Research Article

Enabling Self-Organization in Embedded Systems with Reconfigurable Hardware

Christophe Bobda,¹ Kevin Cheng,¹ Felix Mühlbauer,¹ Klaus Drechsler,² Jan Schulte,² Dominik Murr,² and Camel Tanougast³

¹ Computer Engineering Group, University of Potsdam, 14482 Potsdam, Germany

² Kaiserslautern University of Technology, Kaiserslautern, Germany

³ P. Verlaine University of Metz, 57000 Metz, France

Correspondence should be addressed to Jan Schulte, jan@janschulte.com

Received 31 December 2008; Accepted 23 July 2009

Recommended by J. Manuel Moreno

We present a methodology based on self-organization to manage resources in networked embedded systems based on reconfigurable hardware. Two points are detailed in this paper, the monitoring system used to analyse the system and the Local Marketplaces Global Symbiosis (LMGS) concept defined for self-organization of dynamically reconfigurable nodes.

Copyright © 2009 Christophe Bobda et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

Increasing complexity due to rapid progress in information technology is making systems more and more difficult to integrate and control. Due to the large amount of possible configurations and alternative design decisions, the integration of components from different manufacturers in a working systems cannot be done only at design time anymore. Systems must be designed to cope with unexpected runtime environmental changes and interactions. They must be able to organize themselves to adapt to changes and avoid undesirable or destructive behaviors. In heterogeneous environments devices with different abilities are available. An example would be a sensor network consisting of powerful nodes called beacons doing all the computational work and collecting environmental information from less powerful sensor nodes, which of course could also do some in-network processing of the collected data. The ongoing progress in chip technology provides us with steadily increasing computation power whereas the accompanying miniaturization shrinks the device sizes to unprecedented measures at rock bottom prices. It is obvious that even the weakest sensor node will have enough computation power to share its resources with

other nodes in the near future by offering its resources as services to them.

In this work we try to answer the question how nodes in networks may utilize reconfigurable hardware and share their resources with other nodes to optimize load and dependability, thus adapting to a changing environment. To answer this question we investigate self-organization to implement a distributed resource management in these kind of networks. Self-organization is naturally found in biological systems and can be defined as a process in which pattern at the global level of a system emerges solely from numerous interactions among the lower-level components of the system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern [1]. If we translate this definition of self-organization into embedded systems we get something which has been characterized as the vision of autonomic computing [2]. There an autonomic system is a collection of so-called autonomic elements which is an individual system that contains resources and delivers services to humans and other autonomic elements. Figure 1 shows the building blocks of an autonomic element. It consists of an autonomic manager which controls one or

FIGURE 1: An autonomic element as presented in the vision of autonomic computing [2].

more managed elements. The autonomic manager gathers information about the managed element and its surrounding environment by using a monitor and constructs and executes plans based on an analysis of this information. Prior learned knowledge can be used in these steps to construct more intelligent plans, thus behaving more intelligent. Not all of these blocks are necessary to build an autonomic element. It depends heavily on the application to decide which block should be implemented and how.

In our system the *monitor* is implemented as a standalone application which gathers information from the local system and offers these information to neighbour nodes and in turn also gathers information from them. Because the monitor is extensible, the type of information a monitor can offer and collect is manifold and ranges from system information like CPU load, memory usage, used/unused reconfigurable area to positioning information, and offered services. A service a node may offer is, for example, its computational power. These information together with information from a knowledge database, that is built upon antecedent experiences, is used by a resource manager, that implements the *analyze* and *plan* blocks (see Figure 1), to distribute tasks and optimize resource utilization or power consumption in the network. In order to enforce flexibility in such systems and allow single nodes to adapt their behavior to disturbances, we use reconfigurable units, in this case FPGAs. Those devices pose a decent trade-off between computation power, energy consumption, and flexibility. They can be configured and reconfigured to provide hardware acceleration for highly specialized services. Furthermore, they can be partially reconfigured while keeping the rest of the system working. Some are even capable of initiating their reconfiguration themselves keeping the part of their logical circuits that hosts the local operating system up and running while only a small partition that hosts specialized accelerators is being reconfigured [3]. The *execute* block (see Figure 1) implements something like a task manager or reconfiguration manager. It is able to take a piece of hardware (we call it hardware module) either from a local storage or from any other node in the network and reconfigure itself to execute a computation-intensive task.

