A Scalability-Aware Kernel Executive for Many-Core Operating Systems*

Gabor Drescher, Timo Hönig, Sebastian Maier,
Benjamin Oechslein, and Wolfgang Schröder-Preikschat

Friedrich-Alexander University Erlangen-Nuremberg
{drescher,thoenig,sebastian.maier,oechslein,wosch}@cs.fau.de

Abstract. Number, variety, and organization of the on-chip processing elements of many-core processors demand a radical rethink in operating systems design. One may come from a multitude of allocatable units that bestows every execution thread its own core: single-threaded cores will be usual, multi-threaded cores will be unusual. The paper presents a scalability-aware kernel executive, SAKE, that is currently designed against such background targeting at large-scale heterogeneous many-core systems. Benchmarks on a 48-core machine motivate custom system software and special purpose systems for such modern machines.

1 Introduction

Multi-core architectures with a fistful (2–8) of processing elements are actually yesterday's news in the parallel systems community, many-core architectures with 10^3 and more processors on a chip arise on the horizon. Based on [10] an annual increase of 40% for the number of cores per processor has been projected [5] that would bring chips of about 5×10^3 cores by the year 2020. Given the full configuration of Kepler GK110 (i.e., 15 SMX units of 192 cores each [15]), the promise of this projection, namely to have processors with up to 480 cores in 2013, has been already overachieved fivefold at the time being. However, it appears that assembly and operation of several thousands of cores on a single chip must be differentiated. Because of power constraints, the number of per-chip transistors that can switch at full frequency drops exponentially (*utilization wall*): large portions of chip area have to be left passive (*dark silicon*) in order to stay within the power budget of the chip [20]. Nevertheless, compared to the current state, the number of cores being plugged into future (general purpose) processors will increase significantly.

These sorts of parallel processors will be *heterogeneous* in terms of on-chip processing elements, communication facilities, and memory organization. But heterogeneity will also concern non-functional features such as (clock) speed and energy (demand), to enable overall system operation in ecological means. Shared

* This work was supported by the German Research Foundation (DFG), partly under grant no. SCHR 603/8-1 and by the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 823–832, 2014.

and distributed memory will coexist on a single chip. Analog with Intel's single-chip cloud computer (SCC [9]), at a certain level, cache coherence is no longer implemented in hardware. Despite dark silicon, with future many-core systems there will be so many cores available on a chip that every single thread will be able to run on its own private core. Single-threaded cores will be the common case, multi-threaded ones the exceptional case. All this calls for a radical change in the way operating systems manage processors; it also calls for programming and system paradigms suited for very fine-grained parallelism [18,11].

The most important aspect is to tackle *scalability* issues in particular by, but not limited to, the operating systems for these machines. In addition to analytical approaches, constructive methods are much-needed that decrease number and length of sequential program sections—ideally to zero. In particular, this means the continuous/exclusive use of *non-blocking synchronization* to prevent race conditions in cases where critical sections cannot be removed constructional by a clever arrangement of kernel-level data structures and program sections [14]. In addition to this fundamental design decision, a further measure is to rely on application-level control of sharing kernel-level data structures amongst concurrent processes [3]. Another option is to treat, at some higher level of abstraction, a many-core processor as a network of independent processing elements, assuming no inter-core sharing of in-memory data structures at the lowest level but rather bank on message passing for communicating processes even in case of an underlying shared-memory system [1]. Notwithstanding these commendable examples (Synthesis, Corey, Multikernel), they actually break new ground only rudimentary as the traditional view of processor multiplexing is still the dominating aspect when operating a single processing element (i.e., core)—and this is the primary challenge our approach tackles.

Scalable operating systems for many-core processors have to focus on *absolute parallelism* as a mandatory feature and consider pseudo-parallelism as an optional feature that might come into play for other reasons. The former is given only by means of real (i.e., physical) processing elements, whereas the latter implies their virtualization (e.g., in terms of a conventional threading concept). With this view, however, conventional threads do not constitute the adequate minimal subset of operating-system functions to implement concurrency. Rather, an *event-based approach* that especially suggests a single-stack kernel as basis for the many execution "threads" promises a much more efficiently operating parallel-computing platform [11].

