Effective Runtime Resource Management Using Linux Control Groups with the BarbequeRTRM Framework

PATRICK BELLASI, GIUSEPPE MASSARI, and WILLIAM FORNACIARI,
Dipartimento di Elettronica Informazione e Bioingegneria (DEIB), Politecnico di Milano

The extremely high technology process reached by silicon manufacturing (smaller than 32nm) has led to production of computational platforms and SoC, featuring a considerable amount of resources. Whereas from one side such multi- and many-core platforms show growing performance capabilities, from the other side they are more and more affected by power, thermal, and reliability issues. Moreover, the increased computational capabilities allows congested usage scenarios with workloads subject to mixed and time-varying requirements. Effective usage of the resources should take into account both the *application requirements* and *resources availability*, with an arbiter, namely a resource manager in charge to solve the resource contention among demanding applications.

Current operating systems (OS) have only a limited knowledge about application-specific behaviors and their time-varying requirements. Dedicated system interfaces to collect such inputs and forward them to the OS (e.g., its scheduler) are thus an interesting research area that aims at integrating the OS with an ad hoc resource manager. Such a component can exploit efficient low-level OS interfaces and mechanisms to extend its capabilities of controlling tasks and system resources. Because of the specific tasks and timings of a resource manager, this component can be easily and effectively developed as a user-space extension lying in between the OS and the controlled application.

This article, which focuses on multicore Linux systems, shows a portable solution to enforce runtime resource management decisions based on the standard control groups framework. A *burst* and a *mixed workload* analysis, performed on a multicore-based NUMA platform, have reported some promising results both in terms of performance and power saving.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management

General Terms: Algorithm, Design, Performance, Management

Additional Key Words and Phrases: Runtime resource manager, Linux, control groups, performance counters, scheduling, multicore, many core, parallel applications, power, reconfigurability

ACM Reference Format:
Patrick Bellasi, Giuseppe Massari, and William Fornaciari. 2015. Effective runtime resource management using Linux control groups with the BarbequeRTRM framework. ACM Trans. Embedd. Comput. Syst. 14, 2, Article 39 (March 2015), 17 pages.
DOI: http://dx.doi.org/10.1145/2658990

1. INTRODUCTION

Progress in the silicon technology process has enabled the design and the manufacture of computing platforms, featuring levels of parallelism ranging from a few to hundreds

This work has been partially supported by the EU-funded projects HARPA FP7-ICT-2013-10-612069 and CONTREX FP7-ICT-2013-10- 611146.
Authors' addresses: P. Bellasi, G. Massari, and W. Fornaciari, Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20132 Milano, Italy; emails: {patrick.bellasi, guiseppe.massari, william.fornaciari}@polimi.it.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 1539-9087/2015/03-ART39 $15.00
DOI: http://dx.doi.org/10.1145/2658990

of processing elements. Parallel architectures have found market not only in the scope of high-performance computing (HPC) but also among high-end embedded and mobile systems. Nonfunctional constraints, such as area, power, and thermal management, had always been typical design issues of such systems, but today's embedded systems are losing this sort of exclusiveness. Now more than ever, energy efficiency and thermal management, system reliability, and fault-tolerance capabilities are requirements of distributed computing systems (i.e., server farms). Disregarding these requirements would trivially lead to a dramatic increase in maintenance costs, frequency of failures, and loss of performance. As a consequence, it is mandatory to implement effective multiobjective resource management policies to properly exploit modern computing platforms, taking into account the preceding constraints.

General-purpose operating systems (OS) usually aim at keeping the kernel space as efficient as possible, pushing the implementation of complex frameworks toward user space. Therefore, it is reasonable to think about integrating the OS with a resource management framework running as a daemon process. However, this framework requires the control of system resources, which can be achieved only by suitable interfaces and mechanisms provided by the OS itself.

This article focuses on mechanisms supporting runtime resource management on multicore Linux systems. To assess the effectiveness of the proposed solution, experiments have been done targeting a multicore NUMA platform made of four quad-core processor nodes. Two analyses have been carried out, based on two benchmark parallel applications, featuring very different implementation patterns.

The rest of the article is structured as follows. Section 2 briefly outlines related works, and Section 3 provides a short background on control groups, along with the motivations for choosing it. Section 4 summarizes the basics of the proposed runtime management framework, with a detailed description about how control groups have been exploited. Section 5 reports the results of the experiments carried out through two different types of analysis. Finally, conclusions are drawn in Section 6.

2. RELATED WORKS

A class of resource management solutions focuses mainly on the problem of task scheduling, with the goal to optimize the performance/power trade-off.Several proposals, targeting Linux systems, aim at extending the task scheduler of the OS by adding new scheduling classes, such as in Domjan and Gross [2001], Bini et al. [2011], Fu and Wang [2011], and Li et al. [2007]. These proposals focus on specific application domains and are tied to the adoption of customized OS kernels. Conversely, one of the main target requirements focused in this wo[rk is the *portability* of the resource management proposal.

In this regard, a more portable class of solutions includes all of those based on a user-space implementation. In most of them, the resource management actions are limited to the assignment of the set of CPU cores on which each active task is allowed to run, as in Hoffmann et al. [2010], Hofmeyr et al. [2010], Blagodurov and Fedorova [2011], Louvel et al. [2011], and Sondag and Rajan [2011]. Most of these solutions leverage the `sched_setaffinity` syscall to set the CPU affinity mask.

Another class is that of resource managers based on the concept of virtualization, namely the idea of hosting multiple virtual execution environments on the same physical device/machine.This implies the partitioning of the system resources among the virtual environments and the implementation of a hypervisor acting as a global system manager. This is the case of Head et al. [2010], Keller and Lutfiyya [2010], and Ranadive et al. [2011]. Actually, the virtualization is already adopted by both open source and commercial solutions, such as OpenVZ [2010], Linux Vserver [2010], Linux Containers [2010], and Monta Vista [2010]. In particular, some of them are built on top of Linux

Table I. Control Groups Subsystems

Subsystem	Description
blkio	Input/output access to/from block devices
cpu	Control on CPU usage assigned by the scheduler
cpuacct	Report on CPU usage
cpuset	CPUs and memory nodes assigned
devices	Access control to devices
freezer	Suspend/resume tasks
memory	Limits on usage of memory
net_cls	Tagging of network packets for traffic control
net_prio	Priority of network traffic per network interface
ns	Namespace isolation

Control Groups [2006] to setup the execution environment. However, in all of these solutions, resource partitioning is “statically” defined ahead of system deployment.

