-aware meta strategy can respect additional thermal constraints at the price of more performance constraint violations and a lower number of matrix multiplications.

Figure 8 depicts the temperature development over time for all presented self-expression strategies. Only the thermally-aware strategy is able to quickly react to

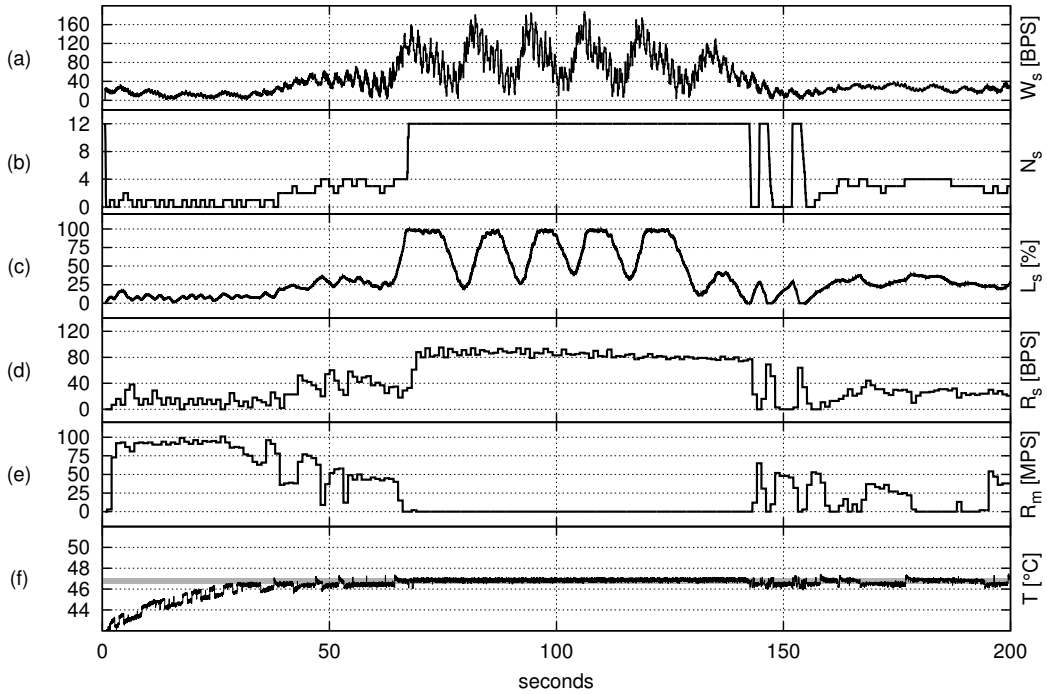

Fig. 7. Measurement results for the thermal-aware meta strategy: The figure is structured as follows: (a) workload, in blocks/s; (b) number of sorting threads; (c) input FIFO fill level, in percent; (d) the rate at which blocks are sorted, in blocks/s; (e) the rate at which matrices are multiplied, in multiplications/s; and (f) the chip temperature over time; in °C. The thermal constraints are highlighted by a gray horizontal bar.

Table II. Comparison between different self-expression strategies over 200 seconds

self-expression strategy	dropped sort blocks	matrix multiplications	maximum temperature
proportional	126	12747	50.2°C
all-or-nothing	0	7467	48.9°C
meta	0	11606	49.7°C
thermally-aware meta	309	5646	47.5°C

temperature peaks that exceed the specified bounds and, therefore, successfully manages the chip temperature. All the other strategies violate the thermal constraints for longer time periods; the proportional strategy even by up to 3.2°C, see Figure 8(a).

5. DISCUSSION AND CONCLUSION

In this paper we have discussed how the need to handle increasingly complex, diverse and contradictory requirements along with the decreased reliability of components have set the ground for research in computing systems exhibiting self-* properties. Over the last decade, several models for self-* systems have been proposed and have also been applied in heterogeneous multicore systems. Our work builds on the reference architectural framework for self-aware computing systems proposed by Becker et al. [Becker et al. 2012] which we have instantiated for an FPGA-based heterogeneous multicore system. Our multicore is based on the ReconOS architecture and programming model which provides a unified abstraction for hard- and software threads and enables a dynamic use and migration of functionality between hard- and software, adding a further degree of freedom for self-adaptation over pure CPU-based systems.