Operating systems may employ cores in a *functionally dedicated* manner to hide latencies, reduce background noise, speed up system operations, but also for energy savings and heat abstraction. This encompasses system-call dispatching to idle or derated cores [21], offloading of interrupt and event handling, asynchronous I/O completion, or messaging assist [6], or even the serialized execution of critical sections in case of high contention [17,12]. Such a use of cores may be organized statically or dynamically in analogy to remote execution facilities developed for distributed systems in order to temporarily confiscate idle workstations [19].

This present paper is on a *Scalability-Aware Kernel Executive* (SAKE) for many-core processors that has an *interrupt model* at the basis of the design and ends in an event-based, single-stack implementation. In this model, which organizes related designs into members of an operating-system family, kernel functions (released due to system calls or traps) always run to completion, but may be overlapped by concurrent activities of the same core (due to interrupt requests) or caused by other cores. The run-to-completion principle particularly implies non-blocking synchronization of concurrent processes competing for (write access to) shared data structures, which happens *lock-free* [14, p. 2–3] in the present implementation variant of SAKE. The single-stack concept provides the basis for strong process locality while executing within the kernel and, thus, prevents stressing of the caches.

The remainder of this paper is structured as follows. Section 2 motivates the need for scalability-aware operating systems to meet the requirements for (future) heterogeneous many-core processors. This motivation is based on measurements of SAKE prototypes taken on a 48-core machine and their comparison to Linux. Section 3 presents our approach building on these prototypes. First, we describe how to statically analyze hot spots in the prospected energy demand of non-sequential programs in order to emit thread migration hints and recommendations for energy saving and heat abstraction purposes. In a second step we show how to exploit this information at run-time. For this we present the concepts and preliminary design of a scalable application run-time executive in terms of energy and heat demands. Finally, Section 3 draws some conclusions and briefly refers to future work.

2 Know Where the Shoe Pinches

A key issue of parallel programs destined for many-core processors is *scalability*. This is particularly true for operating systems. Utilizing today's parallel processing power is one of the recent challenges in computer science worldwide and the impact of the underlying operating system became increasingly apparent. Experiments with Linux 2.6 on a 16-core processor revealed a tremendous decline in performance by a factor of 40 once a second core was in charge of application processing [3]. Even worse, absolutely no performance increase was encountered when the number of cores increased gradually up to 16.

In order to support the SAKE approach we prototyped LAOS[1] to evaluate new system designs regarding performance and scalability on many-core machines. Our primary focus is on thread scheduling and synchronization on modern x86-64 machines. Two different operating-system kernels were implemented as members of the LAOS family. Both provide basic services such as dynamic creation of threads, thread scheduling and common synchronization primitives like semaphores, mutexes, condition variables and barriers. The members mainly differ in architectural matters: *LAKE* represents a novel multi-core event-based design [7] in terms of a *latency-aware kernel executive* and *MAOS* follows a rather

[1] `https://www4.cs.fau.de/Research/LAOS/`

conventional process-based design [13] of a *many-core operating system kernel*. To provide the operating system services in an inherently parallel environment, all kernel-internal synchronization is performed in a non-blocking fashion by employing atomic instructions of the processor such as compare-and-swap and atomic arithmetic instructions. No spinlocks or similar blocking primitives were used for intra-kernel synchronization. In a similar way deferred interrupt handling is established without blocking synchronization.

Both LAKE and MAOS implement the same core functionality. Hence, from a user's point of view, there are no functional differences observable at the system call layer. The strong argument for such an implementation kicks in with the non-functional property of parallel scalability. Performance measurements revealed a huge benefit in performance when compared to recent Linux/Glibc versions. Parallel application benchmarks were carried out, comparing LAKE, MAOS and Linux on the same machine executing an identical program. In the following, three benchmarks are described as well as the hard- and software environment.

All benchmarks were carried out on a 48-core system with AMD 6180SE processors that run at 2.6 GHz. The system comprises eight cache-coherent NUMA nodes and has an overall amount of 64 GB of RAM. The benchmarks for Linux were taken using kernel version 3.8.0 and the GNU C library 2.17. All benchmark programs were compiled with GCC 4.7.3, as were our two custom kernels.

