Proceedings of 1st Workshop on Resource Awareness and Application Autotuning in Adaptive and Heterogeneous Computing (RES4ANT) 2016

Resource-Aware Application Execution Exploiting the BarbequeRTRM

Giuseppe Massari, Simone Libutti,
William Fornaciari, Federico Reghenzani
and Gianmario Pozzi

Politecnico di Milano
DEIB: Dipartimento di Elettronica, Informazione e Bioingegneria
giuseppe.massari@polimi.it

Abstract. Energy efficiency and thermal management have become major concerns in both embedded and HPC systems. The progress of silicon technology and the subsequent growth of the dark silicon phenomena are negatively affecting the reliability of computing systems. As a result, in the next future we expect run-time variability to increase in terms of both performance and computing resources availability. To address these issues, systems and applications must be able to adapt to such scenarios. This work provides a brief overview of the Barbeque Run-Time Resource Manager (*BarbequeRTRM*) and the application execution model that it exploits, in order to deal with run-time performance and available resources variability.

1 Introduction

The need of *resource-aware* and *adaptive* applications is driven by several issues and requirements that are typical of modern computing systems. For instance, embedded mobile devices must deal with the limited energy budget provided by the battery, while HPC centers must afford huge costs due to the power consumption and the cooling of the infrastructure. Furthermore, the *dark silicon* phenomenon affecting modern processors is becoming prominent[1], since it is increasing the amount of silicon area that must be turned off, to guarantee the power envelope of the processor. For all these reasons, a continuous and full usage of the whole set of system computing resources is often impossible to achieve.

On the application side, we can gain efficiency by implementing suitable adaptive behaviors like enabling/disabling the execution of a task, or scaling the accuracy of the output depending on the availability of computing resources. A run-time resource management framework can implement such approach by constraining the resource allocation according to system level requirements or runtime conditions, and providing to the applications suitable interfaces to check and negotiate the resource assignment.

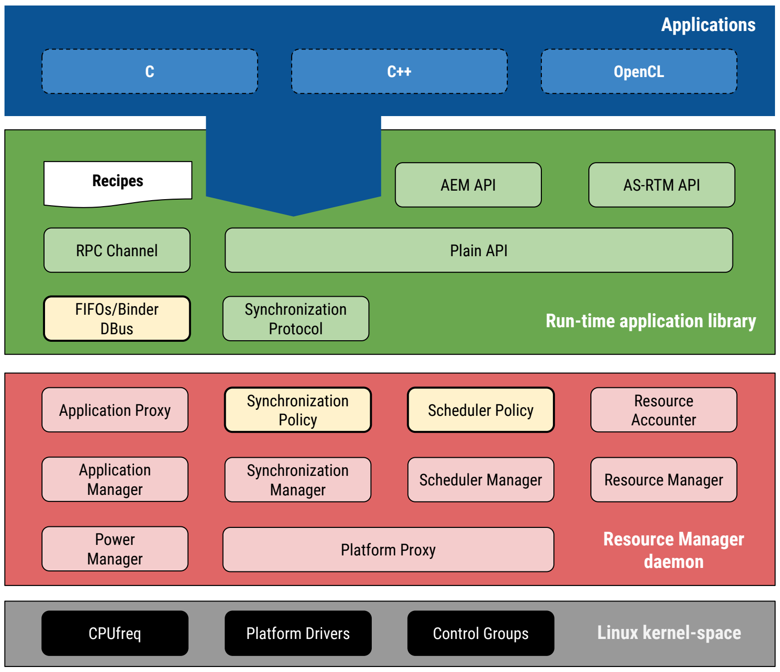


Fig. 1. The BarbequeRTRM Architecture. On top the programming languages supported by the application Run-Timr Library (RTLib). In red the resource manager core, on top of the support provided by the Linux OS to control the system resources.

2 Run-time Resource Management

The *BarbequeRTRM* is a modular and portable run-time resource manager targeting both embedded and High-Performance Computing (HPC) systems. From the hardware resources perspective, the framework can manage homogeneous and heterogeneous multi-core processors, as well as heterogeneous systems including devices characterized by completely different ISA (e.g., CPU and GPU).

The *modularity* of the BarbequeRTRM comes from a software architecture in which we can distinguish between *core* components and *plugin* modules. Typically, the latter are platform-specific extensions and selectable resource management policies.

The *portability* instead, is guaranteed by the exploitation of some underlying Linux operating system frameworks, like `cpufreq` and `cgroups`, that allows the *BarbequeRTRM* to enforce the resource allocation decisions [2].