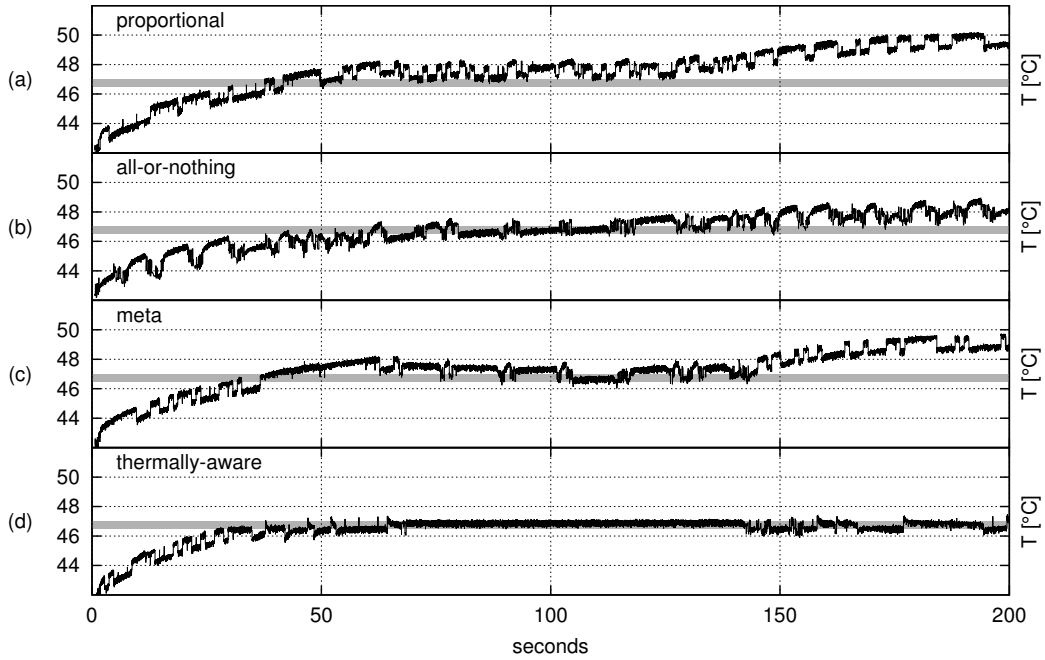

Fig. 8. Temperature development over time for all experiments: The thermal constraints are highlighted by a gray horizontal bar.

As a case study, we have instantiated the self-awareness, self-expression, and monitor/controller components of the reference architectural framework for a heterogeneous multi-core system that executes two applications, block sorting and matrix multiplication, that compete for computing resources. We have examined the system under constraints in performance and temperature using a varying, noisy workload. We have shown how rule-based adaptations strategies can be used to operate the system with one or several, possibly conflicting, constraints. Finally, we have shown that none of the presented strategies is dominant in the sense that it delivers the best performance for all points of operation, but that the awareness of the system of the current operational state can be exploited in a meta-strategy that selects a good self-expression strategy for the current operation point.

The reference architectural model has proven very useful in separating concerns and structuring both the engineering process as well as the component architecture of the runtime system. While adaptive computing systems with limited complexity and basic adaptation strategies could arguably also be implemented using an ad-hoc approach, ad-hoc designs become infeasible for complex systems that comprise many sensors, actuators and more sophisticated algorithmic techniques in their self-awareness and self-expression components.

There are a number of interesting aspects of this research that we would like to expand on in future work. This includes a comprehensive comparison to classical control systems as well as a more detailed investigation of the trade-offs between decision-making granularity and the computational overhead of the self-awareness engine.

REFERENCES

AGARWAL, A., MILLER, J., EASTEP, J., WENTZIAFF, D., AND KASTURE, H. 2009. Self-aware Computing. *Final Technical Report AFRL-RI-RS-TR-2009-161*, 81.