As a novel contribution, a more dynamic exploitation of control groups is proposed for Linux-based multicore systems, which is used as an interface to build a user-space portable and modular runtime resource manager (RTRM).

3. LINUX CONTROL GROUPS

The control groups framework [Linux Control Groups 2006] has been part of the Linux kernel since version 2.6.24. It offers the possibility to group tasks and bind each group to a subset of system resources such as CPUs, memory quota, and I/O bandwidth. In general, a group of tasks can be configured to use a specific set of resources either in a shared or exclusive way. The former option allows overlapping among resource subsets, whereas the latter can be used to configure a set of isolated execution environments, which can be conveniently used to setup a lightweight virtualization solution. Resource partitioning criteria can be either task- or user based. In the second case, for example, a system administration can establish the amount of resources to reserve to the tasks of a user according to its profile. This feature has made control groups mainly attractive in the context of multiuser remote servers and distributed systems.

This article introduces a different use of the framework, going beyond the common view of an administration tool. The following are some of the key aspects that led us to choose this framework as a mechanism to enforce resource management decisions:

—*User-space interface to control the system resources*. This is a strong point in terms of portability and invasiveness of the resource management solutions, as no kernel side modifications are required.
—*High modularity, thanks to a collection of subsystems also known as controllers, leveraging specific OS managers*. This aspect allows optional focus on the control of only a subset of resources classes. Table I lists the currently available subsystems with the resource control capabilities on tasks of control groups.
—*Wide set of control parameters added by subsystems*. This makes control groups the most complete interface toward the system resources, which gives great potential to a runtime resource management framework.
—*Gaining more and more attention*. The number of subsystems included is growing, and consequently the capabilities to control tasks execution and resource assignment.

To better understand how the framework has been exploited, a brief description of the proposed solution to runtime resource management is presented in the following section.

4. RUNTIME RESOURCE MANAGEMENT

The proposed solution has been implemented and made available in the BarbequeOpen Source Project (BOSP).[1] The project includes the RTRM—that is, BarbequeRTRM [Bellasi et al. 2012]—along with a set of benchmarks, sample applications, and libraries.

The rest of this section provides an overview of the framework. As stated previously, its modularity and portability aims at allowing BarbequeRTRM to support different hardware platforms. In the following sections, only the case of Linux-based multicore systems is targeted. Furthermore, for the sake of completeness, the concept of runtime reconfigurable application is introduced by explaining how it has been characterized and implemented in the proposed solution.

4.1. Overview

The basic notion of the framework is that an effective use of system resources starts from the capability of *applications* to adapt their execution to the amount and status of the assigned resources. Therefore, the framework provides support for implementing applications that are runtime reconfigurable. To accomplish this, the applications must (1) integrate the execution model defined by the library (RTLib) provided with the the framework and (2) define a finite set of possible runtime configurations.

Regarding runtime configurations, two different levels of reconfiguration and corresponding configuration information have been identified. The first level identifies the application working modes (AWMs), which define the resource requirements for the achievement of a certain QoS level. The second level is represented by the set of operating points (OPs). A single OP is characterized by a set of application-specific parameters affecting either the quality of the computation or the application performance, but characterized by values ranging only in the boundaries defined by the current set of assigned resources (AWMs).The set of application configurations is defined via a design-time profiling that is supported by design space exploration (DSE) tools. The effectiveness of these techniques has been already proved by prior works (e.g., Yang and Catthoor [2003], Ykman-Couvreur et al. [2005], and Mariani et al. 2010]) that show how DSE is helpful in supporting runtime resource management.

Modern SoC multicore platforms are based on a hierarchical organization of homogeneous computational resources. Accordingly, the proposed resource manager handles the concept of cluster to reference a group of processing elements, usually sharing a local cache memory, also referenced with the term *node* in the domain of NUMA machines.

Given this scenario, the objective of a resource allocation policy is to identify a suitable partitioning of the available resources among the demanding applications, considering (1) the requirements of each application, (2) the status of the hardware resources, and (3) the set of system-wide optimization goals. This is pursued by selecting the "best" AWM for each active application and mapping it on a cluster of resources according to the optimization goals. The application is notified at runtime about its AWM, and hence the corresponding assigned amount of resources, so that, for example, it can tune accordingly its parallelism level. Optionally, the application can exploit an API provided by the RTLib to configure an application-specific RTRM by simply defining a set of goals that would lift the programmer from the burden of selecting an OP. Further details about this point will not be provided, as it is outside the scope of the article.

4.2. Resource Allocation Policy

This subsection introduces a resource allocation policy implemented in the framework. Being an NP-hard problem, a resource allocation heuristic has been designed and

[1]For more information: http://bosp.dei.polimi.it.

implemented. This heuristic is structured in two main steps: *ordering* and *selection*. The ordering step includes the evaluation of each pair $< W_j, C_k >$ for each active application. The evaluation is performed by a multiobjective function, which carries out an index value. This output represents the profit related to the choice of mapping an AWM in a given cluster. Please note that from now on the expression *evaluation metrics* is used in place of *profit*. The evaluation metrics are then exploited to build a descendant ordered list of the whole set of pairs (from each application).

The selection step iterates on the ordered list, picking the pair $< W_j, C_k >$ on top, and assigning it to the application to which it belongs. This builds a triplet $< A_i, W_j, C_k >$, meaning that the application A_i has been scheduled and will use resources, from cluster C_k, required to run in working mode W_j. As the selection proceeds, the pairs including an AWM belonging to an already scheduled application are skipped. The selection ends when all active applications have obtained a cluster-mapped AWM or the availability of resources cannot satisfy the mapping of any AWM. In the case of different priority levels among the active applications, the steps just described are repeated for each set of applications having the same priority. The need to perform a multiobjective optimization has been already explained in Section 1. Considering the pair $(x, y) :=< W_j, C_k >$ as input, the function comes as a linear combination of contributions, each of which focuses on a specific objective:

$$f = aV(x, y) + bR(x, y) + cL(x, y) + dF(x, y).$$

In turn, each contribution is a function returning a goal-specific index value. The closer the value is to 1, the closer the pair (x, y) is to the optimal choice for that specific goal. The following four contributions have been taken into account:

—*Working mode value (V)*: This is a static value (properly normalized) used to express a correlation between the AWM and performance, QoS, or user quality-of-experience (QoE) level that can be reached with the amount of requested resources. In other terms, it is a way to specify how much choosing the AWM is good from the user perspective. The mapping choice y does not affect it, as the contribution completely disregards the overheads and costs to enable the corresponding schedule. Moreover, it does not consider other application or resource states.