To investigate the principles of self-organization in networked embedded systems, we have developed a flexible and extensible monitor to track the state of the surrounding neighbours, which serves as the basis for a powerful and flexible approach to manage the available resources in networked embedded systems with reconfigurable hardware. A framework is shown that enables nodes to (re)distribute tasks in a marketplace-like manner, which delivers the basis technology to elevate implementations of collective task completion to a new level. Here a node that recognizes the need for a certain task to be done formulates it as a query to itself and its neighbors. Every node that offers the execution of a task replies to a query with its cost for fulfilling the job. The inquirer is now able to choose whether it is more appropriate to maybe reconfigure and do the task by itself or to delegate.

Both worlds, namely, self-organization in embedded systems and embedded systems built around reconfigurable hardware, lead to highly adaptive distributed systems.

The remaining part of this paper is structured as follows. The next section gives an introduction into the technological basis we have used to develop and test our system (Section 1). It follows a description of the implemented monitor (Section 2) and an explanation of a framework, which enables nodes to redistribute tasks in a marketplace-like manner (Section 3). Finally we conclude this article and discuss future work.

2. Technological Basis

2.1. FPGA and Partial Reconfiguration. The essential resources that must be available on each node in a distributed cooperative system are a minimal local computation power, a resource management, the ability to communicate and a system to distribute or gather jobs in the network.

Concerning the computation power, to propose a flexible and powerful node, FPGA technology is used. As described by Xilinx [4], their S-RAM-based FPGAs are configured by loading application-specific configuration data: the bitstream into an internal memory. The bitstream is a binary encoded file, that is, the equivalent of the design in a form that can be downloaded into the FPGA device. This basic idea of FPGAs is the most important. It means that it is possible to create an electronic design based on an FPGA remotely through the network.

To develop and test our concepts we use a Virtex-II Pro FPGA sitting on the XUP development board, a Xilinx ML403 board as well as the hydraXC-50 module which are both equipped with a Virtex-4 FX12 FPGA, and a Xilinx ML505 board equipped with a Virtex-5 LX50T.

Besides the configurable logic cells, present inside the FPGA, that allow custom hardware accelerators, the Virtex-II Pro, Virtex-4, and Virtex-5 supply a certain amount of basic, hard-wired circuits (primitives) to extend the device's speed and effectiveness. These are, for example, multipliers, block RAM, and especially a module named ICAP: Internal Configuration Access Port (see Figure 2). This module is the key for a node to change its own reconfigurable logic; it is an internal access to the configuration memory where the

FIGURE 2: Description of Dynamically Reconfigurable SoC with ICAP interface.

current bitstream is stored. In addition, Xilinx FPGAs are capable of partial reconfiguration since the Virtex-II series. That means that single hardware modules can be exchanged while vital parts of the node, like the memory controller or the network interface controller, keep on operating uninterrupted.

To realize a partial reconfigurable system [5], Partial Reconfigurable Regions (PRRs) have to be defined inside the FPGA. This PRRs are regions where Partial Reconfigurable Modules (PRMs) can take place. PRMs represent the electronic circuit of functional units which can be placed according to the application. As for a standard design that uses bitstreams, a PRM is represented by a partial-bitstream. This partial-bitstream can be stored in the system memory (Compact flash card or PROM) and transferred into the configuration memory when it is necessary. Then, it is easy to consider partial-reconfiguration management as a file management. Indeed, bitstreams can be manipulated like common file and send via a network.

All PR Regions and Modules must have exactly the same interface to be connected with the static part of the design. This particularity is a limitation in the design for partial reconfigurable hardware but it is the key to be able to change a part of the design at run-time. This interface between PR Regions and the Static Region (SR) is designed using Bus Macros (BMs) which are used to create a link between the reconfigurable and the fixed logic of the design. They are also fixed to be able to exchange PR Modules and to know where to connect I/Os precisely.

When designing a partial reconfigurable system, the border between static and dynamic parts must be defined. Inside the dynamic part, the area size and the position of each PR Region are also determinant [6]. The development

of such system has a direct influence on the quantity and the size of bitstreams.

PR Modules are synthesized specially for a certain PR region. In consequence, each PR Module is synthesized for each region. This allows the relocation of a module in all PR Regions, locally or in another node. In such systems, the number of partial bitstreams can rapidly increase.