AGNE, A., PLATZNER, M., AND LUBBERS, E. 2011. Memory Virtualization for Multithreaded Reconfigurable Hardware. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. IEEE.

BARTOLINI, D. B., SIRONI, F., MAGGIO, M., CATTANEO, R., SCIUTO, D., AND SANTAMBROGIO, M. D. 2012. A Framework for Thermal and Performance Management. In *Workshop on Managing Systems Automatically and Dynamically (MAD)*.

BECKER, T., AGNE, A., LEWIS, P. R., BAHSOON, R., FANIYI, F., ESTERLE, L., KELLER, A., CHANDRA, A., JENSENIUS, A. R., AND STILKERICH, S. C. 2012. EPiCS: Engineering Proprioception in Computing Systems. In *Proceddings Conf. on Embedded and Ubiquitous Computing (EUC)*. IEEE.

BECKER, T., JIN, Q., LUK, W., AND WESTON, S. 2011. Dynamic Constant Reconfiguration for Explicit Finite Difference Option Pricing. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE Computer Society, Los Alamitos, CA, USA, 176–181.

BORKAR, S. 2005. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE MICRO*, 10–16.

CHEN, J. AND JOHN, L. K. 2009. Efficient program scheduling for heterogeneous multi-core processors. In *Proc. Design Automation Conference (DAC)*. ACM.

COSKUN, A. K., ROSING, T. S., WHISNANT, K. A., AND GROSS, K. C. 2008. Static and Dynamic Temperature-Aware Scheduling for Multiprocessor SoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16,* 9, 1127–1140.

DIGUET, J.-P., EUSTACHE, Y., AND GOGNIAT, G. 2011. Closed-loop–based Self-adaptive Hardware/Software-Embedded Systems: Design Methodology and Smart Cam Case Study. *ACM Transactions on Embedded Computing Systems 10,* 3, 1–28.

DUVAL, S. AND WICKLUND, R. A. 1972. A Theory of Objective Self Awareness. *Academic Press*.

EBI, T., FARUQUE, M. A. A., AND HENKEL, J. 2009. TAPE: Thermal-Aware Agent-Based Power Economy for Multi/Many-Core Architectures. In *Proc. Int. Conf. on Computer-Aided Design (ICCAD)*.

ESTERLE, L., LEWIS, P., BOGDANSKI, M., RINNER, B., AND YAO, X. 2011. A Socio-economic Approach to Online Vision Graph Generation and Handover in Distributed Smart Camera Networks. In *Proc. Int. Conference on Distributed Smart Cameras (ICDSC)*.

EUROPEAN COMMISSION. 2013. Self-Awareness in Autonomic Systems.

GOUKENS, C., DEWITTE, S., AND WARLOP, L. 2007. Me, Myself, and My Choices: The Influence of Private Self-awareness on Preference-behavior Consistency. *Open Access Publ., Katholieke Universiteit Leuven*.

HAPPE, M., AGNE, A., AND PLESSL, C. 2011. Measuring and predicting temperature distributions on FPGAs at run-time. In *Proc. Int. Conf. on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE.

HAPPE, M., HANGMANN, H., AGNE, A., AND PLESSL, C. 2012. Eight Ways to put your FPGA on Fire – A Systematic Study of Heat Generators. In *Proc. Int. Conf. on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE.

HOFFMANN, H., EASTEP, J., SANTAMBROGIO, M. D., MILLER, J. E., AND AGARWAL, A. 2010. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *International Conference on Autonomic Computing (ICAC)*.

HOFFMANN, H., HOLT, J., KURIAN, G., LAU, E., MAGGIO, M., MILLER, J. E., NEUMAN, S. M., SINANGIL, M. E., SINANGIL, Y., AGARWAL, A., CHANDRAKASAN, A. P., AND DEVADAS, S. 2012. Self-aware Computing in the Angstrom Processor. In *Proc. Design Automation Conference (DAC)*. ASM.