—*Reconfiguration and migration overheads (R)*: Reconfiguring an application involves some costs, such as those related to deploying new binary objects. Migrating an application from one cluster to another is expected to have even bigger costs due to the need to migrate local data as well. Such costs should be considered and must contribute to penalize the selection of a pair (x, y) that implies migration and/or reconfiguration (index closer to 0) over other pairs (x, y) according to which no migration/reconfiguration would be required (index closer to 1).

—*Load balancing (L)*: Accessing congested resources should incur higher penalties than accessing relatively free ones. Indeed, for example, it is well known that for some classes or resources, a usage level of around 70% to 80% is the start of system saturation and corresponding decrease in performance. The goal of this contribution is thus to penalize (x, y) pairs that would increase the current contention level. As a consequence, AWMs requiring less resources and the choice of mapping an AWM on poorly used clusters usually determine higher index values.

—*Fairness (F)*: The goal of a fair scheduler is to partition available resources in a fair way among all demanding applications. Applications are the “customers” of the scheduler in charge to better serve them by giving each application the same “opportunities.” This is especially true if, for example, multiple instances of the same application are considered; thus, each instance will have the same priority and the

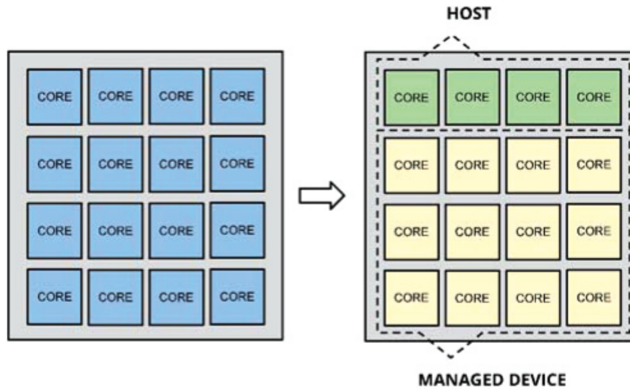

Fig. 1. Example of platform partition. A quad-core HOST node and a MANAGED DEVICE featuring three quad-core nodes.

same set of AWMs. Therefore, the contribution will return a value closer to 1 if the given pair (x, y) does not exceed a fair resource usage threshold.

As can be seen from the general expression, for each contribution function it is possible to specify a weight. As a result, the function can be tuned depending on the aimed behavior.

4.3. Enforcing Resource Management

The portability and modularity of the resource manager allows the support of several types of hardware platforms by implementing a specific *Platform Proxy*. This module is in charge of managing the communication with the platform and performs the required actions to actuate the decisions of the resource allocation policy.

On Linux multicore systems, the Platform Proxy module relies on the exploitation of the control groups framework. This allows the setting of constraints on the number of cores each task is assigned, the CPU time, and the quota of maximum memory usage.To enforce constraints on these classes of resource, three control group subsystems are needed: `cpuset`, `cpu`, and `memory`.[2]

In general, control groups can affect the execution of all tasks running on a system, whereas the proposed resource manager targets only reconfigurable applications. This introduces the need for two separate execution. The HOST domain defines resources free to be used by unmanaged tasks, and the MANAGED DEVICE domain defines a set of resources being reserved only for runtime managed applications. These domains are configured by the resource manager according a "platform description" provided via a simple text file. For example, a NUMA machine, featuring multiple multi-ore nodes, can be configured using a single node as HOST and the remaining nodes as MANAGED DEVICE (Figure 1).

A second level of resource partitioning can be defined by the platform description. This level can be used to track the physical layout of the managed device to consider aspects of data locality in the resource allocation. Considering the example of a NUMA machine, it is possible to specify for node included in the managed device, the set of CPU cores, the maximum CPU quota reserved on each core, and a corresponding memory node ID, as well as a reserved memory quota.

When started, BarbequeRTRM mounts and configures a `cgroup` file system according to the platform description using the user-space interface provided by the control

[2]Please note that the CPU bandwidth controller is available only since version 3.2 of the Linux kernel.

groups framework, which is based on a virtual file system. Information contained in the platform description file is parsed to build the initial `cgroup` hierarchy. This operation transposes the hierarchical description of the platform into the `cgroup` virtual file system by creating directories and subdirectories accordingly. Once done, the hierarchy will contain a couple of directories (i.e., `host` and `mdev`) plus the set of `mdev`'s subdirectories, starting with `node*`, at one per node/cluster. Each directory will contain all attribute files defined by the `cgroup` subsystems.

Once the initial hierarchy is ready, BarbequeRTRM migrates all tasks currently running on the system into the host partition. As a result, the resources defined by the managed device partition are free. From this point on, these resources can be allocated to demanding applications only by the resource manager, which has exclusive control over them. Therefore, whenever a reconfigurable application is launched, a new subdirectory is created and inserted into the `cgroup` hierarchy. The name of the subdirectory references the application through a merge of the process ID number (PID) and a short form of the application name. The `cgroup` attributes are then filled with the information related to the assigned resources according to the decisions of the resource allocation policy.

As a result, the information filling the `cgroup` hierarchy will drive the actions taken by the Linux scheduler and the memory manager, constraining the tasks specified (runtime reconfigurable applications) to use only the set of the resources assigned by the BarbequeRTRM framework.

5. EXPERIMENTAL RESULTS

An experimental evaluation of the proposed framework has been done to assess its capability to efficiently manage resources considering different workload congestion levels and workload mixes.

A first set of experiments evaluates the impact of using control groups to optimize the execution of an increasing workload, which is based on multiple instances of a benchmark application. A second set of experiments evaluates our framework behaviors on managing mixed workload scenarios, which are based on different combinations of two benchmark applications, running on the same NUMA machine.

5.1. Experimental Setup

The experiments have been done using two applications from the PARSEC v2.1 benchmark suite [Bienia et al. 2008]. These two applications have been selected because they exhibit two different and complementary parallelization models.

The first one, Bodytrack, is a good example of data parallelism where the input dataset is decomposed and processed in parallel by a configurable number of threads. The second benchmark, Ferret, is instead an example of task parallelism where the input stream is processed by a four-stage pipeline, with each stage exploiting data parallelism as well.

