Exploiting Linux Control Groups for Effective Run-time Resource Management

Patrick Bellasi, Giuseppe Massari, William Fornaciari
DEI - Politecnico di Milano
Via Ponzio, 34/5
Milano, Italy
Email: {bellasi,massari,fornacia}@elet.polimi.it

***Abstract*—The extremely high technology process reached by the silicon manufacturing (under the 32nm) has led to production of computational platforms and SoC, featuring a considerable amount of resources. Whilst 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, as well as target of congested workload usage scenarios. Effective usage of the resources should take into account both the *application requirements* and *resources availability*, with an arbiter, namely a Resource Manager, which is in charge to solve the resource contention among demanding applications. Current Operating Systems are always less effective in playing this role, so that there is the opportunity to integrate them with an ad-hoc resource manager. Such integration should rely on suitable interfaces and mechanisms, providing extensive capabilities of controlling tasks and system resources. This paper, focusing on multi-core Linux systems, shows a portable approach to enforce run-time resource management decisions, based on the standard Control Groups framework. Early tests, performed on a 16 core NUMA machine, have reported some promising results both in terms of performance and power saving.**

I. Introduction

The progress in the silicon technology process has made possible the design and the manufacture of computing platforms, featuring levels of parallelism ranging from a few to hundreds 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. Non-functional constraints, as area, power and thermal management, had always been typical design issues of such systems, but nowadays embedded systems are loosing this sort of exclusiveness. Energy efficiency, along with thermal management, system reliability and fault-tolerance capabilities are more than ever requirements of distributed computing systems too, as for instance server farms. Disregarding this requirements would trivially lead to a dramatic increase of maintenance costs, frequency of failures and loss of performance. As a consequence, it is mandatory to implement effective resource management policies, to properly exploit modern computing platforms, taking into account the constraints above. From the system software side, this means to integrate general purpose Operating Systems (OS) with suitable resource management frameworks. However, an effective integration requires the OS to expose interfaces and mechanisms, through which the resource manager could enforce its decisions. This paper focuses on *mechanisms* supporting run-time resource management on multicore Linux systems.

The rest of the paper is structured as follows. The next section briefly resumes related works, while Sec. III provides a short background on Control Groups, along with the motivations of our choice. Sec.IV summarizes the basics of the proposed run-time management framework, with a detailed description about how we exploit Control Groups. Sec.V reports some results obtained by running a benchmark workload on a NUMA machine. Finally, conclusions are drawn in Sec.VI.

II. Related works

A class of resource management approaches focuses mainly on the problem of task scheduling, with the goal to optimize the performance/power trade-off. Several proposals, targeting Linux systems, have been aimed at extending the scheduler of the operating system, by adding new scheduling classes, like in [2] [4] and [7]. The portability of these proposals is obviously tied to the adoption of a customized kernel.

A more portable class of solutions, includes all the approaches implemented in user-space. 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 [3] [11] [6] and [5]. These solutions exploit the `sched_setaffinity` syscall to set the CPU affinity mask.

Another class is that of resource manager 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 an *hypervisor* acting as a global system manager. Both open-source and commercial solutions are currently available, like OpenVZ, VServer, Linux Containers and [10]. Some of them are built on top of Linux Control Groups [8] to setup the execution environment. Anyway, resource partitioning is "statically" defined ahead of system deployment.

As a novel contribution, we propose a more dynamic usage of Control Groups, by managing resources of a Linux system through a portable and modular Run-Time Resource Manager (RTRM) running in user-space.

TABLE I: Control Groups sybsystem.

Sybsystem	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

III. Linux Control Groups

The Control Groups framework [8] is 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, e.g. CPUs, memory quota and I/O bandwidth. In general, a group of tasks could be configured to use a specific set of resources either in a *shared* or *exclusive* way. The former option allows overlapping among resource subsets, while the latter could be used to configures a set of *isolated execution environments*, which could be conveniently used to setup a light-weight vitualization solution. Resource partitioning criteria could be task-based 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 multi-user remote servers and distributed systems.

In this paper we introduce a different usage of the framework, going beyond the common view of an administration tool. Some of the key aspects, that led us to choose this framework as a mechanism to enforce resource management decisions, are resumed thereafter.

- User-space interface to control the system resources. This is a strong point in terms of *portability* and *invasiveness* of the resource management approach, since 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 to optionally focus on the control of just a subset of resources classes. Tab. I lists the currently available subsystems with the resource control capabilities on tasks of a cgroup.
- Wide set of control parameters, added by subsystems. This makes Control Group the most *complete* interface towards the system resources, which gives great potential to a run-time 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 we exploit this framework, a brief description of our approach to run-time resource management is presented in the following section.