The size of a bitstream has a direct impact on the systems performance. Moreover, FPGA resources like CLBs or BRAMs [7] are directly related with the configuration memory [4]. As explained before, the configuration of an FPGA is done by loading a bitstream into the internal configuration memory (see Figure 2). Thus, the time necessary to reconfigure the FPGA depends on the bitstream's size.

Virtex-4 and Virtex-5 boards have been used to test the partial-reconfiguration, in a network context, and to implement a self-organisation concept. The first implementation is the control of the partial reconfiguration of one board by another. On each board, a System on Chip (SoC) is implemented. It includes a MicroBlaze soft-core processor used to run a software that controls the reconfiguration via the ICAP port. Using a custom link between boards, a command is sent to request a dynamic reconfiguration. On each board, bitstreams and partial-bitstreams are stored in a compact flash card.

Beside that, a hydraXC board has been used in conjunction with a MicroBlaze soft-core CPU and an uClinux operating system to rapidly develop the monitoring system as described in the next section.

2.2. System Development and Linux. Linux was also used as general purpose operating system on the other development

FIGURE 3: Layout of our sample system: the small highlighted area is reconfigurable.

boards. Linux serves our needs in several ways. It encapsulates the difficulties to access hardware and thus facilitates resource deployment through its abstract driver interfaces. For example, the ICAP is incorporated into the Linux system with John Williams' device driver [3]. Especially the fully developed networking abilities are a convenient way to realize communication. The innumerable quantity of applications for any kind of need and furthermore adaption and development of software in high-level programming languages pose a big plus. The use of a standard Linux kernel and applications also speeds up development for another reason: the large community that exists and the vast database of solutions to standard problems.

Linux provides us with a fully featured TCP/IP-stack that we have used as a basis for developing and prototyping our applications. We use tools like *telnet* to control devices remotely and *wget* to transfer data. It is clear that certain application fields have to adapt the used communication protocols to fulfill their needs, for example, energy awareness.

Using partially reconfigurable hardware devices we are in the position to equip our devices with a virtually adaptive behaviour. To keep development easy at the beginning our nodes provide only one region of fixed size that can be reconfigured separately from the rest of the system as shown in Figure 3.

With this, we are able to realize a hardware-software codesign for optimized performance. Tasks that can efficiently be computed in hardware may be executed in a specialized hardware module. Other tasks that are not available as a hardware module or are more efficiently solved in software are then executed on the local CPU. We use the ICAP to allow the CPU to reconfigure the device itself to utilize free reconfigurable space or to replace unused hardware modules. The individual situational behavior of a device is determined by the controlling application which decides whether to replace a local hardware accelerator

or to employ a neighbouring node according to a set of parameters stored locally in simple files. These decisions are based on the analysis of the surrounding neighbourhood and constructing plans based on these information as explained in the introduction.

3. Targeted Application

Sensor networks used to consist of many very small sensor nodes to collect data and at least one data sink that gathers all the information and offers an interface to other networks. In scenarios with more complex data aggregations beacons are introduced which provide more computation power, collect data, and transmit preprocessed information to the destination.

Our targeted application is a sensor network that is able to track an object optically over a large area where several cameras have to cooperate to keep the target within sight. A sensor node is now equipped with a video camera and has a certain computation power to extract features of object recognition from the taken pictures. Information about what the cameras detect will be send to the user's terminal.

For this, we built up a sample application for the XUP development board that captures the video signal from a camera, digitizes it, and applies a filter to the stream to provide it for further processing. The whole task is solved in hardware on the FPGA with the filter being partially reconfigurable at run-time. Figure 3 shows the physical layout of our sample system. There the small boxed area is reconfigurable, whereas the bigger region encloses fixed logic like the Ethernet controller or the SD-RAM controller. Bus-Macros are represented between the reconfigurable and the static part (yellow). The lighter lines on the picture are wire connections between logic blocks.

Since our system is flexible and expandable, the loss of camera nodes or computation nodes (beacons) can be compensated to a certain degree in terms of surveyed

area as well as computation power for picture analysis. This is achieved by distributing the work the failing node contributed to other units. Also new camera nodes or beacons can be added as the task of the network and therefore the demand for computation power changes.

4. Monitoring System

We developed a monitoring system that consists of three parts. The main component of the system is the monitor daemon. This daemon permanently monitors the neighbourhood. The second part is an interface, which we have implemented in c++ and java for simplifying access to the daemon. The third and last component is a java application used to log into a daemon to display its knowledge about its neighbourhood.

