



Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825



Fakultät für **Informatik**

GI/ITG Fachgruppentreffen Betriebssysteme

12. Oktober 2007, Universität Karlsruhe (TH)

Herausgeber: Prof. Dr. Frank Bellosa

Interner Bericht 2007-23

ISSN 1432-7864

Inhaltsverzeichnis

NOVA und Bastei - Neue Ansätze Mikrokernel-basierter Systeme	1
<i>Stefan Kalkowski, TU-Dresden</i>	
Transparent, Power-Aware Migration in Virtualized Systems	3
<i>Jan Stoess, Christoph Klee, Stefan Domthera, Frank Bellosa, Universität Karlsruhe (TH)</i>	
Ein selbstoptimierendes Echtzeitbetriebssystem für verteilte selbstoptimierende Systeme	9
<i>Simon Oberthür, Carsten Böke, Franz Rammig, Universität Paderborn</i>	

NOVA und Bastei - Neue Ansätze Mikrokern-basierter Systeme

Stefan Kalkowski (TU-Dresden)

7. Oktober 2007

Die Betriebssystem-Gruppe der TU-Dresden befasst sich seit längerem mit verschiedenen Aspekten Mikrokern-basierter Systeme, wie insbesondere Sicherheit[10][7][6], Echtzeitfähigkeit[9][8][4] und Virtualisierung[5][2] bestehender Betriebssysteme und darauf laufender Software.

Aktuell wird im Rahmen des ROBIN-Projekts[1] an einer verbesserten Koexistenz von existierenden Betriebssystemen und einer Multi-Server Umgebung, auf Basis des neu entwickelten Hypervisors **NOVA**, gearbeitet. Die Multi-Server Umgebung **Bastei** wurde ebenfalls von Grund auf neu konzipiert und ist das vorläufige Ergebnis von zehn Jahren Forschung unserer Betriebssystem-Gruppe auf diesem Gebiet. In diesem Vortrag soll die Arbeit der TUD-Betriebssystem-Gruppe im Rahmen des ROBIN Projekts vorgestellt werden.

1 NOVA

Die NOVA OS Virtualization Architecture umfasst einen im Kernel-Modus laufenden, minimalen Hypervisor, sowie eine Virtual-Machine-Monitor Komponente, die im nicht-privilegierten Modus läuft. Mittels NOVA können unmodifizierte, vorhandene Betriebssysteme innerhalb virtueller Maschinen direkt neben Echtzeit- und sicherheitskritischen Anwendungen ausgeführt werden. Hierbei werden zur verbesserten Performance die neuen Virtualisierungsfunktionen derzeitiger x86 Plattformen ausgenutzt. NOVA wurde von Beginn an auf eine vereinfachte formale Verifikation hin entwickelt[11].

2 Bastei

Die Aufteilung vormals monolithischer Betriebssystemkerne in einen Mikrokern einerseits und mehrere, separierte Komponenten andererseits, sowie die Anforderung die *Trusted Computing Base* einer jeden Anwendung möglichst klein zu halten, um deren Sicherheit zu erhöhen, wirft die Frage auf, wie diese Komponenten zu gestalten und anzuordnen sind, um die gegenseitigen Abhängigkeiten minimal zu halten. Diese Frage versucht das C++ Framework Bastei[3] mittels einiger weniger, vorgegebener Komponenten und Protokolle zu beantworten.

Darüberhinaus versucht Bastei, neben herkömmlichen Zugriffs-Kontroll-Schutz-Mechanismen für die Gewährleistung von Vertraulichkeit und Integrität, durch "die Bezahlung" von Serverdiensten *Denial of Service* -Attacken zu begegnen. Die dezentrale Anordnung von Sicherheitspolitiken erlaubt ausserdem eine vereinfachte Administration auch komplexerer Systeme. Die Verwendung lokaler Namensräume ermöglicht die Re-Konfiguration bestehender Systeme ohne Komponenten neu übersetzen oder verlinken zu müssen.

Literatur

- [1] Robin project web page, October 2007.
- [2] Robert Baumgartl, Martin Borriss, Hermann Härtig, Michael Hohmuth, and Jean Wolter. Linux-Portierung auf den Mikrokern L4. *Wissenschaftliche Beiträge zur Informatik*, (1), 1996. In German.
- [3] Norman Feske and Christian Helmuth. Design of the Bastei OS architecture. Technical Re-

port TUD-FI06-07-Dezember-2006, TU Dresden, 2006.

- [4] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.
- [5] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.
- [6] C. Helmuth, A. Westfeld, and M. Sobirey. μ SINA - Eine mikrokernbasierte Systemarchitektur für sichere Systemkomponenten. In *Deutscher IT-Sicherheitskongress des BSI*, volume 8 of *IT-Sicherheit im verteilten Chaos*, pages 439–453. Secumedia-Verlag Ingelsheim, May 2003.
- [7] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The nizza secure-system architecture., 2005.
- [8] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 124–133, Austin, Texas, USA, December 2002.
- [9] Martin Pohlack, Ronald Aigner, and Hermann Härtig. Connecting real-time and non-real-time. Technical report, Technical University of Dresden, February 2004.
- [10] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: Three case studies, April 2006.
- [11] Henrik Tews, Bart Jacobs, Erik Poll, Marko van Ekelén, and Peter van Rossum. Specification and verification of the nova microhypervisor. Technical report, Raboud Universiteit Nijmegen, 2007.

Transparent, Power-Aware Migration in Virtualized Systems

Jan Stoess Christoph Klee Stefan Domthera Frank Bellosa

System Architecture Group, University of Karlsruhe, Germany

E-mail: {stoess, cklee, domthera, bellosa}@ira.uka.de

Abstract

This paper explores the use of hypervisor-based virtualization technology as a means to enable power management in server systems. Our approach focuses on the dynamic mapping of physical processors and hosts to virtual machines. We have developed a multi-tiered infrastructure that enables dynamic migration of virtual machine execution flow at two different levels: within and across computer nodes. Within a node, our infrastructure dynamically allocates and re-allocates virtual processors to their physical counterparts. Across nodes, our infrastructure employs live migration to relocate complete guest operating system instances to distinct physical hosts.

1 Introduction

Power and thermal management continue to emerge as critical factors in modern enterprise computing environments, and have evolved to a systemic challenge that needs to be addressed by all involved components, including the operating system (OS).

There exists a considerable body of research on OS-based power and thermal management. However, the monolithic structure of traditional OSes effectively hinders rapid integration of advanced power management strategies into mainline systems. Their lack of extensibility proves inadequate to respond to the demanding power and thermal challenges of modern computer systems.

Hypervisor-based Virtualization systems offer a way out of the dilemma. With their advantageous structure based on a small kernel and the rest of infrastructure running atop, they permit the whole OS stack to be designed with power and thermal management as inherent design criteria. Virtualization thereby permits the power management to be made available to the guest operating systems, but without depending on their particular instances – and while still being careful to maintain application isolation, a key property among many businesses.