These applications have been modified to integrate them with the BarbequeRTRM framework. The integration effort has been mainly devoted to making these applications runtime tunable by allowing change to the number of threads used for data processing while the application is running. Although such an effort could be avoided if applications are designed and developed to be runtime tunable, the additional code required to integrate an application with the framework is limited to a few code lines. Both the framework and the modified application used for these tests are freely available online.[3]

[3]Starting from the project Web site: http://bosp.dei.polimi.it.

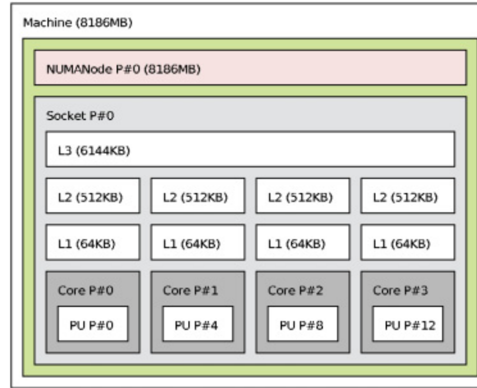

Fig. 2. The memory hierarchy of the target machine used for experiments.

Table II. Target Platforms Ppartitioning among HOST and MANAGED DEVICE

	AMD–NUMA Machine	
	CPUs IDs	Memory Nodes
HOST partition	0,4,8,12	0
MDEV partition	1–3,5–7,9–11,13–15	1–3

Table III. Resource Clusterization for the MANAGED DEVICE

	AMD–NUMA Machine			
	CPU IDs	Time Quota	Memory Nodes	Memory (MB)
NODE1	1,5,9,13	100	3	6,000
NODE2	2,6,10,14	100	2	6,000
NODE3	3,7,11,15	100	1	6,000

Regarding the hardware platforms, experiments have been carried out using a four-node NUMA machine, with each node (Figure 2) consisting of a quad-core AMD Opteron processor 8378, running up to 2.80GHz, for a total count of 16 processing units (PUs). The available PUs have been organized into a *host partition*, used to run the framework and all other generic system services and applications, and a *managed device partition*, which defines instead the set of resources used to run applications managed by our framework. The partitioning defined for the two target platforms is reported in Table II, whereas in Table III the clusterization of the managed device into three nodes is specified. It is worth noticing that the managed device has been clustered by matching the memory hierarchy, a solution that addresses an improved exploitation of caches.

As a final remark,all experiments have been conducted by using the Linux CPUFreq framework, specifically its on-demand policy, to select the best CPU frequency based on runtime effective workload. Moreover, in experiments not using the proposed framework, the standard Linux CFS scheduler was completely free to allocate applications and their threads using all 12 PUs available in the 3 CPUs of the managed device.

5.2. Workload Burst Analysis

The burst analysis (BA) targets the evaluation of the framework benefits in terms of time and energy required to complete a batch workload. The scalability of these benefits is also investigated by considering an increasing number of concurrently running applications and their parallelization level. To this purpose, the Bodytrack workload has been selected because it exhibits a quasilinear scalability with the number of threads,

Table IV. Performance Metrics Collected during Tests

Goal	Description
CTIME	Time [s]: Workload completion time [s]
POWER	Power [W]: System power consumption [W]
TASK-CLOCK	Ticks: Task clock ticks
CTX	Context switches: Total number of context switches
MIG	Migrations: Total number of CPU migrations
PF	Page faults: Total number of page faults
CYCLES	Cycles: Total number of CPU cycles
FES	Front-end stalls: Total number of front-end stalled cycles
FEI	Front-end idles: Total number of front-end idle cycles
BES	Back-end stalls: Total number of back-end stalled cycles
BEI	Back-end idles: Total number of back-end idle cycles
INS	Instructions: Total number of executed instructions
SCPI	SPC: Effective stalled cycles per instruction
B	Branches: Total number of branches
B-RATE	Branches rate: Effective rate of branch instructions
B-MISS	Branch miss: Total number of missed branches
B-MISS-RATE	Branch miss quota: Effective percentage of missed branches
GHZ	GHz: Effective processor speed
CPU-USED	CPUs utilized: CPU utilization
IPC	IPC: Effective instructions per cycle

thus allowing a better evaluation of parallelization-level impact with respect to the amount of assigned PUs.

A test configuration is defined by the number of concurrently running instances of Bodytrack and their maximum parallelization level. For each configuration, the original unmodified version of the benchmark has been compared to the one integrated with the proposed framework. The original version is executed with the specified number of threads and freely scheduled by the Linux kernel on the CPUs of the managed device partition. To the contrary, the runtime managed version is capable of running with the same maximum number of threads or with a reduced number, depending on the amount of resources (i.e., CPU time) assigned by the BarbequeRTRM framework, and it is scheduled by the Linux kernel only on the subset of CPUs assigned within the managed device partition.

The standard Linux *perf* framework and a user-space tool have been used to collect, for each test configuration, a rich set of performance counters ranging from architectural metrics to OS events. A detailed list of all considered metrics is represented in Table IV; all metrics but the IPC should be considered of class "the lower the better"—that is, a lower value corresponds to a better behavior. The number of instructions per cycle (IPC) is the only exception; indeed, a higher instruction execution rate corresponds to a better exploitation of the computational resources of the CPUs. Moreover, the average power consumption during the execution of a configuration test has been collected via the IPMI interface available on the test machine. It is worth noticing that the collected values correspond to the overall system consumption in Watts.[4]

A detailed report on all collected metrics is reported in Tables V and VI for two representative scenarios. Table V refers to applications configured to run with only one thread, whereas Table VI reports results for applications running with eight threads. Metrics related to the original unmodified application are reported under the U columns, whereas the M columns are for BarbequeRTRM managed applications. For each metric, based on 30 repetitions of the experiment, the mean and 95% and 99%

[4]This comprises not only CPUs but also (and mainly) disks and cooling fan usage.