IV. Run-time resource management

The enforcement of resource constraints requires a proper definition of how applications and system resources are characterized, from the run-time resource manager standpoint.

Regarding *applications*, it is required that they exhibit capabilities of "run-time reconfiguration". To support this feature, applications must a) be integrated into the execution model defined by a library provided by the resource manager (RTLib), and b) define a finite set of possible run-time configurations. Our proposal exploits two levels of reconfiguration, and corresponding configurations. The highest abstraction level, identifies *Application Working Modes (AWMs)*. An AWM defines a set of resource requirements needed to get a certain QoS level. The lowest abstraction level instead identifies the *Operating Points (OPs)*. An OP is a set of application specific-parameters affecting the quality of the application, in the boundaries of the QoS that can be obtained given the current AWM. The set of configurations of an application is defined with the support of offline profiling and Design Space Exploration (DSE) tools. The plausibility of this assumption comes from prior works [9], [12], [13] wherein it has been shown how much DSE could be helpful in supporting run-time resource management.

Regarding *system resources*, looking at the current evolution of modern platforms and SoC, very common design solutions configure architectures featuring homogeneous and hierarchical layouts. Accordingly, our resource manager handles the concept of *cluster*, to reference a group of processing elements, usually sharing a local memory, also referenced with the term "node" in the domain of NUMA machines.

Given this scenario, the objective of the 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 run-time about its AWM, hence the corresponding assigned amount of resources, so that for example it could 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 Operating Point (OP). We will not provide further details about this point[1], as well as for scheduling and resource allocation policies, which are out of the scope of the paper. Anyway, it is worth to say that the modular design of the BarbequeRTRM framework [1] allows to plug-in multiple scheduling and resource allocation policies.

Enforcing resource management

The portability and modularity of the resource manager allows to support several types of hardware platforms, by implementing a specific *Platform Proxy*. This module is in

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

TABLE II: Platform description

	CPUs IDs	Memory Nodes
HOST	0,4,8,12	0
MDEV	1-3,5-7,9-11,13-15	1-3

a) Target devices

	CPUs IDs	Time Quota	Mem Nodes	Memory (MB)
NODE	1,5,9,13	100	3	6000
NODE	2,6,10,14	100	2	6000
NODE	3,7,11,15	100	1	6000

b) Clusterization of managed resources

charge of managing the communication with the platform, and perform 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 to set 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 Groups subsystems are needed: `cpuset`, `cpu` and `memory`[2].

In general, Control Groups can affect the execution of all the tasks running on a system, whilst our resource manager targets on reconfigurable applications. This introduces the need of two separate execution domains. A domain defines resources free to be used by *unmanaged* tasks, another one instead defines a set of resources reserved just for run-time managed applications. These domains are called respectively *host* and *managed device* and are confiugured by the resource manager by using a "platform description", which is a simple text file. For example a 16 core NUMA machine, featuring four nodes, could be configured using a single node as host, and the remaining three nodes as managed device.

A second level of resource partitioning could be defined by the platform description. This level could be used to track the physical layout of the managed device, in order to consider aspects of data locality in the resource allocation. For example, it is possible to specify for each NUMA 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. Tab. II summarizes a possible platform description for the aforementioned machine, this is also the configuration used for the experiments presented in the following section.

When started, our resource manager mounts a `cgroup` file-system and configure it, according to a specified platform description, using the user-space interface provided by the Control Groups framework, which is based on a virtual file-system. Information contained in the platform description file are parsed, in order to build the initial `cgroup` hierarchy. Basically, this operation transpose the hierarchical description of the platform into the `cgroup` virtual file-system, by creating directories and sub-directories 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*`, one per node/cluster. Each directory will contain all the attribute files defined by the `cgroup` subsystems.

Once the initial hierarchy is ready, the resource manager migrates all the 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 onwards, these resources can be allocated to demanding applications only by the resource manager, which has exclusive control on them. Therefore, whenever a reconfigurable application is launched, a new sub-directory is created and inserted into the `cgroup` hierarchy. The name of the sub-directory references the application through a merge of the 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 scheduling and 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 (reconfigurable applications) to use only the set of the resources assigned by the resource manager.