This paper explores the use of hypervisor-based virtual-

ization technology as a means to enable power management in server systems. In particular, our approach focuses on the dynamic mapping of physical processors and hosts to virtual machines (VMs). We have developed a multi-tiered infrastructure that enables dynamic migration of VM execution flow at two different levels: within and across computer nodes. Within a node, our infrastructure dynamically allocates and re-allocates virtual processors (vCPUs) to their physical counterparts. Across nodes, our infrastructure employs live migration [5, 16] to relocate complete guest OS instances to distinct physical hosts.

There are plenty of power management goals that can be achieved using migration techniques; they typically fall into one of the categories workload consolidation or multi-core thermal balancing. Migration can also be combined with dynamic voltage and frequency scaling (DVFS) to yield even more power savings. We are currently working on integrating these algorithms into our prototype. We afterwards describe in detail concepts and implementation of our intra- and inter-node migration mechanisms.

Our multi-tiered migration prototype is based on the L4 micro-kernel as the hypervisor, and Linux 2.6 kernel instances running on top of it. For guest OS management, the prototype includes a user-level VM monitor (VMM) that provides the virtualization based on L4's core primitives. Our prototype supports virtual and physical multiprocessing on x86-based, medium-scale multiprocessing systems with up to 16 processors. The guest kernel instances run on dedicated L4 kernel threads, one per allocated vCPU. Whenever the guest kernel creates a new address space to run a task, the VMM spawns additional L4 threads for each vCPU, which serve as vessels executing the program code. When a guest OS kernel transfers control to the user level task, the VMM dispatches the representative L4 thread on that virtual processor.

For intra-node migration of vCPUs, our prototype dynamically changes the mapping of guest OS code to physical processors. Migration is transparent and does not involve the guest OS. L4 provides a kernel primitive to migrate a thread to a different processor. When migrating a vCPU, the VMM simply migrates all representative L4

threads. Also, in case the guest already has a different virtual CPU running on the destination processor, the VMM effectively avoids the thread migration, and merely switches the references to vCPU-specific data structures appropriately. Switching references is a cheap operation, as all vCPU-local state is accessed via a special processor segment. For synchronization and serialization, our prototype uses memory locks and L4's low-overhead cross-processor messaging functionality.

For inter-node migration, we have implemented a live VM migration facility capable of relocating the state of a VM to a different node. Before migration, the VMM suspends all threads associated with the VM, stores their execution state in a special memory object, and generates a snapshot of the guest physical memory. Via a special management VM, it then transfers VM memory and state across the network to the destination, where it is unmarshaled and brought to execution again.

As an initial evaluation, we have developed a thermal balancing policy for vCPUs of single guest OS instances. Based on energy profiles of individual vCPUS, which we estimate based on processor performance counters [1], our policy strives to prevent overheating by assigning vCPUs to physical processors in a way that the processor energy dissipations are equalized.

In the rest of the paper, we first present the design of our migration prototype in Section 2, and its initial performance evaluation in Section 3. We discuss related approaches in Section 4, and finally conclude in Section 5.

2 Design

The following section presents the core design of our multi-tiered migration prototype. We begin with describing the basic architecture of our migration facility. We then describe power management algorithms that decide when and where to migrate virtual CPUs or computers according to power or thermal considerations; we are currently working on integrating these algorithms into our prototype. We afterwards describe in detail concepts and implementation of our intra- and inter-node migration mechanisms.

2.1 Basic Architecture

Our multi-tiered migration prototype is based on the L4 micro-kernel and Linux 2.6 kernel instances running on top of it. It supports virtual and physical multiprocessing on x86-based, medium-scale multiprocessing systems with up to 16 processors.

As a minimalistic kernel endeavor, L4 only provides three basic kernel abstractions: threads, address spaces and inter-process communication (IPC); richer and more complex operating system functionality is implemented on top

of L4, at user level [18]. Although different in conception and goals [10, 14], micro-kernels can also serve as hypervisors for virtual machine systems, and there exist several approaches to provide virtualization on top of L4 [3, 13, 17].

We use a recent implementation of the L4 -kernel, code-named L4Ka::Pistachio. We will hence use the term L4 for both the abstract kernel and concrete implementation. The virtualization is based on a user-level virtual machine monitor (VMM) component running on top of L4, which provides the virtualization services based on the core primitives of the micro-kernel. For improved performance, the VMM is split into an in-place component running within the address-space of the guest OS, and an external module named resource monitor running in a separate address space, with extended privileges. A large fraction of the VMM code executes in place; only if unavoidable, for instance for reasons of security, the in-place part calls into the external module.

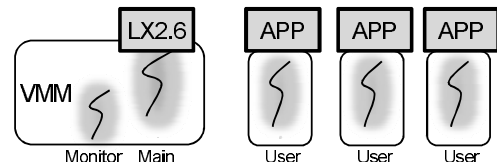


Figure 1. L4-Based Virtualization architecture

Our VMM maps each guest's virtual processor to a set of corresponding L4 threads (Figure 1, which serve as vessels for guest kernel and applications. The guest kernel is represented by two L4 thread, with one thread serving as the main context for the virtualized guest operating system code, and the other thread acting as the in-place resource monitor, exception handler, and scheduler of the main thread. To execute guest user code, the afterburner spawns an additional L4 thread per user level address space and virtual processor. Whenever the guest kernel transfers control to an application, the VMM on that virtual processor dispatches the appropriate L4 user thread.

2.2 Power-Aware Migration Algorithms

The primary goal of power management is to reduce energy and heat consumption of a computer systems. OS-directed power management thereby attempts to achieve that reduction by means of software running at the lowest layer of the computer system. Spatial migration of computation across processors and nodes bears plenty of opportunities for OS-directed energy management, particularly in the context of workload consolidation and heat reduction:

Workload consolidation. Migration can dynamically con-

solidate VMs or vCPUs during phases of underutilization, and re-allot them during phases of high load. Idle machines or processors are put into low-power sleep states, saving energy and avoiding server sprawl [2, 21].

Thermal balancing. Migration can balance heat production across cores, chips, or complete nodes. In combination with a profiling step determining heat characteristics of individual virtual CPUs or guest OSes, migration helps to either move hot execution streams to colder processors [7, 19], or conversely, to co-schedule execution streams that are complementary in their heat profiles, in order to remedy thermal hot spots [9]. Finally, core hopping policies can move executions streams across cores to distribute the heat over a greater area [15].

Combined Migration and DVFS. Emerging generations of x86-based processors will feature multiple clock and voltage domains, where frequencies and voltages of different cores and chips can be adjusted independently. Depending on the clock and voltage interdependencies of individual cores and the transition costs of frequency and voltage scaling, intra-node migration can dynamically arrange virtual CPUs among physical cores or chips, which, combined with dynamic frequency and voltage scaling, allows to actually conserve power. For instance, virtual processors can be spread among multiple spare cores, which are then run with slower voltage and frequency. As power and voltage are related in a cubic fashion, spreading computation saves power without losing actually performance.