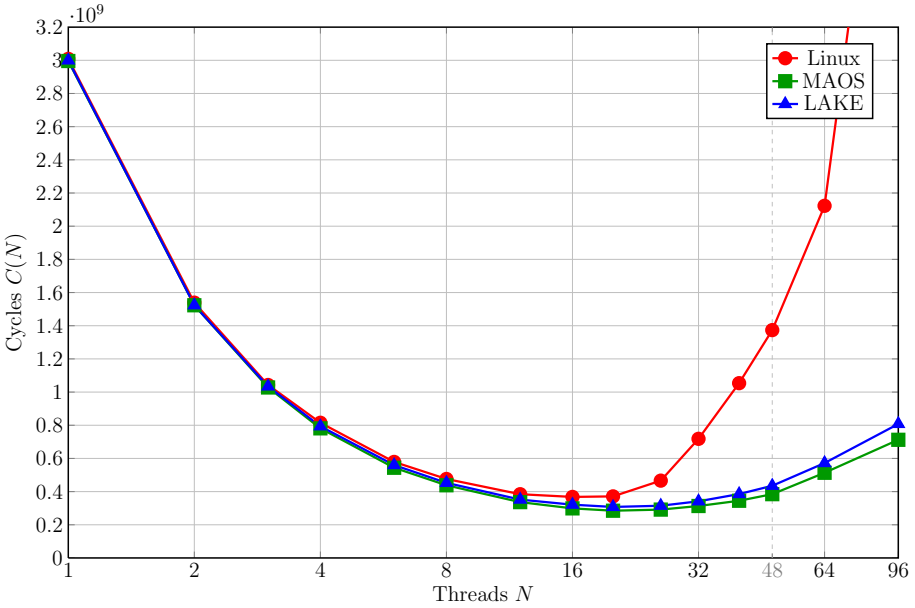

Fig. 1. Parallel iterative Jacobi benchmark, CPU cycles vs. number of threads

The *Jacobi* test uses the correspondent iterative method to solve a boundary value problem in parallel. The problem domain is geometrically split up into

Table 1. Maximum speedup in benchmarks

	Linux	MAOS	LAKE
Jacobi	8.18	10.52	9.75
Dijkstra	5.16	6.88	6.27
ServerClient	12.42	33.69	15.39

Table 2. Number of cores where performance saturates

	Linux	MAOS	LAKE
Jacobi	16	20	20
Dijkstra	12	16	20
ServerClient	16	40	48

stripes. Threads work in parallel on these stripes and blocking synchronization is used to communicate whether common values between neighboring threads and stripes are ready for access. In the *Dijkstra* benchmark the shortest paths between nodes in a weighted graph are computed. Each iteration of the algorithm is carried out in parallel and a synchronization barrier is needed to separate the iteration rounds. Performance in high contention scenarios on synchronization primitives is measured in the *ServerClient* test. Threads contend on mutex and semaphore operations that guard access to shared buffers. The number of buffers, number of threads, and ratio of producers and consumers is adjustable. Here, four producers and one consumer are created per CPU core and two global buffers are accessed by turns. All problem sizes are chosen to fit in the processor caches to eliminate main memory bottlenecks. Clearly, the benchmarks could be made more efficient with regards to synchronization, but our main focus here is on kernel level scalability. The results are comparable, since exactly the same benchmarks were performed on all systems.

All benchmarks show similar results and scaling behavior as observable in Table 1 and 2. Therefore, we only discuss the Jacobi benchmark in greater detail. In Figure 1 two observations can be made: First, the amount of system time with up to eight threads is negligible compared to the computation carried out by the user threads, regardless whether a fully featured Linux system or one of our systems is used. Second, increasing the number of threads reveals the scaling characteristics of the OS kernels themselves and the overall performance of the system is affected. The thread management overhead in Linux quickly leads to a performance drop, In contrast to this, both our kernels are able to push the maximum speedup of the algorithms forward and show a much slower performance degradation when the management overhead becomes prevalent (see Table 1 and Table 2).

These advantages support the aim to design and implement custom system software, or even special purpose systems, where exceptional performance or other superior non-functional properties are crucial. Such scalable kernels can serve as a basis for our application run-time executive as described in Section 3.2. This executive then provides an execution model for applications that in turn can be tuned towards optimizing for non-function properties like energy consumption by employing techniques covered in Section 3.1.