V. Experimental Results

An experimental evaluation of the proposed framework has been done in order to assess its capability to efficiently manage resources on highly congested workload scenarios. Specifically we targeted the evaluation of the framework capabilities considering a number of concurrently running applications ranging from 1 to 12, as well as different parallelization levels. The experiments have been done using the Bodytrack application from the PARSEC v2.1 benchmarks suite. This application has been modified to integrate it with the BarbequeRTRM framework. The integration effort has been mainly devoted to make this application run-time runnable by simply switching the number of threads[3] used for data processing while the application is running. While such an effort could be avoided if applications are designed and developed to be run-time tunable, the additional code required to actually integrate the application with the framework has been limited to a really few lines. Both the framework and the modified application used for this tests are freely available on-line.

The experiments have been carried out on a four nodes NUMA machine, each node consisting of a Quad-Core AMD Opteron Processor 8378 running up to 2.8GHz[4], for a total count of 16 Processing Elements (PE). These PEs have been organized into a *host partition* with 4 CPUs, used to run

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

[3]Threads number could be changed after each frame has been processed.

[4]Experiments have been conducted by exploiting CPUFreq and its on-demand policy to select the best CPU frequency based on run-time workload

TABLE III: 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 - CPUs utilization
IPC	IPC - Effective Instructions-per-Cycles

the framework and all the other generic system services and applications. The remaining 12 CPUs have been placed into a *managed device partition*, used to run only applications managed by our framework, and further divided into 3 clusters with 4 CPUs each one.

A test configuration is defined by the number of concurrently running instances and the parallelization level. For each configuration we compared the original unmodified version of Bodytrack with the one integrated with our 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 run-time managed version is capable to run with the same maximum number of thread or with a reduced one, depending on the amount of resources (i.e. CPU time) assigned by the BarbequeRTRM framework, and scheduled by the Linux kernel just on the subset of CPUs assigned within the managed device partition.

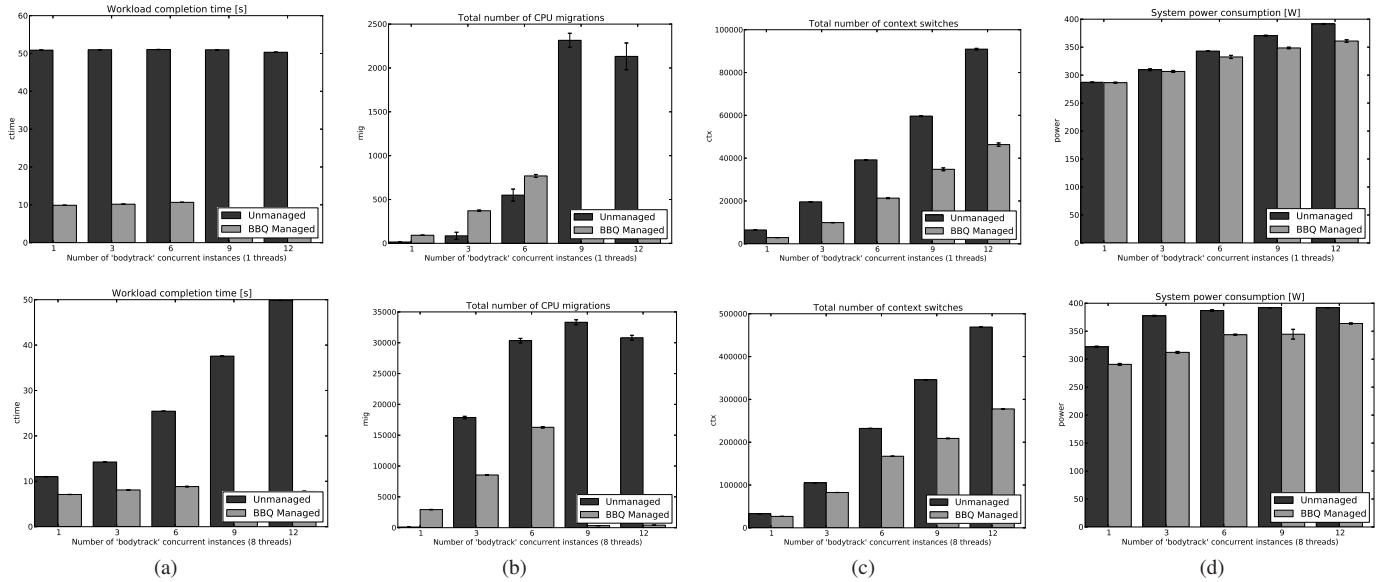
The standard Linux *perf* framework and user-space tool has been used to collect, for each test configuration, a rich set of performance counters, ranging from architectural metrics to operating system events. A detailed list of all the considered metrics is represented in Tab. III; all the metrics but the IPC should be considered of class "the lower the better", i.e. a lower value correspond to a better behavior. The number of "Instructions per Cycles" (IPC) is the only exception, indeed an 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 to notice that the collected values correspond to the overall system consumption in [W][5].