2.3 Migrating virtual CPUs

In our L4-based virtualization architecture, each virtual CPU is represented by a set of L4 threads hosting the execution flow of that CPU. In order to migrate a virtual CPU, it is therefore principally sufficient to relocate all corresponding L4 threads to the destination processor. L4 already provides a system call to modify the particular processor a given thread should run on. Changing the processor will cause L4 to migrate the thread to a different processor instantly.

However, virtual CPU migration is expected to take place frequently, in the time frame of normal scheduling and load balancing intervals. Furthermore, single virtual CPU may consist of a magnitude of L4 threads, depending on the number of guest applications currently executing. L4 thread relocation is therefore a performance-critical factor in our migration facility, and we have developed two important improvements over the original L4 version, which enable our virtual CPU migration to scale well with increasing number of L4 threads: The first technique, *batch migration*,

extends the L4 interface to allow migrate of multiple threads in a single blow. The second technique, *pure user-level migration*, applies if the guest already has a set of representative threads on the destination processor; it then avoids the kernel-provided migration path and resorts to a scheme implemented completely at user-level.

2.3.1 Batch Migration

The current L4 version permits migration on a per-thread base only; to migrate multiple threads, the system call must be invoked several times subsequently. Such a solution has two serious implications on the migration performance: first, the migration path crosses the kernel-user boundary for every single thread; second, migration requires synchronized access to thread control blocks and other data structures, thus the kernel must issue cross processor interrupts, again for every thread. In presence of the substantial costs of system call transitions and interrupt handling on x86-based processors, such an implementation causes intolerable overhead when migrating multiple threads.

We have therefore developed a kernel-based batching migration scheme, which allows a user-level scheduler component to relocate multiple threads in a single shot. To migrate a set of threads simultaneously, the user-level VMM passes thread identifiers and their prospective destination processors to the L4 kernel migration system call. The kernel then constructs per source processor lists of the threads to be migrated and sends them, by means of its internal cross processor mailbox subsystem, to the respective processors (see Figure 2). Once notified, each source processor releases its local thread subset from the processor-local run queues, and updates all thread-local data structures appropriately. It then requests, again via cross processor messaging, the respective destination processors to integrate the migrated threads into their local queues. In contrast to the original migration scheme requiring a kernel-user transition and a inter-processor interrupt *per thread*, our new scheme requires only *a single* kernel-user transition and as many inter-processor interrupts as there are different source/destination processor tuples.

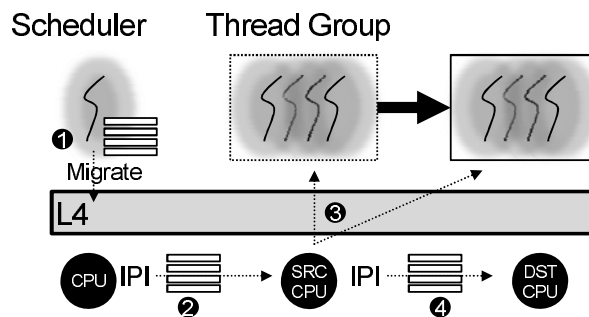


Figure 2. Batch migration

2.3.2 Pure User-Space Migration

Our second important optimization enables a pure user level implementation in case the guest already has another virtual CPU running on the destination processor. Our VMM then effectively skips thread migration and merely switches the user-level references to virtual processor specific data structures appropriately. Our VMM currently accesses all processor-specific data via a special processor segment set to a different value for each virtual processor. Under the presumption that two virtual processors run within the same address space, the VMM can switch the two processors' location by simply preempting the guest kernel threads at a well-defined code location, switching the reference to vCPU local data, and reactivating the threads again (see Figure 3).

In theory, pure user space migration a very simple and cheap operation, since it only requires exchange of a simple segment register and allows all L4 threads to stay on their original physical location. However, it also requires the execution stream of both virtual processors to be serialized, which we currently achieve by defining explicit points in the execution stream where the switching may take place. For the synchronization, we must use memory locks and L4's cross-processor messaging system. Furthermore, the pure user-level solution can only be performed between processors of the same guest OS, and only if the processors run within the same address space. This is only the case if, the VMM and guest kernel access vCPU-local data using an indirection scheme as described above, rather than private mappings and separate address spaces. For all other cases, we must resort to the default kernel-provided batch migration scheme.

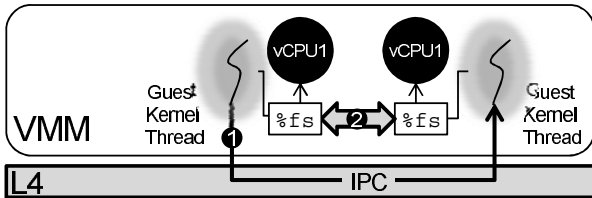


Figure 3. Pure User-Level Migration

2.4 Migrating virtual Computers

For inter-node migration, we have implemented a live VM migration facility capable of relocating the state of a VM to a different node. Our migration mechanism runs within the VMM, and no modifications to guest OSes are necessary. Our prototype currently supports the rather simple stop-and-copy migration; an effort to implement more elaborate pre-copy migration [5, 20] and to integrate live migration of virtual network devices [6] is underway. Stop-

and-copy migration basically consists of the three phases i) suspending the VM, ii) migrating the VM execution state to the destination node, and finally iii) resuming the VM on that node. The VM execution transferred during migration consists of the guest physical memory, the contents of the virtual processor registers, and VMM meta information such as L4 thread identifiers of guest kernel threads.

As described previously, the L4 VMM spawns a set of L4 threads per vCPU, to host kernel and application code. The threads of a guest application run within their own address space, which is constructed recursively from the address space of the guest kernel [18]. That is, whenever an application suffers a page-fault, the in-place VMM parses the guest kernel's page table hierarchy. If it finds a valid translation, it transparently inserts the translation into the application's address space, by means of L4's memory mapping primitives. When migrating the guest, it is therefore sufficient to transfer the guest physical memory, since it includes the guest's page table hierarchy. The VMM on the destination reconstructs the application's address spaces lazily, by again reading the page tables on page-faults and inserting the mappings when necessary.

Similar to the memory state, the execution state of the application threads is stored within the guest kernel's data structures – that is, in guest physical memory – and thus does not need to be migrated. It is therefore sufficient to transfer the execution state of the main kernel threads to the destination node. On the destination, the VMM will spawn new L4 threads and address spaces whenever the migrated the guest kernel tries to run an application that does not have a L4 thread representative yet.