4.1. Daemon. The monitor daemon runs in an uClinux environment as a background service. Other applications may access the daemon by using a simple interface. The task of the daemon is to permanently collect information from and about other devices in the neighbourhood and simultaneously broadcast its own state. Information a device may gather is

- (i) devices in the neighbourhood,
- (ii) their state (e.g., CPU load, memory usage, position, etc.),
- (iii) what capabilities they have (e.g., services they offer).

4.1.1. Device Discovery—Recognition of Surrounding Devices. The daemon periodically broadcasts a ping message to inform other devices, that it is still alive. This is done by every device in the network. When the daemon receives a ping message from another device A, it updates its list of known neighbours. If device A is not yet in the list, it is added to the list and a counter for that device is set. If device A is already in the list, the counter is reset to a standard value. Therefore the daemon keeps a list with all known neighbours with a counter value for each node. The daemon now periodically walks through the list in a specified interval (e.g., every 10 seconds) and decrements each counter. When a counter of a device, let us say device B, has reached a given limit, a ping message is sent to node B. If device B does not respond with a ping message within a given time, device B is removed from the list of known neighbours. In this case an event can be generated to inform other devices of this situation.

4.1.2. Service Discovery—Retrieving a Node's Capabilities. As well as the ping messages, each device in the network periodically broadcasts a list of information, that it is offering. For example, device A may have a list of its own hardware accelerators. Device A then broadcasts, that it is capable of sending a list of its hardware accelerators. Another device B may be able to give information about its state, like CPU load, memory usage, and so forth. Then device B periodically sends a message that other devices can retrieve the system state of device B. Therefore each node in the

network knows which information it can obtain from its surrounding nodes.

While ping messages are sent in quite a short interval, the lists are broadcasted less often, since we do not expect them to change very often in stable environments. It is thinkable that the frequency of broadcasting these lists is gradually adapted when a node detects an unstable environment (e.g., moving nodes).

4.1.3. Being Communicative—Obtaining Information. Obtaining information is done in a published/subscribed manner. If a device A is interested in receiving information from a device B, for example, the CPU load, device A subscribes to this information. Device B then periodically broadcasts the requested information. If another device C is interested in the same information, device C also subscribes to this information; however, device B recognizes the subscriptions being the same and only sends the information once every period. This mechanism minimizes network traffic.

Basically the daemon consists of two interfaces, a node-to-node (NTN) and a node-to-application (NTA) interface, a tiny database, and a set of plug-ins. The daemons communicate with each other using the NTN interface (see Figure 4).

If an application wants to get information from a daemon, it uses the c++ or java interface (as described in the next section) to connect to the NTA interface and collect the desired information. Using two interfaces, the complexity of the protocol for node-to-node communication can be kept small. The database is needed for storing information from neighbours and allows simple search queries. So far, we have explained how distribution of information is done in the network, but where do the information come from? Therefore the daemon uses a plug-in mechanism. For instance, there may be a plug-in that provides information about the available hardware accelerators, or a plug-in to catch system information from the underlying operating system. Additional plug-ins can easily be integrated in the system to enhance the daemons functionality.

4.2. The c++/Java Interface. The interface provides access to the daemon for information interchange. This is basically a c++ library which can be included by other applications, but also a lightweight java version has been implemented. Therefore the interface provides a basic set of functions to retrieve information from the daemon and to configure the way it monitors the surrounding environment. This, for instance, makes it possible to retrieve the list of known neighbours by a single function call. The communication between the interface and the daemon service is done by using UDP sockets. This allows connections to daemons that are more than one hop away by using a routing service or a lower level routing protocol. The complexity of retrieving information from nodes that are far away can hereby be taken out of the monitor daemon.

Getting information from the monitor daemon using the c++/java interface is done in a polling way. Therefore, if an application is interested in information (e.g., the hardware

FIGURE 4: Communication between monitor daemons using the NTN interface and with applications using the NTA interface.

accelerators of node A), it must explicitly call a function to retrieve the information. However, we have also implemented an event-mechanism to push critical information to connected applications. This might be the disappearance of a neighbour device A, which causes the system to reorganize itself to adapt to the changing environment. Therefore the daemon system may send crucial system events to any connected application.

4.3. Monitoring-Application. The last component of the monitor system is a java-based application. Using this application, it is possible to log into a monitor-daemon and display the daemon's knowledge of its environment (see Figure 5).