Table V. Performance Metrics (1 Instance, 1 Thread)

	Mean		95% CI		99% CI	
Metric	U	M	U	M	U	M
ipc	1.080	1.235	0.000	0.002	0.000	0.003
bei	53.942	45.386	0.031	0.070	0.041	0.092
ins [G]	126.869	27.450	0.039	0.023	0.052	0.030
ctime	50.883	9.881	0.028	0.005	0.037	0.007
power	287.233	286.533	0.457	0.636	0.602	0.837
ghz	2.292	2.217	0.001	0.002	0.001	0.003
b-miss [M]	168.028	26.191	2.026	0.148	2.666	0.195
b-rate	250.667	309.050	0.209	0.460	0.275	0.605
ctx [K]	6.468	2.897	0.007	0.004	0.009	0.005
mig	15.433	92.233	3.055	1.123	4.021	1.478
fei	1.405	1.546	0.026	0.017	0.034	0.022
task-clock [K]	51.226	10.030	0.029	0.004	0.038	0.006
cpu-used	1.007	1.015	0.000	0.000	0.000	0.000
b [G]	12.841	3.100	0.007	0.004	0.010	0.006
pf [K]	30.708	52.186	0.006	0.001	0.008	0.001
fes [G]	1.650	0.344	0.030	0.004	0.040	0.005
scpi	0.500	0.370	0.000	0.000	0.000	0.000
bes [G]	63.324	10.091	0.044	0.009	0.059	0.012
cycles [G]	117.392	22.234	0.054	0.020	0.071	0.026
b-miss-rate	1.309	0.845	0.016	0.005	0.021	0.006

Table VI. Performance Metrics (12 Instances, 8 Threads)

	Mean		95% CI		99% CI	
Metric	U	M	U	M	U	M
ipc	1.070	1.325	0.000	0.002	0.000	0.002
bei	54.341	39.930	0.010	0.022	0.013	0.029
ins [P]	1.524	0.225	0.000	0.000	0.000	0.000
ctime	49.828	7.767	0.026	0.060	0.035	0.080
power	392.000	363.767	0.000	0.801	0.000	1.055
ghz	2.393	2.380	0.000	0.001	0.000	0.001
b-miss [G]	1.920	0.174	0.004	0.001	0.005	0.001
b-rate	259.622	381.845	0.059	0.382	0.078	0.503
ctx [K]	469.009	277.739	0.366	0.522	0.482	0.687
mig [K]	30.818	0.412	0.293	0.015	0.386	0.019
fei	1.412	1.545	0.010	0.005	0.013	0.006
task-clock [K]	594.929	71.434	0.103	0.039	0.136	0.051
cpu-used	11.940	9.202	0.005	0.072	0.007	0.095
b [G]	154.457	27.277	0.024	0.033	0.031	0.043
pf [K]	363.717	621.619	0.024	0.052	0.032	0.069
fes [G]	20.099	2.629	0.142	0.007	0.187	0.010
scpi	0.510	0.300	0.000	0.000	0.000	0.000
bes [G]	773.565	67.898	0.160	0.061	0.210	0.080
cycles [P]	1.424	0.170	0.000	0.000	0.000	0.000
b-miss-rate	1.243	0.637	0.003	0.003	0.004	0.003

confidence intervals are reported, the last of which clearly state the significance of the computed mean values.

For better readability, Figure 3 reports a graphical comparison between the original version (unmanaged) and the one integrated with the framework (BBQ managed) in the two representative scenarios and the most interesting metrics.

The first column represents the time required to complete the workload for the given number of concurrently started instances of Bodytrack. In the case of only one thread,

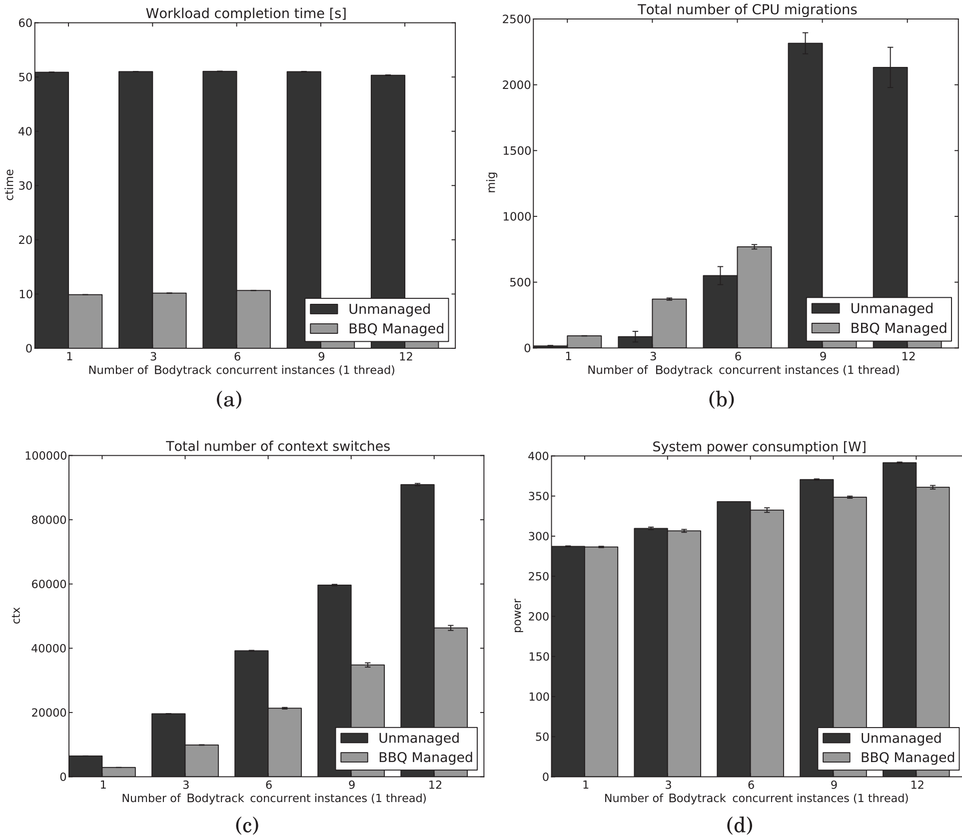

Fig. 3. Workload burst performance scalability using an increasing number of single-threaded instances of Bodytrack. (a) Completion time. (b) Effective processor speed. (c) Context switches. (d) System power consumption.

the unmanaged version is five times slower than the integrated one. This is explained by looking (refer to Table V) at the number of executed instructions (ins) (i.e., $126 \cdot 10^9$ vs, $27 \cdot 10^9$), missed branches (i.e., $168 \cdot 10^6$ vs. $26 \cdot 10^6$), and context switches (i.e., $6 \cdot 10^3$ vs. $3 \cdot 10^3$). These results reflect inefficient code execution, probably due to an inefficient exploitation of the memory resource, as confirmed by the increased average number of stalls on both the front- and back-end stages of the pipeline. The BarbequeRTRM managed applications get an amount of resources assigned, as well as a subset of the available processors where their threads can be scheduled for execution. This results in a more efficient code execution, which is also certified by the improved IPC index (i.e., 1.235 instead of only 1.080) of the original unmanaged application.