IBM. 2003. An architectural blueprint for autonomic computing. Tech. rep., IBM.

JONES, P., CHO, Y., AND LOCKWOOD, J. 2007. Dynamically optimizing FPGA applications by monitoring temperature and workloads. In *Proc. Int. Conf. on VLSI Design (VLSID)*. IEEE.

LELAND, W., TAQQU, M., WILLINGER, W., AND WILSON, D. 1994. On the self-similar nature of ethernet traffic (extended version). *Networking, IEEE/ACM Transactions on 2,* 1, 1–15.

LEWIS, P. R., CHANDRA, A., PARSONS, S., ROBINSON, E., GLETTE, K., BAHSOON, R., TORRESEN, J., AND YAO, X. 2011. A Survey of Self-Awareness and Its Application in Computing Systems. In *Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*.

LÜBBERS, E. AND PLATZNER, M. 2009. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS) 9*.

MULAS, F., ATIENZA, D., ACQUAVIVA, A., CARTA, S., BENINI, L., AND DEMICHELI, G. 2009. Thermal balancing policy for multiprocessor stream computing platforms. *IEEE Transactions on CAD of Integrated Circuits and Systems 28,* 12, 1870–1882.

NIU, X. Y., TSOI, K. H., AND LUK, W. 2011. Reconfiguring distributed applications in FPGA accelerated cluster with wireless networking. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

PAULSON, L. 2003. DARPA Creating Self-aware Computing. *IEEE Computer 36,* 3, 24.

RECONOS. 2013. A programming model and OS for reconfigurable hardware.

SANTAMBROGIO, M. D., HOFFMANN, H., EASTEP, J., AND AGARWAL, A. 2010. Enabling Technologies for Self-aware Adaptive Systems. In *Proc. Conf. on Adaptive Hardware and Systems (AHS)*. IEEE.

SCHMECK, H., MÜLLER-SCHLOER, C., CAKAR, E., MNIF, M., AND RICHTER, U. 2011. *Organic Computing — A Paradigm Shift for Complex Systems Autonomic Systems*. Vol. 1. Springer Basel, Chapter Adaptivity and Self-organisation in Organic Computing Systems, 5–37.

SERUGENDO, G. D. M., GLEIZES, M.-P., AND KARAGEORGOS, A. 2011. *Self-Organizing Software – From Natural to Artificial Adaptation*. Springer.

SIRONI, F., BARTOLINI, D. B., CAMPANONI, S., CANCARE, F., HOFFMANN, H., SCIUTO, D., AND SANTAMBROGIO, M. D. 2012. Metronome: Operating System Level Performance Management via Self-adaptive Computing. In *Proc. Design Automation Conference (DAC)*. ACM.

SIRONI, F., CUOCCIO, A., HOFFMANN, H., MAGGIO, M., AND SANTAMBROGIO, M. 2011. Evolvable Systems on Reconfigurable Architecture via Self-aware Adaptive Applications. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*.

SIRONI, F., TRIVERIO, M., HOFFMANN, H., MAGGIO, M., AND SANTAMBROGIO, M. 2010. Self-aware Adaptation in FPGA-based Systems. In *Int. Conf. on Field Programmable Logic and Applications*. IEEE.

STERRITT, R. AND HINCHEY, M. 2010. SPAACE IV: Self-Properties for an Autonomous and Autonomic Computing Environment - Part IV A Newish Hope. In *IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe)*.

STRASSEN, V. 1969. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356.

WOOLDRIDGE, M. J. 2009. *An Introduction to MultiAgent Systems* Second Ed. Wiley.

ZEPPENFELD, J., BOUAJILA, A., STECHELE, W., BERNAUER, A., BRINGMANN, O., ROSENSTIEL, W., AND HERKERSDORF, A. 2011. Applying ASoC to Multi-core Applications for Workload Management. In *Organic Computing — A Paradigm Shift for Complex Systems*, C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds. Autonomic Systems Series, vol. 1. Springer Basel, 461–472.

Received ; revised ; accepted Feburary 2014