By graphically displaying the neighbouring devices with their information, we have developed a simple tool to observe the system-behaviour during run-time.

5. Local Marketplaces Global Symbiosis

We defined a simple concept to deploy our new features: LMGS—Local Marketplaces Global Symbiosis. The work is distributed according to principles of supply and demand within the network.

The simple idea is that every device does exactly what it deems to be the best according to its stored parameters. The minimal software to actively “take part in the game” consists of two elements: a customer which issues requests to the network to have a certain job done and a purveyor

that answers requests for jobs with the costs it charged if it would be commissioned. The effort a device makes to create the answer can vary widely. According to its computation power, knowledge, and storage capacity this can reach from a simple return of standard values to a complex measuring where the load, the utilization of the node's components, or the probability for the effectiveness of anticipated reconfiguration might be taken into account. In sensor networks, for example, communication is the most expensive action; so a distribution of work should be chosen that minimizes overall traffic, maybe by executing a lot of tasks locally through reconfiguring the node.

5.1. Requests. Requests issued by the customer component of a node consist of a tuple as follows:

$$\text{Request} = \{\text{Source}, \text{Target}, \text{Requester}, \text{Data_Volume}, \text{Task}, \text{Max_Hops}\}, \quad (1)$$

where *Source* contains the data source for the task, *Target* the data sink, and *Requester* the device which posted the inquiry. *Data_Volume* holds the quantum of data to be processed with *Task*. *Max_Hops* specifies the maximum number of times a request may be relayed by a node's neighbors.

Source, *Target*, and *Requester* may be partly or completely identical, given they are not, we included mechanisms to distribute jobs within a channel between data source and target. The values can be one-to-one identifiers of nodes or

FIGURE 5: Communication between monitor daemons using the NTN interface and with applications using the NTA interface.

might as well contain wild-cards to address groups of nodes so that, for example, modifications that have to be applied to a set of devices can be launched by a single command.

The *Data_Volume* will be taken into account when an offer is generated for the request. It usually influences the decision whether it is more appropriate to solve a task in software or in hardware and if the data may be processed remotely or if the communication costs for that would be too high.

In order to be able to specify a *task* or a whole group of tasks, we suggest an hierarchical nomenclature which comprises every single service that can be rendered in the network under one root node.

As *Task* in the request structure is not limited to one atomic job, it may be a list of tasks. These lists may contain jobs that are to be processed sequentially or in parallel reaching from a single data-source to a single target, to complex data flows with multiple sources and targets.

Finally the restriction to *Max_Hops* ensures that inquiries are not simply flooded through the net but stay local working off jobs at a kind of in situ marketplace when sensible. This is of course only if the local neighbors provide appropriate solutions to tasks at a reasonable price. The composition of this "price" will be presented in Section 4.2. The idea behind that is obvious: since we are able to reconfigure devices to serve virtually any need, even small localized groups are highly adaptable and will be able to cope with most challenges with optimal efficiency. Thus data will in general be kept in a spatially narrow cloud, and

communication is minimized. In the current and the next section we will explain how this is reconcilable with the claim of superregional cooperation.

To publish and find services to fill in a valid request basically three mechanisms can be deployed. First one central directory server knows which device offers which services and has to be prompted for every job to work off. If it fails, the whole network is paralyzed. New services have to be entered, causing additional traffic.

Second, searches for certain services are flooded through the whole network. This is very flexible, but rather inefficient.

In this work, we developed a third approach, a hybrid solution between the two previously mentioned ones. Here data are flooded only within the closest neighborhood. Only if the answers from them are insufficient, say because none of the next nodes wants to execute the requested task, the job is advertised again, this time with a higher number of maximum relays. Additionally devices keep a more or less extensive list of other remote hosts, that satisfy a certain request. In this manner not only local offers but also more distant ones will be taken into account. Distant ones possibly suit the current situation better. The maintenance of this list is closely related to the structure of offers replied to a request which will be covered in Section 4.2.

Generally, requests will be posed to the direct neighbors and to the issuer itself. A neighbor that finds the *Max_Hops* greater than one reduces that counter and sends the request to all its neighbours. Especially the answer from the node itself is interesting in this regard: with the possibility to

reconfigure, scenarios can be managed where communication cost is very high and nodes have to cut back on transporting lots of data through the net.