3 The SAKE Approach

In the following we describe the SAKE approach and its benefits for future operating system design. We outline the advantages of our approach over traditional commodity operating system designs, especially for special purpose operation scenarios in which parallel computing tasks are performed on many-core platforms featuring heterogeneous computing cores.

The SAKE architecture consists of two main components: A static code analysis component which exploits symbolic execution to determine prospective run-time behavior of program code and a scalability-aware run-time execution component that efficiently executes these program code snippets by evaluating the run-time hints previously deposited by the static code analysis component.

At the example of the ARM big.LITTLE [4] platform we demonstrate how to exploit unique characteristics of upcoming many-core processors most efficiently.

3.1 Proactive Hot-Spot Analysis of Energy Needs

Modern processors designs provide novel features to decrease the energy consumption most effectively. The introduction of new energy saving features has a significant impact on system software as they demand for sophisticated control mechanisms. This task is implemented at operating system level. Software-driven hardware features which are essential to increase energy-efficiency are lying fallow without such low-level support of the system software.

However, semiconductor companies choose different approaches to turn this endeavor into reality. Intel, for example, introduces new sleep states (C-states C6 and C7) for their latest generation of x86 processors (codename Haswell) which allows to reduce the energy consumption during idle periods to a tenth compared to the previous generation of Intel chips. In contrast to this, ARM currently works on multi-core chip designs (ARM big.LITTLE [4]) which feature heterogeneous processing cores that vary significantly with regards to their computing speed and energy demand. Compared to other multi- or many-core processors which have heterogeneous computing cores, the ARM big.LITTLE features heterogeneous computing cores that share the same architecture (ARMv7) but differ in various other aspects. The chip offers powerful Cortex-A15 cores (pipeline with 15 to 24 stages, out-of-order execution) as well as energy-efficient low-performance Cortex-A7 cores (pipeline with 8 to 10 stages, in-order execution). As the Cortex-A7 cores operate 2.3x to 3.8x more energy-efficient than the Cortex-A15 cores, operating systems need to ensure tasks are assigned properly depending on their performance requirements.

To support the operating system at correctly assigning tasks to CPU cores, we are analyzing program code proactively ahead of run-time to determine performance- and energy-hotspots of program code. For this procedure we use the SEEP framework [8] which exploits symbolic execution techniques in order to automatically extract possible program paths and their expected run-time behavior. During this code analysis we extract both, performance and energy requirements of program code at function level. To determine energy estimates

for program code, platform-specific energy profiles are being used. These energy profiles allow offline analysis (no manual energy measurements are required) which makes it possible to automate the energy analysis process. In parallel to the energy estimation analysis we are recording performance counter events to extract run-time characteristics. This information reveals the resource and performance demand of each individual code snippet.

The results of the analysis are deposited by means of run-time hints that are later on used by the run-time executive to assign tasks to the most suitable CPU core. Further, the task scheduler can merge these hints with run-time information to improve the task migration strategies during execution.

3.2 Concepts of a Scalability-Aware Run-Time Executive

The application-level execution model of SAKE running on top of the low-level OS-kernel is derived from OctoPOS [16]. It focuses on leveraging micro-parallelism in applications, hence, the run-time system has to deal with a potentially large number of possibly short execution snippets that in principal can be executed in parallel. This leads to the following requirements:

To cope with the large amount of entities, the representation of the individual pieces of a program has to be slim and efficient to both keep the size of the individual item in memory and its initialization time low. As execution snippets can be short, the overhead for switching from one snippet to the next also has to be kept low. Otherwise it would not pay off to split the application into small snippets, as switching overhead would start to dominate the overall run-time of the application program.

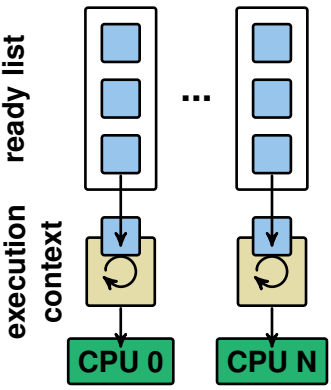

Fig. 2. Normal execution

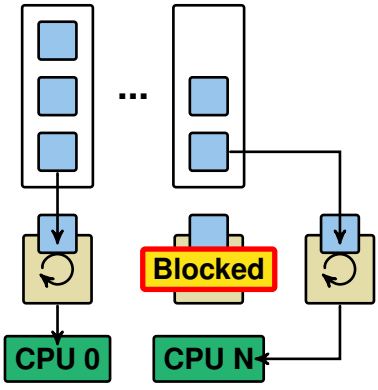

Fig. 3. Handling of blocking operations

Therefore, execution snippets are represented as function pointers with an associated data pointer used to store the arguments. In contrast to traditional threads, however, snippets should have run-to-completion semantics. This enables an implementation where there is no run-time state in form of a stack