As expected, in the scenarios with just 1 thread, the completion time is not changing with the increasing number of concurrent instances.The experiment considers up to 12 concurrent instances that can be independently scheduled on the 12 CPUs of the managed device. However, increasing the number of instances, it has been noticed, as expected, that an increased number of migrations and context switches for the unmanaged application occur. This is due to the higher contention among all applications, which results in an increased Linux scheduler activity. The same effects are mitigated in the case of the BarbequeRTRM managed application, as the proposed framework reduces the degrees of freedom of the Linux scheduler by assigning, via control groups,

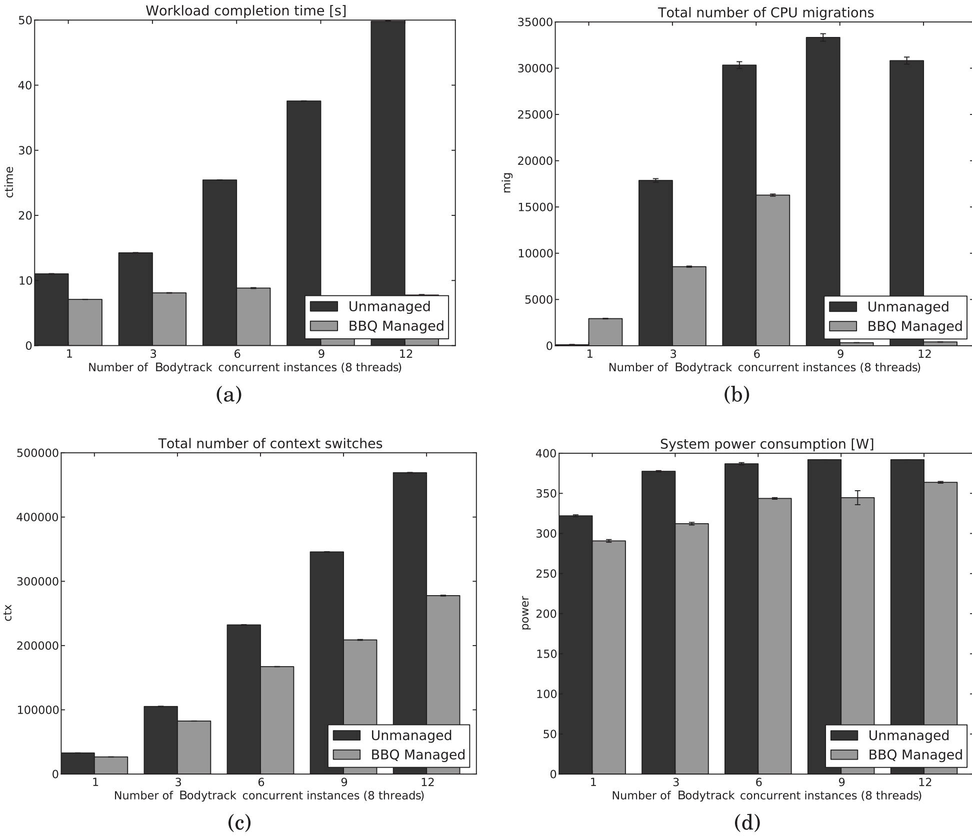

Fig. 4. Workload burst performances scalability using an increasing number of Bodytrack with eight-thread parallelization level. (a) Completion time. (b) Effective processor speed. (c) Context switches. (d) System power consumption.

specific resources (i.e., CPUs) to each application. The benefits of this constraining of the scheduler are even more evident in scenarios with 9 and 12 concurrent instances. Indeed, in these cases, due to the high number of instances, the fair resources allocation policy provided with the framework assigns a single CPU to each instance, thus drastically reducing the number of migrations. A higher efficiency in code execution reduces system power consumption, which improves by 0.2% and up to 8.6%, ranging from 1 to 12 instances.

Similar results can be found in the second row of Figure 4, where scenarios with eight threads per instance are reported. In these scenarios, the workload mix produces high system congestion. The Linux CFS scheduler does its best to bewith all threads of the concurrently running applications. However, this policy has a significant impact on the number of migrations and context switches, with a corresponding degradation of all other architectural metrics, as reported in Table VI. To the contrary, the goal of the BarbequeRTRM framework is either to grant a reasonable minimum amount of resources to an application or to postpone its execution. In scenarios with high congestion, the net effect is a partial serialization of demanding applications. As the figure shows, this results in a smaller completion time and a higher power efficiency, which always improves between 7% and 11%. It is worth noticing that the combined effect on reduced workload completion time and better power efficiency has an even more

Table VII. Performance Speedups by Scenario

Scenario	ctime [%Δ]	power [%Δ]	energy [%Δ]
1 Thread, 1 Instance	80	0.2	16
1 Thread, 12 Instances	84	7.8	655
8 Threads, 1 Instance	35	9.7	339
8 Thread, 12 Instances	84	7.2	604