However, previous approaches to live migration of an L4-based VM have shown, that quite a lot of cooperation with the micro-kernel is required to extract and insert valid execution to and from L4 threads [11, 12]. The root cause of that overhead lies within L4 itself: some parts of L4's thread control blocks that are required when checkpointing and restoring L4 threads – register frames saved on the kernel stack, and information on attempted or ongoing IPC operations, to give examples – cannot be extracted easily from the L4 directly; they are rather available to L4 itself only.

Similar to intra-node migration, we have therefore again added an enhancement to original L4 version that enables effective control over thread execution from user-level. A detailed description of our improved kernel version can be found in a different paper [22]. Here, we restrict ourselves to presenting the the improvements fundamental to thread migration. Our new L4 versions gives user level schedulers full control over dispatching, by vectoring out all thread preemptions to the user-level. As a result, there is only one thread running at a time per processor; all other threads are waiting to receive reactivation messages from user-level schedulers. Furthermore, our new L4 version propagates

all user-relevant execution state to the user-level, by means of special IPC messages. Conversely, a user-level resource manager can update thread execution state, also by means of a special IPC to that thread; L4 then installs the state update transparently into the thread's control block before activating that thread. Our VMM can therefore easily migrate a thread by checkpointing the exported execution state and transferring it to the destination; the VMM peer on the destination then spawns a new thread and associated address space, and reactivates the thread by sending an IPC containing the transferred execution state (see Figure 4).

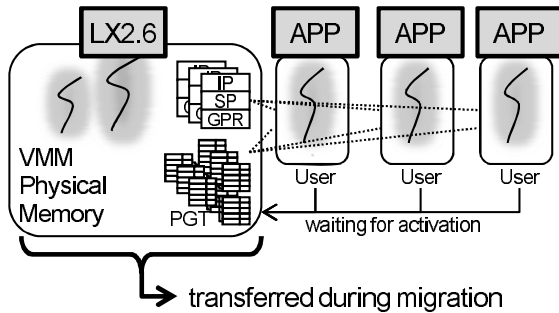


Figure 4. Preemption IPC and reply

3 Initial Evaluation

We have conducted initial measurements on a 3 GHz Pentium D830 with 2 cores and 2 GByte memory. The guest kernel executes on 2 vCPUs dynamically balanced among the two available cores. Table 5 shows the performance of a full kernel compilation and of the `netperf` benchmark run from an external client over a Gigabit NIC, under different re-balancing frequencies.

F [Hz]	KBuild [sec]	Netperf [$\frac{MB}{sec}$]
0	186	848.25
10	191	847.67
100	200	854.01
1000	206	851.24

Figure 5. Kernel Build and Netperf performance for different migration frequencies.

4 Related Approaches

Several research efforts focus like our approach on migrating or balancing computational load across cores or nodes in server systems and data centers. Except for VMware's recently announced distributed power management software [23], which is closed-source and unpublished

as of yet, all of those approaches focus on migrating or balancing tasks, jobs, or network streams; none of them has investigated virtual machine migration as a tool for energy and temperature management server systems.

The Load Concentration approach by Pinheiro et al. [21] proposes to distribute the load of server cluster in a way that hardware resources can be put in low-power modes. Load distribution is based on checkpointing and migrating whole applications running on a special version of the Linux operating systems. Similarly, Chase et al. propose to use reconfigurable switches to balance the network load offered to a pool of servers so that individual servers can be powered off [4].

Elnozay et al. propose and evaluate different voltage scaling policies for cluster power management in web server farms [8]. We believe that virtual machine migration could serve as a cluster reconfiguration mechanism that helps to extend the scope of such policies to other applications than specific web servers.

5 Conclusion

In this paper, we have proposed virtual machine migration as a means to pursue power management in virtualized systems. Our approach focuses both on migrating virtual among physical processors, and on migrating complete virtual machines among different nodes. We have developed a multi-tiered infrastructure that dynamically migrates virtual machine execution within and across computer nodes, and are currently exploring different power-aware migration schemes that help to preserve power as well as to keep the temperature of different processors and nodes balanced. Initial performance measurements indicate that our prototype is a promising approach to OS-directed power management.

References

- [1] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, New Orleans, LA, Sept. 2003.
- [2] R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11), Nov. 2004.
- [3] S. Biemueller and U. Dannowski. L4-based real virtual machines - an api proposal. In *Proceedings of the First International Workshop on MicroKernels for Embedded Systems*, Sydney, Australia, Jan. 2007.
- [4] J. Chase, D. Anderson, P. Thakur, and A. Vahdat. Managing energy and server resources in hosting cen-

- ters. In *Proceedings of the Eighteenth Symposium on Operating System Principles (SOSP'01)*, Oct. 2001.
- [5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation*, May 2005.
- [6] S. Domthera. Live migration of virtual network devices. Study thesis, System Architecture Group, University of Karlsruhe, Germany, July 2006.
- [7] J. Donald and M. Martonosi. Techniques for multi-core thermal management: Classification and new exploration. In *Proceedings of the 30th annual international symposium on Computer architecture*, Boston, MA, June 2006.
- [8] M. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proceedings of the Second Workshop on Power Aware Computing Systems*, Feb. 2002.
- [9] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Sept. 2004.
- [10] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.
- [11] J. G. Hansen and A. K. Henriksen. Nomadic operating systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 2002.
- [12] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.
- [13] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg. The performance of μ -kernel based systems. In *Proceedings of the 16th Symposium on Operating System Principles*, Saint Malo, France, Oct. 1997.
- [14] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *ACM Operating Systems Review*, 40(1):95–99, 2006.
- [15] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In I. Verbauwhede and H. Roh, editors, *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, Seoul, Korea, Aug. 2003.
- [16] O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *Proceedings of the USENIX 2007 Annual Technical Conference*, Santa Clara, CA, June 2007.
- [17] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.
- [18] J. Liedtke. On μ -kernel construction. In *Proceedings of the 15th Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.
- [19] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the 1st EuroSys conference*, Leuven, Belgium, Apr. 18–21 2006.
- [20] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX 2005 Annual Technical Conference*, June 2005.
- [21] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, Sept. 2001.
- [22] J. Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *ACM Operating Systems Review*, 41(4):59–68, 2007.
- [23] VMWare. *VMware Infrastructure 3 Key Features and Benefits Summary*. VMware, Inc., 2007.

Ein selbstoptimierendes Echtzeitbetriebssystem für verteilte selbstoptimierende Systeme *

Simon Oberthür
Heinz Nixdorf Institut
Universität Paderborn
Fürstenallee 11
33102 Paderborn
simon@oberthuer.net

Carsten Böke
Heinz Nixdorf Institut
Universität Paderborn
Fürstenallee 11
33102 Paderborn

Franz Rammig
Heinz Nixdorf Institut
Universität Paderborn
Fürstenallee 11
33102 Paderborn

ABSTRACT

Selbstoptimierende mechatronische Systeme passen sich an verändernde Bedingungen zur Laufzeit an. Statische Systemsoftware für eine solche Anwendungsklasse muss für den allgemeinen Fall konfiguriert sein und ist dadurch nicht optimal. Deshalb wurde ein als Multiagentensystem realisiertes selbstoptimierendes RTOS entwickelt, welches sich zur Laufzeit optimal an die sich dynamisch ändernden Anforderungen von selbstoptimierenden Anwendungen anpasst. Hierfür wurden Strategien auf RTOS-Ebene entwickelt, um die Dienststruktur dynamisch an die Anforderungen der Anwendungen anzupassen. Das System muss dabei starken Echtzeit- und Sicherheitsbedingungen genügen.