associated with each snippet. Instead, as can be seen in Fig. 2, execution snippets one after another can be executed atop of execution contexts. Once a snippet finishes, which means its function returns, no state is left within the execution context. So the next item can be executed on the same context just by calling its function pointer without involving a rather costly context switch.

This scheme somewhat resembles Cilk [2] whose execution model enforces strict run-to-completion semantics for its procedures. It works fine as long as execution snippets have run-to-completion semantics and the application programmer takes care of ensuring this property. However, to ease programmability and to provide support for typical parallel design patterns like fork-join execution models, SAKE's run-time executive, however, allows blocking operations. As can be seen in Fig. 3 in the case an execution snippet performs a blocking call, for example to wait for other snippets to complete their execution, the associated execution stack cannot be used to execute the following snippet, as the execution state is still needed on resumption. In this case the executive provides a new stack to execute the following snippets and saves the context of the blocking snippet to the stack of the old context. Consequently, the cost for context switching only has to be paid when the application program really needs it.

The whole model can be implemented in a cooperative fashion, as all participants are entities of the same application. Hence, timing isolation between single execution snippets by employing techniques like time slicing is not necessary. This further contributes to an efficient implementation.

The granularity of execution snippets is application defined and generally provides more control to the programmer than a traditional threaded approach. Apart from exploiting micro-parallelism by massive parallelization, this also can be used to optimize for other, non-functional properties like energy awareness. To leverage the energy efficiency benefits provided by the big.LITTLE hardware platform energy properties for execution snippets can be precalculated by employing the techniques described in Section 3.1. and then attached to the description of the single execution snippets. At run-time, this data can then be used as hints to select a proper core for each single execution snippet. This enables the application programmer to optimize for energy efficiency on a per-function level.

During program execution there is no need for explicitly migrating a program to another core, as this is done implicitly by distributing the annotated snippets to the core best matching the energy profile considering the current condition of the hardware. When there are no snippets left for a core, it automatically enters a sleep state until new execution snippets are available.

4 Conclusion

With future many-core systems, single-threaded cores will be the rule and multi-threaded ones the exception. This calls for a radical change in the way operating systems manage processors respectively processor cores. A key issue of parallel programs destined for many-core processors is *scalability*, and this is particularly true for operating systems. Constructive methods are much-needed that decrease the portion of sequential program sections ideally down to zero.

The SAKE approach presented in the paper has its focus on constructive measures that mainly concern architectural features of operating-system kernels designated to run (massively-parallel) many-core processors. Confirmed by experiments on a 48-core machine with prototyped event-based and process-based operating-system kernels, specialization is the preferred way to go in order to best exploit hardware features in favor of excellent application performance. Linux, by way of example of a prominent representative of an operating system that transforms a many-core processor into a general-purpose (abstract) machine, easily becomes second quality as far as scalability is concerned.

We aim at improving performance of operating-system kernels in strictly application-oriented manner. Particularly this includes measures for energy-efficient execution of software to enable overall system operation in ecological means. SAKE bases on static program analysis to automatically extract run-time hints for energy-aware parallel processing by the kernel. In addition, the kernel complements this with a straw-weight processing model of run-to-completion execution snippets of blocking capabilities to efficiently support fork-join patterns of standard parallel programs—"SAKE wa hyaku-yaku no ch$\bar{o}$."