A detailed report on all the collected metrics is reported in Tab. IV and Tab. V for two representative scenarios. The first table refers to applications configured to run with just one thread while on the second table we have results for applications running with 8 threads. Metrics related to the original unmodified application are reported under columns U, while clumsy M is for BarbequeRTRM managed applications. For each metric we report, based on 30 repetition of the experiment, the mean as well as 95% and 99% confidence intervals. These last clearly states the significance of the computed mean values.

For better readability, Fig. 1 reports a graphical comparison between the original version (unmanaged) and the one integrated with our 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 case of just one thread the unmanaged version is 5 time slower than the integrated one. This could be explained by looking (ref Tab. IV) 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 measures are index of an 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. To the BarbequeRTRM managed applications not only is assigned a workload congruent amount of resources but its threads are also pinned to predefined processors. This results into an improved code execution efficiency, which is also certified by the improved IPC index, i.e. 1.235 instead of just 1.080 of the original unmanaged application. As expected, the completion time in the scenarios with just 1 thread, is not changing with the increasing number of concurrent instances. Indeed, we considered just up to 12 concurrent instances which could be quite independently scheduled on the 12 CPUs of the managed device. However, increasing the number of instances, as expected, we notice an increased number of migrations and context switches for the unmanaged application. This is due to the higher contention among all the applications which results into an increased Linux scheduler activity. The same effects are mitigated in the case of BarbequeRTRM managed application, since our framework reduces the degrees of freedom of the Linux scheduled by assigning via control groups specific resources (i.e. CPUs) to each application. The benefits of this constraining of the scheduler are even more evident on scenarios with 9 and 12 concurrent instances. Indeed, in these cases, due to the high number of instances the fair resources allocation policy of our framework assigns a single CPU to each instance, thus drastically reducing the number of migrations. An higher efficiency in code execution turns into a reduce system power consumption, which improves of 0.2% up to 8.6% ranging from 1 to 12 instances.

Similar results could be found on the second row of Fig. 1, where scenarios with 8 threads per instance are reported. In these scenarios the workload mix produces high system congestion. The best-effort Linux CFS scheduler does its best to be with all the threads of the concurrently running

[5]Which comprises not only CPUs but also (and mainly) disks usage and cooling fans.



(a) (b) (c) (d)

Fig. 1: Workload burst performances: a) completion time, b) effective processor speed c) context switches and d) system power consumption

TABLE IV: Performance metrics (1 instance, 1 thread)

Metrics	**Mean**		**95%CI**		**99%CI**	
	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 V: Performance metrics (12 instances, 8 threads)

Metrics	**Mean**		**95%CI**		**99%CI**	
	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

TABLE VI: Performance speed-ups 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 Thread - 1 Instance	35	9.7	339
8 Thread - 12 Instances	84	7.2	604

applications. However, this policy has a significant impact on the number of migrations and context switches, with a corresponding degradation of all the other architectural metrics, as reported in Tab. V. 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. The net effect is, in scenarios with high congestion, to partially serialize the execution of demanding applications. As the figure demonstrate, this results into a smaller completion time but also an higher power efficiency, which always improves between 7% and 11%. It is worth to notice that the combined effect on reduced workload completion time and better power efficiency has a even more interesting effect on the overall energy consumption, which improves up to 6 times on more congested scenarios, as reported in Tab. VI.

An overall graphical representation of the speedups for each considered metric is reported in Fig. 2 for the four scenarios of the previous table. In this figure, a positive bar corresponds to an improvement while a negative bar represent a deficiency of the integrated application with respect of the original one. These graphs indicates that the BBQ managed version of the