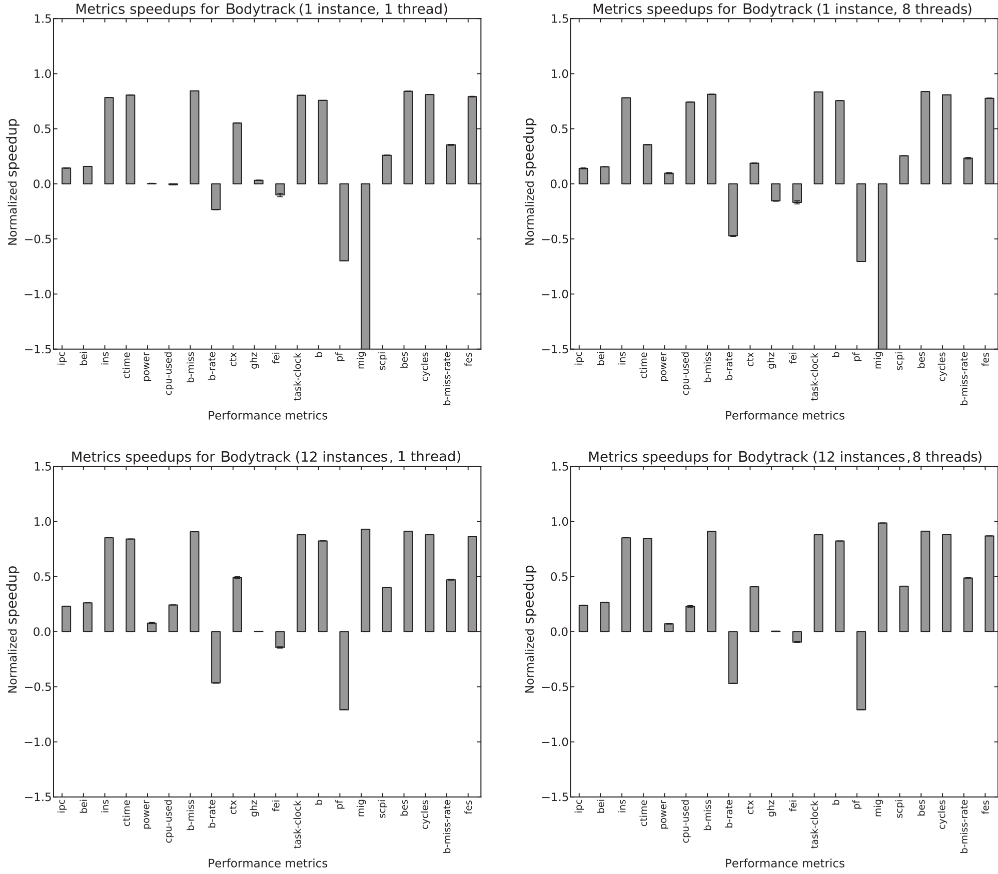

Fig. 5. Benefits and loss on the considered performance metric. Positive bars mean speedup, whereas negative ones mean degradation of the runtime managed Bodytrack compared to the original unmanaged version.

interesting effect on the overall energy consumption, which improves up to six times in more congested scenarios, as reported in Table VII.

An overall graphical representation of the speedups for each considered metric is reported in Figure 5 for the four scenarios of the previous table. In this figure, a positive bar corresponds to an improvement, whereas a negative bar represent a deficiency of the integrated application with respect to the original one. These graphs indicate that the BBQ managed version of the application performs much better in all congestion levels corresponding to the different scenarios.

Among the negative metrics, the number of migrations (mig) can be noticed; however, it is worth noticing that this degradation is only a measure for lower-congestion scenarios (i.e., first two graphs), where the number of migrations, even being different,

Table VIII. Workload Mix Scenarios and Runtime Configurations

	Threads Count				Workload Mix			
Workload	AWM0	AWM1	AWM2	AWM3	L1	M1	H1	H2
Bodytrack	2	2	3	5	2	3	6	4
Ferret	1	1	3	4	2	3	4	6
CPU quota	50	100	200	400				

is a relatively small number and with the same order of magnitude in both cases (refer to Table V). Instead, when the congestion level increases (i.e., the last two graphs), the difference in number of migrations, operated by the Linux CFS scheduler on the original unmanaged version, increases by up to two orders of magnitude difference with respect to the runtime managed version. At higher levels of congestion, the benefits of the constrained resources assignment operated by the BarbequeRTRM framework is better appreciated.The page faults (pf) metric is also always degraded for the integrated version, as well as the branch rate (b-rate). This is due to the way the original application has been integrated with the BarbequeRTRM framework. Indeed, the body of the main processing loop has been moved into a call-back method provided by the runtime management library. This code reorganization inhibits some compiler optimization, namely loop unrolling could not be applied, which is the main reason for those degradations. A different integration is possible by moving a loop portion within the required call-back method; however, this has been left for future investigations.

All other metrics benefit from integration with the BarbequeRTRM framework. It is worth noticing an overall better exploitation of the computational resources thanks to the resources assignment operated by the BarbequeRTRM framework. The effective CPU utilization (cpu-used) is always improved thanks to a significant reduction of front-end and back-end stalls (fes and bes), which corresponds to a reduction in the number of stalled cycles per executed instruction (scpi). These last three architectural metrics indicate that the instruction stream is well optimized, and such results have been achieved by offsetting compile-time optimization discussed so far with a better assignment of resources at runtime.

5.3. Mixed-Workload Analysis

These experiments focus on mixed workloads, where different applications run concurrently, and targets an evaluation of the the BarbequeRTRM framework benefits generalization to different classes of applications.

Different runtime scenarios have been considered, each one defined by a mix of Bodytrack and Ferret workloads, which are concurrently in execution and configured according to data reported in Table VIII. Each workload has been configured with four AWMs that correspond to a different amount of threads and quota of CPU time. The matching between the optimal number of threads given a certain time quota has been identified at design time by exploiting a DSE tool to analyze all possible configurations. Thus, for example, the best completion time for a single instance of Bodytrack (respectively Ferret) running alone on four PUs (i.e., 400% CPU quota) has been identified, which corresponds to a five-thread parallelization level (respectively four threads for Ferret).

The workload mix configurations are defined in terms of Bodytrack and Ferret instances, specifically the four different scenarios reported in Table VIII have been considered. These scenarios have been identified such that all applications, when managed by the BarbequeRTRM framework, cannot be scheduled in the maximum working mode. For example, even in the lightest scenario L1, to run all applications with maximum resources, a total of 16 PUs would be necessary (400% CPU quota by 2 + 2 instances), whereas only 12 PUs are available in the managed device (refer to Table II).

Table IX. Performance Speedups by Scenario

Scenario	Workload Mix	ctime [%Δ]	power [%Δ]	energy [%Δ]
L1	2 Bodytrack, 2 Ferret	21.7	15.6	34.0
M1	3 Bodytrack, 3 Ferret	25.3	7.4	30.9
H1	6 Bodytrack, 4 Ferret	22.5	6.7	27.7
H2	4 Bodytrack, 6 Ferret	26.0	–0.3	25.8

Thus, when the scenario is run using the BarbequeRTRM framework, because of the fairness optimization metrics (cfr. Section 4), the workloads will be co-scheduled just on AWM2, where a total of 8 PUs are required and a whole CPU is not used, thus reducing power consumption.