5.2. Offers. The response to a concrete query contains two elements: the cost-vector that the replying device is estimating for supplying the service and the local list of known providers that are also capable of satisfying this particular request:

$$\text{Offer} = \{\text{CostVectors}, \text{List_of_known_providers}\}. \quad (2)$$

Costs mean figures given as multipliers of a base cost. For example, the transmission of one byte of data over a wireless bluetooth connection will be a lot more expensive than over a wired Ethernet link.

We identified three dimensions of costly actions: time consuming, energy consuming, and space consuming. Thus a device's purveyor calculates the cost vector for a task A according to

$$C_A = \begin{pmatrix} Z_K & E_K & P_K \end{pmatrix}^T \cdot W, \quad (3)$$

where Z_K denotes the cost for the local effort concerning time, E_K for energy, and P_K for required space. W contains the willingness of the device to spend part of the specific resource to locally execute task A as a diagonal matrix. A battery driven device might, for example, want to lower its willingness to accept very energy consuming jobs as it runs low on battery power. The final cost-vector C_A is passed to the issuer as part of the offer.

The list of additional service providers is being built up through logging of messages indicating the commission of a node for a particular task or through deliberate writing. On the one hand devices that relay an acceptance message (basically a request with a specially formulated task) to a device store the target device and task together with an expiration time. On the other hand devices can advertise their capabilities by giving out a request to every other participant of the net to amend its local list of providers. If it intends to stay in the community for a longer period, the expiration time may be set accordingly, attracting all sorts of requesters, locally and remote.

5.3. Negotiation Example. The flow of a negotiation bases on sending requests, retrieving responses, determining the most appropriate solution, and commissioning a purveyor (Figure 6). When evaluating the replies, the contained lists of alternative, maybe remote, providers may be taken into account and selected ones may be prompted for a bid. This enables the system to incorporate both: decentralized and distributed computing as well as central services like the storage of gathered and processed data.

When enough answers came back in or after an amount of time, the customer determines the optimal partner to commission. The weighted cost including communication is therefore calculated according to

$$A : CC_A = \vec{C}_A \cdot \vec{G}_A = \sum_i C_{A_i} \cdot G_{A_i}. \quad (4)$$

FIGURE 6: LMGs: issue a request (top left picture), retrieve answers (top right picture), and commission job (bottom left picture).

Manipulating the components of G_A allows the device to emphasize a rather fast, energy-efficient or space preserving execution of a task. Depending on the computation-power and -willingness the node might accept the earliest offer or may run a multigoal optimization on the plenty of data retrieved to find the best trade-off.

6. Conclusions

In this work we have shown an approach for a distributed resource management based on self-organization which utilizes dynamically reconfigurable hardware to optimize the resource utilization in constrained networks.

Our work also contributes to dependability in networked embedded systems as disappearing nodes are easily detected, so that tasks can be reassigned to nodes willing to utilize their spare resources. To implement our approach we have developed a monitor system that collects information from the surrounding neighbourhood, that can be used for device discovery, service discovery, and the detection of disappearing nodes due to failures. For example, this system can be deployed to prototype self-organizing routing algorithms, load balancing, resource management, and so forth. Furthermore we have described our targeted application, which is a distributed intelligent camera system which consists of nodes built upon reconfigurable hardware. Our approach for a self-organizing resource management is based upon local marketplaces (LMGSs) where work is distributed

according to the principles of supply and demand within the network to optimize the resource utilization, especially that of the reconfigurable hardware.

References

- [1] S. Camazine, J.-L. Deneubourg, N. R. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau, *Self-Organization in Biological Systems*, Princeton University Press, Princeton, NJ, USA, 2003.
- [2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
- [3] J. W. Williams and N. Bergmann, "Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, T. P. Plaks, Ed., pp. 163–169, CSREA Press, Las Vegas, Nev, USA, June 2004.
- [4] Xilinx, "Virtex-5 FPGA Configuration User Guide," June 2009, http://www.xilinx.com/support/documentation/user_guides/ug071.pdf.
- [5] Xilinx, "Early Access Partial Reconfiguration User Guide," September 2008, http://kom.aau.dk/~ylm/rc/mm4-5/ug208_92.pdf.
- [6] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, "Design of homogeneous communication infrastructure for partially reconfigurable fpgas," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '07)*, Las Vegas, Nev, USA, June 2007.
- [7] Xilinx, "Virtex-5 FPGA User Guide," June 2009, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.