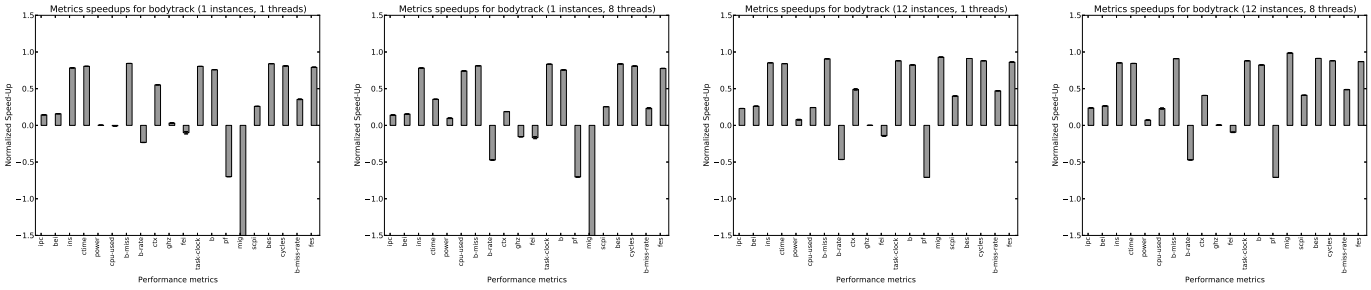


Fig. 2: Benefits and loss on the considered performance metric. Positive bar represents speed-ups, while negative ones corresponds to degradations, of the run-time managed Bodytrack with respect to the original version.

application is much better performing in all the congestion levels corresponding to the different scenarios.

Among the negative metrics we could notice the number of ”migrations” (mig), however it is worth to notice that this degradation is reported just on lower congestion scenarios (i.e. first two graphs) where the number of migrations even being different is a relative small number and with the same order of magnitude in both cases (ref Tab. IV). When instead the congestion level increases (i.e. last two graphs) the difference in number of migrations operated by the Linux CFS scheduler on the original unmanaged version increases a lot up to a two order of magnitude difference with respect to the case of the run-time managed version. At higher level of congestion we could better appreciate the benefits of a constrained resources assignment operated by the BarbequeRTRM framework.

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 run-time management library. This code re-organization inhibits some compiler optimization, namely loop-unrolling could not be applied and that’s the main reason for that degradations. A different integration is possible, by moving a loop portion within the required call-back method, however this has been left as future investigations.

All the other metrics benefits from the integration with the BarbequeRTRM framework. It is worth to notice 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 on the number of stalled-cycles per executed instruction (scpi). These last three architectural metrics indicate that the instruction stream is well optimized, and such a results has been achieved by treading compile time optimization discussed so far with a better assignment of resources at run-time.

VI. Conclusion

This work introduces a new user-space approach to run-time resource management which extends the advanced and efficient resources control capability offered by modern Linux kernels with suitable resources partitioning policies.

In this paper, we focused on evaluating the effectiveness to exploit the Control Group Linux framework to mandatory assign a set of computational resources to concurrently running applications. The experiments have been based on a workload from the PARSEC v2.1 benchmark suite, which has been made run-time tunable and integrated with our framework. Initial results show the effectiveness of the proposed approach considering a wide range of performance metrics. The proposed solution improves up to 80% the *execution time* and 6 time the *energy efficiency* for the considered workload, in many and different system congestion scenarios.

Future works targets the integration of other representative workloads as well as the assessment of the proposed approach on more realistic usage scenarios.

References

[1] Bellasi, P. and Massari, G. and Fornaciari, W. A RTRM proposal for multi/many-core platforms and reconfigurable applications. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, july 2012.

[2] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. Romero, and C. Scordino. Resource management on multicore systems: The actors approach. *Micro, IEEE*, 2011.

[3] S. Blagodurov and A. Fedorova. User-level scheduling on numa multicore systems under linux, 2011.

[4] X. Fu and X. Wang. Utilization-controlled task consolidation for power optimization in multi-core real-time systems. 2011.

[5] H. Hoffmann, M. Maggio, M. D, A. Leva, and A. Agarwal. SEEC: A framework for self-aware computing. Technical report, 2010.

[6] S. Hofmeyr, C. Iancu, and F. Blagojević. Load balancing on speed. 2010.

[7] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. 2007.

[8] Linux. Control Groups, 2006.

[9] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria. An industrial design space exploration framework for supporting run-time resource management on multi-core systems. 2010.

[10] MontaVista. Beyond Virtualization: the MontaVista approach to Multi-core SoC and Resource Allocation and Control, 2010.

[11] T. Sondag and H. Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. 2011.

[12] P. Yang and F. Catthoor. Pareto-optimization-based run-time task scheduling for embedded systems. 2003.

[13] C. Ykman-Couvreur, E. Brockmeyer, V. Nollet, T. Marescaux, F. Catthoor, and H. Corporaal. Design-time application exploration for mp-soc customized run-time management. 2005.