References

1. Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The Multikernel: A new OS architecture for scalable multicore systems. In: Matthews, J.N., Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating System Principles (SOSP 2009), pp. 29–44. ACM, New York (2009)
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Ferrante, J., Padua, D., Wexelblat, R.L. (eds.) Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 1995), pp. 207–216. ACM, New York (1995)
3. Boyd-Wickizier, S., Chen, H., Chen, R., Mao, Y., Kaashoek, M.F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., Zhang, Z.: Corey: An operating system for many cores. In: Draves, R., van Renesse, R. (eds.) Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), pp. 43–57. USENIX Association, Berkeley (2008)
4. Brian, J. Advances in big.LITTLE technology for power and energy savings. Tech. rep., ARM Ltd. (2012)
5. Brorsson, M.: Scalability and programmability in the manycore era. Draft Synopsis for an EU FP7 STREP Proposal (January 2009)
6. Brüning, U., Giloi, W.K., Schröder-Preikschat, W.: Latency hiding in message-passing architectures. In: Siegel, H.J. (ed.) Proceedings of the 8th International Symposium on Parallel Processing (IPPS 1994), pp. 704–709. IEEE Computer Society, Washington, DC (1994)
7. Drescher, G. Applying eventbbased design principles to a many-core OS kernel. Master's thesis, University of Erlangen-Nuremberg (2012)
8. Hönig, T., Eibel, C., Kapitza, R., Schröder-Preikschat, W.: SEEP: Exploiting symbolic execution for energy-aware programming. In: Proceedings of the Fourth Workshop on Power-Aware Computing and Systems (HotPower 2011), pp. 17–22. ACM, New York (2011)

9. Intel Labs. The SCC platform overview. Rev. 0.80, Intel Corporation (January 2012)
10. IRTS. International technology roadmap for semiconductors (2007), http://www.itrs.net/Links/2007ITRS/Home2007.htm
11. Knauerhase, R., Cledat, R., Teller, J.: For extreme parallelism, your OS is sooooo last-millennium. In: Proceedings of the 4th USENIX Workshop of Hot Topics in Parallelism (HotPar 2012). USENIX Association, Berkeley (2012)
12. Lozi, J.-P., David, F., Thomas, G., Lawall, J., Muller, G.: Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In: Heiser, G., Hsieh, W. (eds.) Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 2012), pp. 65–76. USENIX Association, Berkeley (2012)
13. Maier, S.: Leveraging non-blocking synchronization in a process-based many-core OS kernel. Master's thesis, University of Erlangen-Nuremberg (2013)
14. Massalin, H., Pu, C.: A lock-free multiprocessor OS kernel. Tech. Rep. CUCS-005-91, Department of Computer Science, Columbia University, New York, NY 10027, USA (June 1991)
15. NVIDIA Corp. NVIDIA's next generation CUDA compute architecture: Kepler GK110. Whitepaper V1.0 (2012)
16. Oechslein, B., Schedel, J., Kleinöder, J., Bauer, L., Henkel, J., Schröder-Preikschat, W.: OctoPOS: A parallel operating system for invasive computing. In: McIlroy, R., Sventek, J., Harris, T., Roscoe, T. (eds.) Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA 2011), USB Proceedings of Sixth ACM European Conference on Computer Systems (EuroSys 2011), pp. 9–14 (2011)
17. Suleman, M.A., Mutlu, O., Qureshi, M.K., Patt, Y.N.: Accelerating critical section execution with asymmetric multi-core architectures. In: Soffa, M.L., Irwin, M.J. (eds.) Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIV), pp. 253–264. ACM (2009)
18. Teich, J., Henkel, J., Herkersdorf, A., Schmitt-Landsiedel, D., Schröder-Preikschat, W., Snelting, G.: Invasive computing: An overview. In: Hübner, M., Becker, J. (eds.) Multiprocessor System-on-Chip: Hardware Design and Tool Integration, ch. 11, pp. 241–268. Springer Science+Business Media, Springer, New York (2011)
19. Theimer, M., Lantz, K.A., Cheriton, D.R.: Preemptable remote execution facility for the V-System. In: Baskett, F., Birrell, A., Cheriton, D.R. (eds.) Proceedings of the Tenth ACM Symposium on Operating System Principles (SOSP 1985), pp. 2–12. ACM, New York (1985)
20. Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., Taylor, M.B.: Conservation cores: Reducing the energy of mature computations. In: Hoe, J.C., Adve, V.S. (eds.) Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV), pp. 205–218. ACM (2010)
21. Wentzlaff, D., Agarwal, A.: Factored operating systems (fos): The case for a scalable operating system for multicores. ACM SIGOPS Operating Systems Review 43(2), 76–85 (2009)