1. EINLEITUNG

Steigende Komplexität ist eines der Hauptprobleme von modernen mechatronischen Echtzeitsystemen, wie beispielsweise beim Automobil. Um diese Komplexität zu beherrschen, ist ein Ansatz diese Systeme mit „Self-X-Eigenschaften“ auszustatten, wie Selbstoptimierung [7], Selbstreflexion bzw. der Fähigkeit der Selbstanpassung. Solche dynamischen Anwendungen stellen besondere Anforderungen und Herausforderungen an unterliegende Hardware/Software-Schichten verteilter Echtzeitsysteme [13].

Für diese Anwendungen wird in dem Teilprojekt C2 des SFB614 „Selbstoptimierende Systeme des Maschinenbaus“, ein Echtzeit-Betriebssystem (RTOS) für verteilte selbstoptimierende Systeme bereitgestellt. Hierfür wurden grundlegende Konzepte für ein solches selbstoptimierendes RTOS entwickelt. RTOS-Dienste werden auf einem Berechnungsknoten (CPUs bzw. FPGAs) lokal optimiert und Ressourcen dynamisch unter den Anwendungen selbstoptimierend

*Diese Arbeit ist im Sonderforschungsbereich 614 – Selbstoptimierende Systeme des Maschinenbaus, Universität Paderborn – entstanden und wurde auf seine Veranlassung unter Verwendung der ihm von der Deutschen Forschungsgemeinschaft zur Verfügung gestellten Mittel veröffentlicht.

verteilt.

Als Basis für das selbstoptimierende RTOS wurde die konfigurierbare Betriebssystembibliothek DREAMS (Distributed Real-Time Extensible Application Management System) [5] verwendet. DREAMS war in der ursprünglichen Version lediglich offline konfigurierbar. Durch eine Konfigurationsbeschreibung wurden die konfigurierbaren Komponenten der Bibliothek ausgewählt. Nach dem Kompilieren des Betriebssystems waren somit nur die Komponenten und Eigenschaften in der konfigurierten Betriebssysteminstanz enthalten, die von der Anwendung benötigt werden. Techniken zur automatisierten Erzeugung einer solchen Konfigurationsbeschreibung wurden im Rahmen des DFG SPP1040 TERECS (Tools for Embedded Real-Time Communication Systems) [2] erarbeitet.

TEReCS unterscheidet strikt zwischen Wissen über die Anwendung und Wissen von Experten über das zu konfigurierende Betriebssystem. Informationen über die Anwendungen wurden über eine Anforderungsbeschreibung spezifiziert. Diese Spezifikation beschreibt abstrakt das Verhalten und Bedingungen (Zeitschranken) der Anwendung. Das Verhalten der Anwendung wird durch die verwendeten Systemaufrufe des Betriebssystems beschrieben. Es beschreibt, welche Systemprimitiven zu welchem Zeitpunkt benötigt werden. Zusätzlich werden in der Anforderungsbeschreibung die benötigte Hardware-Plattform und -Topologie spezifiziert.

In einer Wissensbasis wird der zulässige Entwurfsraum des konfigurierbaren Betriebssystems über die Abhängigkeiten der Dienste in einem UND/ODER-Graph vollständig beschrieben. In dieser Domänen-spezifischen Beschreibung des Betriebssystems sind Optionen, Kosten und Randbedingungen spezifiziert und führen so zu einer Überspezifikation, die alternative Implementationsvarianten enthält. Aus der Domänen-spezifischen Beschreibung wurde durch den Konfigurationsprozess mit Hilfe der Anforderungsbeschreibung der Anwendung der Entwurfsraum auf das benötigte Minimum verkleinert. Der Entwurfsraum enthält die gültigen Konfigurationen für die zu generierende Betriebssystem-Laufzeitplattform.

2. ECHTZEITBETRIEBSSYSTEM FÜR SELBSTOPTIMIERENDE SYSTEME

Das hier um Selbstoptimierung erweiterte RTOS DREAMS passt sich gegebenen Umständen (variierendes Anwendungs-

feld und/oder variierende Anforderungen der Anwendungen) dynamisch an. Hiermit ergibt sich ein System, welches effizient die Verwaltung von selbstoptimierenden Anwendungen unterstützt und entsprechende Dienstleistungen zur Verfügung stellt.

Die Forderung nach Effizienz beinhaltet, dass nur die tatsächlich benötigten Dienste in das Betriebssystem integriert sind. In einem selbstoptimierenden System mit veränderlichen Parametern, Algorithmen oder gar Strukturen ändern sich diese Anforderungen an zu erbringende Dienste aber dynamisch, was bei Beibehaltung der Forderung nach Effizienz eine Dynamik der unterliegenden Dienststruktur erfordert. Agentensysteme auf hohem Abstraktionsniveau und unter Echtzeit wirkende Reglersysteme haben stark unterschiedliche Anforderungen an ein Echtzeit-Betriebssystem (RTOS). Insbesondere aber die Vorgabe, dass sich Anwendungen selbstoptimierend an die Umgebung anpassen, führt zu stark variierenden Anforderungsprofilen.

Eine dynamische Adaption des RTOS an die Anwendung wird durch eine effiziente Instanziierung des Gesamtsystems, welches mit möglichst geringem Aufwand arbeitet, erreicht. Unterschiedlichste Anforderungen von Anwendungen zu disjunkten Zeitpunkten werden mit minimalen Ressourcen erfüllt. Durch das entwickelte selbstoptimierende RTOS werden nur die Anforderungen erfüllt und somit Dienste geladen, die aktuell benötigt werden.

Um ein dynamisch rekonfigurierbares RTOS aufzubauen, galt es folgende weiterführenden Fragestellungen zu beantworten: Zunächst musste der Entwurfsraum der Rekonfiguration des RTOS modelliert werden, um die Möglichkeiten der Anpassung des Systems an die Anwendungen zu modellieren. Die Rekonfiguration soll stattfinden, wenn sich die Anforderungen der Anwendungen verändern. Deshalb musste eine Schnittstelle zwischen Anwendung und Betriebssystem definiert werden, die den Austausch dieser Informationen ermöglicht. Mit diesen Informationen muss eine zu diesen Anforderungen adäquate Konfiguration des Systems ausgewählt werden, wozu ein Ressourcen-Management-System zu entwickeln war. Das von uns entwickelte System besteht daher aus den folgenden Modulen: *Profile Framework*, *Online-TEReCS* und *Flexible Resource Management*.