The same set of metrics used for the burst analysis have been collected during the execution of these scenarios. Each scenario has been executed multiple times to build statistics on its completion time and power consumption. The performances of the original applications and those integrated with our framework have been compared. In the first case, the Linux scheduler only has been in charge to allocate CPU time and threads on the 12 PUs of the managed device, whereas in the second case, the number of threads, the PUs to be used, and the allowed CPU quota have been defined by the BarbequeRTRM framework.

The results of this analysis are summarized in Table IX, where a constant improvement on the completion time,which is never lower than 20%, can be noticed. The resources allocation proposed by the framework corresponds not only to better performances but also to reduced power consumptions. This results in an overall improved energy efficiency, which is never lower then 25%, given the considered scenarios.

6. CONCLUSIONS

This work introduces a new user-space solution for runtime resource management that extends the advanced and efficient resources control capability offered by modern Linux kernels with suitable resources partitioning policies. In this article, we focused on evaluating the effectiveness to exploit the control group Linux framework to mandatory assignment of a set of computational resources to concurrently running applications.

Experiments have been based on workloads from the PARSEC v2.1 benchmark suite, which has been made runtime tunable and integrated with our framework. Initial results show the effectiveness of the proposed solution considering a wide range of performance metrics. The proposed solution provides an up to 80% improvement in execution time and six times the energy efficiency for the considered workload in many and different system congestion scenarios.

A further analysis, which considered mixed workloads of two applications from the same benchmark suite, shows the extension of benefits produced by our solution to the management of different classes of applications. These benefits target not only performances, which are boosted by 20% at least, but also energy efficiency, which is improved at least by 25% in the evaluated scenarios.

Overall, this work fosters a more runtime-aggressive exploitation of Linux control groups not only as a container framework to grant isolated execution contexts but also to improve application performances and system-wide energy consumption. Future works target the integration of other representative workloads, the evaluation of different target architectures, and the assessment of the proposed solution on more industrial usage scenarios.

REFERENCES

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International*

Conference on Parallel Architectures and Compilation Techniques (PACT'08). ACM, New York, NY, 72–81. DOI:http://dx.doi.org/10.1145/1454115.1454128

Enrico Bini, Giorgio Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Karl-Erik Arzen, Vanessa Romero, and Claudio Scordino. 2011. Resource management on multicore systems: The ACTORS approach. *IEEE Micro* 31, 3, 72–81.

Sergey Blagodurov and Alexandra Fedorova. 2011. User-level scheduling on NUMA multicore systems under Linux. Retrieved December 18, 2014, from http://kernel.org/doc/ols/2011/ols2011-clavis.pdf.

Hans Domjan and Thomas R. Gross. 2001. Managing resource reservations and admission control for adaptive applications. In *Proceedings of the International Conference on Parallel Processing*. 499–506.

Xing Fu and Xiaorui Wang. 2011. Utilization-controlled task consolidation for power optimization in multi-core real-time systems. In *Proceedings of the IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Vol. 1. 73–82. DOI:http://dx.doi.org/10.1109/RTCSA.2011.65

Michael R. Head, Andrzej Kochut, Charles Schulz, and Hidayatullah Shaikh. 2010. Virtual hypervisor: Enabling fair and economical resource partitioning in cloud environments. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS)*. 104–111. DOI:http://dx.doi.org/10.1109/NOMS.2010.5488444

Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. 2010. *SEEC: A Framework for Self-Aware Computing*. Technical Report MIT-CSAIL-TR-2011-046. Massachusetts Institute of Technology, Cambridge, MA.

Steven Hofmeyr, Costin Iancu, and Filip Blagojević. 2010. Load balancing on speed. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 147–158.

Gaston Keller and Hanan Lutfiyya. 2010. Replication and migration as resource management mechanisms for virtualized environments. In *Proceedings of the 6th International Conference on Autonomic and Autonomous Systems (ICAS)*. 137–143. DOI:http://dx.doi.org/10.1109/ICAS.2010.27

Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. Article No. 53.

Linux Containers. 2010. Linux Containers Home Page. Retrieved December 18, 2014, from http://lxc.sourceforge.net.

Linux VServer. 2010. Linux-VServer Home Page. Retrieved December 18, 2014, from http://linux-vserver.org/Welcome_to_Linux-VServer.org.

M. Louvel, J. Tous, J.-P. Babau, and A. Plantec. 2011. Ensuring QoS of multimedia applications in heterogeneous home networks: The CPU use case. In *Proceedings of the 9th International Conference on Embedded and Ubiquitous Computing (EUC)*. 19–26. DOI:http://dx.doi.org/10.1109/EUC.2011.34

Giovanni Mariani, Prabhat Avasare, Geert Vanmeerbeeck, Cliaiztal Ykman-Couvreur, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 2010. An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE)*. 196–201.

Paul Menage. 2006. CGroups. Retrieved December 18, 2014, from http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

MontaVista. 2010. *Beyond Virtualization: The MontaVista Approach to Multi-Core SoC and Resource Allocation and Control*. Retrieved December 18, 2014, http://mvista.com/download/Whitepaper-Beyond-Virtualization.pdf.

OpenVZ. 2010. OpenVZ Home Page. Retrieved December 18, 2014, from http://wiki.openvz.org/Main_Page.

Patrick Bellasi, Giuseppe Massari, and Williams Fornaciari. 2012. A RTRM proposal for multi/many-core platforms and reconfigurable applications. In *Proceedings of the 7th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. 1–8.

Adit Ranadive, Ada Gavrilovska, and Karsten Schwan. 2011. ResourceExchange: Latency-aware scheduling in virtualized environments with high performance fabrics. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 45–53. DOI:http://dx.doi.org/10.1109/CLUSTER.2011.14

Tyler Sondag and Hridesh Rajan. 2011. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 11–20.

Peng Yang and Francky Catthoor. 2003. Pareto-optimization-based run-time task scheduling for embedded systems. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS)*. ACM, New York, NY, 120–125. DOI:http://dx.doi.org/10.1145/944645.944680

Chantal Ykman-Couvreur, Erik Brockmeyer, Vincent Nollet, Theodore Marescaux, Francky Catthoor, and Henk Corporaal. 2005. Design-time application exploration for MP-SoC customized run-time management. In *Proceedings of the International Symposium on System-on-Chip*. 66–69. DOI:http://dx.doi.org/10.1109/ISSOC.2005.1595646

Received June 2013; revised May 2014; accepted July 2014