Insgesamt stellt das RTOS eine Menge konfigurierbarer RTOS-Komponenten zur Verfügung, die durch den Online-Konfigurator *Online-TEReCS* konfiguriert werden. Dieser wird durch Profile gesteuert, die durch ein *Profil-Framework* definiert werden können. Über das Profil-Framework werden auch die durch Deaktivierung von RTOS-Diensten frei gewordenen Ressourcen den Anwendungen zur Verfügung gestellt. Die strategischen Rekonfigurationsentscheidungen werden von dem o.a. flexiblen Ressourcen-Management-System getroffen.

2.1 Profile Framework

Als Eingabe für die Online-Konfiguration des RTOS wird der Bedarf an Ressourcen jeder ausgeführten Anwendung benötigt. Die Systemprimitiven des Betriebssystems werden hierbei ebenfalls als Ressourcen aufgefasst.

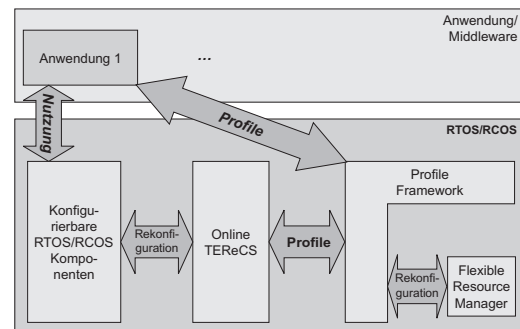


Figure 1: Überblick selbstoptimierendes RTOS DREAMS

Um die benötigten Dienste zur Bereitstellung der Ressourcen rechtzeitig zu instanzieren, ist es zusätzlich notwendig, dass Anwendungen den Bedarf möglichst frühzeitig mitteilen können. Je mehr Informationen das RTOS-System über die Anwendung besitzt, desto optimaler kann es sich auf die Anwendung einstellen. Deshalb ist es wichtig, eine Schnittstelle zur Anwendung zu schaffen, über die diese möglichst einfach zusätzliche Informationen bereitstellen kann.

Als Schnittstelle aus diesen Anforderungen ist ein Profile Framework [11] entwickelt worden. Mit diesem Framework kann eine Menge von Profilen pro Anwendung definiert werden. Je Profil wird eine Implementationsvariante der Anwendung mit einem unterschiedlichen Bedarf an Ressourcen beschrieben. Pro Ressource ist die maximale Belegung der Ressource für jedes Profil angegeben, die die Anwendung benötigt, wenn das Profil aktiviert ist. Jede Allokation von einer Ressource bedarf einer vorherigen Mitteilung an die Ressourcenverwaltung des Systems. Pro Ressource ist die maximale Verzögerung bei einer Allokation durch die Ressourcenverwaltung zu definieren. Zusätzlich muss definiert werden, zwischen welchen Profilen umgeschaltet werden darf bzw. kann. Hierbei wird unterschieden zwischen Profilwechseln, die von der Anwendung bzw. vom Ressourcenmanager initiiert werden. Das bedeutet, dass Anwendungen *On Demand* ihr Profil wechseln können. Um unter harten Echtzeitbedingungen zwischen den Profilen wechseln zu können, muss die Dauer der Rekonfiguration bekannt sein. Deshalb muss die maximale Dauer (WCET, Worst-Case-Execution-Time) bei vom Ressourcenmanager initiierten Wechseln zwischen Profilen angegeben werden. Jedes Profil erhält zusätzlich eine Gewichtung, die die Qualität der Implementationsvariante des Profils widerspiegelt.

Da selbstoptimierende Anwendungen sich mit der Zeit verändern können, können Profile dynamisch zur Laufzeit angelegt, entfernt und verändert werden. Der Entwurfs-Prozess von mechatronischen Systemen basiert heutzutage auf dedizierten Entwicklungswerkzeugen wie MATLAB/SIMULINK oder CAMEL-View. Durch Codegeneratoren wird automatisch Programmcode erzeugt. Die Modellierung von Anwendungen auf Basis des Profile Frameworks stellt, neben den Vorteilen, einen zusätzlichen Aufwand für den Entwickler dar. Um diesen Aufwand möglichst gering zu halten, wurde die Erzeugung von Profilen für selbstoptimierende Anwendungen in Entwurfstechniken und Entwurfsmethoden integriert [4]. So wurde ein Verfahren entwickelt, um aus er-

weiterten hybriden Real-Time Statecharts halb-automatisch Profile für eine Anwendungen zu generieren.

2.2 Online-TEReCS

Auf Basis des Offline-Konfigurators TEReCS wurde eine onlinefähige Version des Konfigurators entworfen [12]. Mit Hilfe dieses Konfigurators können sowohl die Möglichkeiten der Rekonfiguration wie auch die Abhängigkeiten der feingranularen und konfigurierbaren Dienststruktur des Systems beschrieben werden. Außerdem wird die eigentliche Rekonfiguration über den Konfigurator vorgenommen.

TEReCS wurde ursprünglich für die Offline-Konfiguration von RTOS-Systemen auf Quelltext-Ebene mit feingranularen optionalen Komponenten entworfen. Da eine feingranulare Rekonfiguration zur Laufzeit zu komplex ist, werden nun grobgranulare Strukturen rekonfiguriert. Hierdurch wird der Konfigurationsraum eingeschränkt und somit die Laufzeit für die Online-Auswahl einer Konfiguration minimiert. Die Möglichkeit, in TEReCS auf dem Entwurfsraum eine Hierarchie zu definieren, ermöglichte die Wiederverwendung des Konfigurators. So können auf Sourcecode-Ebene grobgranulare vorkonfigurierte Cluster bzw. RTOS-Agenten (z. B. Systemhierarchie, Speicherverwaltung, Scheduling, Kommunikation, Gerätetreiber, etc.) bereitgestellt werden, die zur Laufzeit ausgewählt werden. Ein RTOS-Agent umfasst dabei eng zusammenhängende Komponenten. Die Ableitungshierarchie sowie Komponenten der Klassenobjekte eines RTOS-Agenten sind hierbei zur Laufzeit statisch, nur die Abhängigkeiten der Aufrufe von Funktionen von Objekten anderer RTOS-Agenten können sich ändern. RTOS-Agenten können in verschiedenen Ausprägungen offline vorkonfiguriert werden. Der RTOS-Agent Systemhierarchie kann beispielsweise einmal mit und einmal ohne Unterstützung von Multi-Threading vorkonfiguriert sein. Im ersten Fall benötigt dieser Agent Funktionalitäten des Agenten Scheduling, im zweiten Fall werden diese Funktionalitäten nicht benötigt.

Jede Abhängigkeit von RTOS-Agenten untereinander wird auf interne Primitiven abgebildet. Diese Primitiven können von anderen RTOS-Agenten verwendet werden. So modellieren auf der höheren Hierarchie-Ebene die alternativen RTOS-Agenten wie im ursprünglichen TEReCS-Ansatz einen UND/ODER-Graphen. Zusätzlich enthält der Graph auch Implementierungsalternativen von Agenten, die unterschiedlichen Ressourcenbedarf haben, z. B. Implementierungen in Software und alternativ in rekonfigurierbarer Hardware.

Zur Laufzeit identifiziert der Konfigurator anhand der z. Z. benötigten Systemprimitiven, welche RTOS-Agenten benötigt werden. Konkret bedeutet dieses, dass die Dienste identifiziert werden, die für die Erfüllung der aktuell benötigten Systemprimitiven gebraucht werden. Hierbei wendet der Online-Konfigurator auf den UND/ODER-Graph der Ebene der RTOS-Agenten den selben Algorithmus an, wie der Offline-Konfigurator auf den UND/ODER-Graph der Menge der feingranularen Komponenten.

Informationen über die aktuell benötigten Systemprimitiven erhält das Online-TEReCS Modul indirekt über die aktivierten Profile der Anwendungen. Zu jedem RTOS-Agenten wer-

den zu den Implementationsvarianten entsprechende Profile angelegt. Somit wird für die RTOS-Agenten ebenfalls – wie für die Anwendungen – pro Variante ein Profil angelegt. Zusätzlich wird pro RTOS-Agent ein Profil (Sperr-Profil) angelegt, in denen die Ressourcen von TEReCS, die von dem aktiven RTOS-Agenten zur Verfügung gestellt würden, als vollständig belegt markiert sind. Dieses Sperr-Profil wird aktiviert, wenn der RTOS-Agent nicht ins laufende System konfiguriert ist und die Ressourcen, die von diesem RTOS-Agenten zur Verfügung gestellt werden, somit nicht für Anwendungen zur Verfügung stehen. Die Aktivierung dieser Profile – und damit die Aktivierung der Rekonfiguration des RTOS – erfolgt durch den Flexible Resource Manager. Hierdurch wird gewährleistet, dass nie ein Profil einer Anwendung aktiviert wird, ohne dass alle benötigten Dienste – und somit durch den Dienst angebotene Ressourcen – zur Verfügung stehen.

2.3 Flexible Resource Manager

Das Hauptziel des Flexible Resource Managers (FRM) [11] ist es allgemein, die Auslastung der Ressourcen zu optimieren und die Qualität des Systems zu steigern. Als Basis verwendet der FRM die Informationen aus den Profilen der Anwendungen und RTOS-Agenten. Durch Deaktivierung und Aktivierung von Profilen einer Anwendung bzw. von RTOS-Agenten nimmt er aktiv Einfluss auf das System. Mit Hilfe der Qualität der Profile und einer Gewichtung der Anwendungen wird eine Qualitätsfunktion des Systems aufgestellt, die der FRM maximiert. Die Komplexität des zugrunde liegenden Optimierungsproblems des Systems ist NP-vollständig, da der Lösungsraum exponentiell zu der Anzahl der Profile wächst. In der ersten Phase wurden daher zwei einfache Optimierungsheuristiken implementiert. Eine Strategie, die nur für kleine Mengen von Profilen anwendbar ist, bestimmt durch eine vollständige Suche die optimale Konfiguration von Profilen und versucht diese zu aktivieren. Die zweite Strategie ist eine einfache Heuristik, die von der aktuellen Konfiguration – durch Aktivierung weniger Profile – eine lokale bessere Konfiguration auswählt.

Ursprünglich war eine Simulation der zu aktivierenden RTOS-Agenten geplant. Da eine komplexe Simulation zu viele Ressourcen verschwendet – und somit nicht den Anwendungen zur Verfügung steht – beschränkt sich der FRM auf eine Simulation des Ressourcenbedarfs mit anschließendem Akzeptanztest. Hierbei wird überprüft, ob alle Echtzeitschranken der Anwendungen bei der Rekonfiguration eingehalten werden können und genügend Ressourcen für die Anwendungen zur Verfügung stehen.

2.4 Überbelegung von Ressourcen

Der FRM optimiert nicht nur die benötigten RTOS-Systemagenten, indem er die von TEReCS bereitgestellten Profile aktiviert bzw. deaktiviert, sondern stellt außerdem zugesicherte, aber zeitweise ungenutzte Ressourcen unter harten Echtzeitbedingungen anderen selbstoptimierenden Anwendungen zur Verfügung. Ressourcen können ungenutzt sein, wenn Anwendungen diese für Worst-Case Szenarien anfordern – aber zwischenzeitlich nicht benötigen – oder nur während Rekonfigurationen verwenden.

Durch den FRM wird eine weitere Ebene der Selbstoptimierung ermöglicht. Somit kann nicht nur innerhalb einer An-

wendung Selbstoptimierung oder eine speziell vordefinierte hierarchische Selbstoptimierung betrieben werden. Der FRM kann Ressourcen unter den Anwendungen vermitteln und somit die Qualität einer Gruppe von Anwendungen global erhöhen. Ein weiterer Vorteil ist, dass Selbstoptimierung auch zwischen Anwendungen erschlossen wird, zwischen denen keine explizite Abhängigkeit besteht. Durch die Modellierung im Profile Framework wird dies ermöglicht und eröffnet neues Potenzial zur Selbstoptimierung. Das RTOS passt sich somit nicht nur an die selbstoptimierenden Anwendungen an, sondern versucht die Qualität des Systems selbstständig zu optimieren, indem die Ressourcen möglichst so an die Anwendungen verteilt werden, dass die Qualität des Systems gesteigert wird. Als Grundlage für die Optimierung verwendet der FRM die erwähnte Qualitätsfunktion, in die die Qualitätskoeffizienten der Profile und eine Gewichtung der Anwendung eingehen. Liegt die Anwendung als Software-Implementation und als Variante in rekonfigurierbarer Hardware vor, aktiviert der FRM die aktuell qualitativ beste Implementationvariante.

3. ÄHNLICHE SYSTEME

Eingebettete Betriebssysteme [14], wie z. B. VxWorks [15], QNX [10], PURE [1] oder EPOS [8] haben eine feinkörnige Dienst-Architektur und ermöglichen eine auf die Anwendung angepasste Konfiguration der Dienste, womit sie für Architekturen mit geringen Ressourcen geeignet sind. Jedoch wird das dynamische Verhalten der einzelnen Knoten nicht unterstützt. Außerdem fehlt ein Mechanismus zur transparenten Kooperation zwischen den auf mobilen Knoten laufenden Tasks. Weiterhin sind solche Betriebssysteme nicht reflektorisch und können keine Selbstadaption auf neue Anforderungen der Anwendung oder Änderungen der Umgebung vornehmen.

Einige akademische Betriebssysteme wie etwa Apertos [16] können als reflektorisch bezeichnet werden. Diese reflektorischen (oder rekonfigurierbaren) Betriebssysteme haben die Möglichkeit, ihren aktuellen Zustand zu bewerten und darauf basierend ihre momentane Struktur zu ändern, um sich auf neue Anforderungen der Umgebung oder der Anwendung anzupassen. Ihre Dienste werden durch Objekte implementiert, die mit Meta-Objekten korrespondieren. Die Meta-Objekte analysieren das Objektverhalten und die Anforderungen der Anwendung zur Laufzeit und rekonfigurieren das Betriebssystem, um die Anforderungen besser zu unterstützen. Trotz dieser für das beschriebene Szenario wünschenswerten Eigenschaften haben die existierenden reflektorischen Betriebssysteme einen oder beide der oben erwähnten Nachteile: Sie wurden nicht konzipiert, um alle Charakteristika der beschriebenen Umgebung zu berücksichtigen, und bzw. oder stellen keine Transparenz für verteilte Anwendungen zur Verfügung.

Die Adaptive Resource Management (ARM) Middleware [6] stellt ein Optimierungsframework zur Beschreibung von verteilten Echtzeitsystemen für dynamische Umgebungen vor. Dieses Framework basiert ebenfalls auf einem Profilmodell und modelliert dynamische Veränderungen anhand von Ankunftszeiten, Last- und Dienstgüte. Die Middleware versucht, die Systemauslastung zu maximieren, indem eine gültige und optimale Ressourcenbelegung gesucht wird. Der Übergang zwischen Profilen ist durch atomare Ope-

rationen mit konstanter und vernachlässigbarer Umschaltzeit realisiert. Dieser Ansatz ist beispielsweise für Echtzeit-Bildverarbeitung möglich (ein Szenario, in dem ARM eingesetzt wird), indem atomar zwischen Bildverarbeitungsalgorithmen umgeschaltet werden kann. Für selbstoptimierende mechatronische Controller ist dieser Ansatz nicht anwendbar. Zwischen Controller-Algorithmen kann nicht immer atomar rekonfiguriert werden; sie verlangen Überblenden oder können nur zu bestimmten Zeitpunkten atomar umgeschaltet werden.

Der Dynamic QoS Manager (DQM) [3] implementiert ein Quality of Service Level Model für weiche Echtzeitanwendungen. Er ist als Middleware auf einem General-Purpose-Betriebssystem realisiert. Der DQM optimiert die globale Auslastung, indem er entsprechende QoS Level der Anwendungen aktiviert. Der QoS Level einer Anwendung charakterisiert den Ressourcenbedarf und den Nutzen bei Aktivierung des Levels. Der Ressourcenbedarf wird hierbei durch Messungen zur Laufzeit bestimmt. Dieser Ansatz ist nur für weiche Echtzeitsysteme anwendbar.

4. ZUSAMMENFASSUNG

Dynamische Anwendungen verlangen nach innovativen neuen Dienststrukturen auf der Ebene des Betriebssystems. Durch die vorgestellten Erweiterungen wurde unser Offline-konfigurierbares Echtzeitbetriebssystem DREAMS zu einem zur Laufzeit selbstoptimierenden Echtzeitbetriebssystem. Dieses deaktiviert ungenutzte Dienste und stellt freigewordene Ressourcen den Anwendungen zur Verfügung. Anhand einer selbstoptimierenden Anwendung und eines rekonfigurierbaren Echtzeitkommunikationsagenten wird das System in [9] genauer erläutert.

5. REFERENCES

- [1] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. On the development of object-oriented operating systems for deeply embedded systems - the pure project. In *ECOOP Workshops*, page 26, 1999.
- [2] C. Böke. *Automatic Configuration of Real-Time Operating Systems and Real Time Communication Systems for Distributed Embedded Applications*. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, 2004. ISBN 3-935433-51-4.
- [3] S. Brandt and G. J. Nutt. Flexible soft real-time processing in middleware. *Real-Time Systems*, 22(1-2):77-118, 2002.
- [4] S. Burmester, M. Gehrke, H. Giese, and S. Oberthür. Making mechatronic agents resource-aware to enable safe dynamic resource allocation. In *Fourth ACM International Conference on Embedded Software (EMSOFT'2004)*, 27 - 29 Sept. 2004.
- [5] C. Ditze. *Towards Operating System Synthesis*. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, 2000.
- [6] K. Ecker, D. Juedes, L. Welch, D. Chelberg, C. Bruggeman, F. Drews, D. Fleeman, and D. Parrott. An optimization framework for dynamic, distributed real-time systems. *International Parallel and Distributed Processing Symposium (IPDPS03)*, page

111b, April 2003.

- [7] U. Frank, H. Giese, F. Klein, O. Oberschelp, A. Schmidt, B. Schulz, H. Vöcking, and K. Witting. *Selbstoptimierende Systeme des Maschinenbaus - Definitionen und Konzepte.*, volume 155 of *HNI-Verlagsschriftenreihe*. Heinz Nixdorf Institut, Universität Paderborn, Paderborn, 2004.
- [8] A. A. Fröhlich and W. Schröder-Preikschat. Epos: An object-oriented operating system. In *ECOOOP Workshops*, page 27, 1999.
- [9] B. Griese, S. Oberthür, and M. Porrman. Component case study of a self-optimizing rcos/rtos system: A reconfigurable network service. In *Proceedings of International Embedded Systems Symposium 2005*, Manaus, Brazil, 15 - 17 Aug. 2005.
- [10] R. Oakley. QNX microkernel technology: A scalable approach to realtime distributed and embedded system. In *Proc. of the Embedded Computer Conference (ECC'94)*, June 1994. <http://www.qnx.com>.
- [11] S. Oberthür and C. Böke. Flexible resource management - a framework for self-optimizing real-time systems. In B. Kleinjohann, G. R. Gao, H. Kopetz, L. Kleinjohann, and A. Rettberg, editors, *Proceedings of IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'04)*. Kluwer Academic Publishers, 23 - 26 Aug. 2004.
- [12] S. Oberthür, C. Böke, and B. Griese. Dynamic online reconfiguration for customizable and self-optimizing operating systems. In *Fifth ACM International Conference on Embedded Software (EMSOFT'2005)*, 18 - 22 Sept. 2005. Jersey City, New Jersey.
- [13] F. J. Rammig. Autonomic distributed real-time systems: Challenges and solutions. In *7th International Symposium on Object-oriented Real-time Distributed Computing, ISORC 2004*. IEEE Computer Society, IEEE Computer Society Press, 12 - 14 May 2004.
- [14] J. A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3):237 - 253, Nov 2004.
- [15] W. R. Systems. Vxworks 5.4 - product overview, June 1999.
- [16] Y. Yokote. The apertos reflexive operating system: The concept and its implementation. In *OOPSLA Proceedings*, pages 414-434, 1992.