2.1 Abstract Execution Model

The resource manager exposes its services to the applications through a run-time library (*RTLib*). The library accomplishes a two-fold objective: 1) to provide a

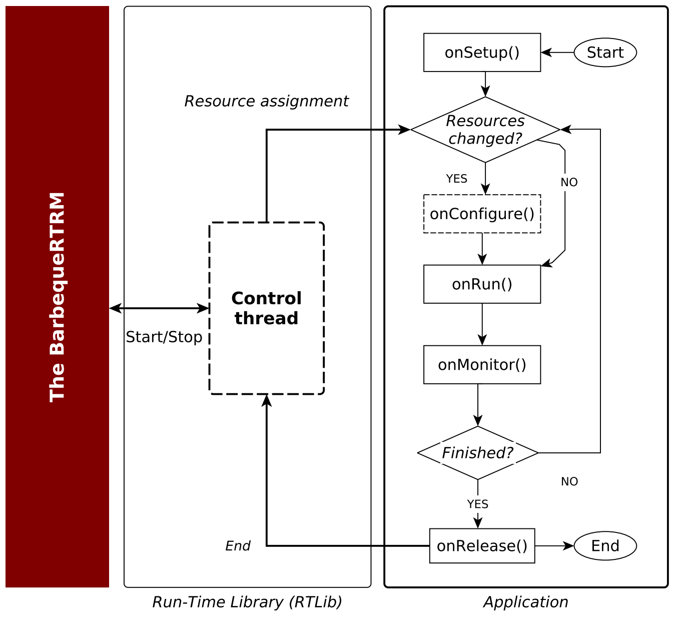


Fig. 2. Abstract Execution Model

communication channel between the resource manager and the applications; 2) to expose an execution model to support the implementation of the resource-aware adaptive execution of the applications[3].

In Figure 2 we show the Abstract Execution Model (AEM), that the run-time manageable applications must implemented accordingly. This execution model is put in place by defining and implementing a suitable C++ class, derived from the `BbqueEXC` class provided by the RTLib.

At run-time, the `BbqueEXC` member functions are called by a control thread, which is responsible of synchronizing the application execution with the decisional process of the resource manager. The rationale behind each member function implementation is the following:

`onSetup()`: setting up the application (initialize variables and structures, starting threads, . . .). `onConfigure()`: check the amount of assigned resources and configure the application accordingly. `onRun()`: single cycle of computation (e.g., computing a single frame during a video encoding). `onMonitor()`: performance and QoS monitoring. `onRelease()`: cleanup and termination code.

Therefore, once the application ends the initialization step (`onSetup`), the control thread waits for the resource allocation decision coming from the BarbequeRTRM. As soon as it has been received, the `onConfigure` function is called. In this function, the application can then check the amount of assigned resources, and configure itself accordingly, before starting (or continuing) the execution, as sketched here below.

```
RTLIB_ExitCode_t BlackscholesEXC::onConfigure(int8_t awm_id){
    // Get the number of CPU cores assigned
    GetAssignedResources(RTLIB_ResourceType::PROC_NR, nr_cpu);

    // Configure ...
}
```

The functions `onRun` and `onMonitor` are then sequentially called and executed in a loop, until the entire computation is over.

The RTLib estimates the current performance of the application, in terms of *cycles-per-second (CPS)*, such that the application could check the gap between the required performance level and the one currently achieved. After that, the application can notify the resource manager about this gap.

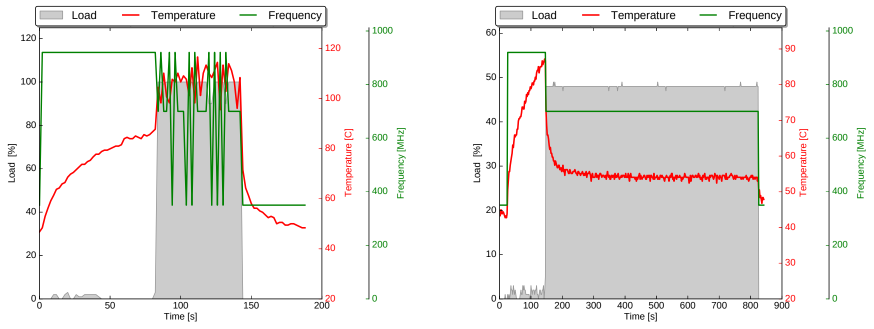
Considering also that the performance goal can vary depending on input data and external events, a effective approach is to exploit the `SetCPSGoal` function to specify the performance goal and the notification rate, as shown in the following example of `onMonitor` implementation:

```
RTLIB_ExitCode_t BlackscholesEXC::onMonitor() {
    // Specific event condition triggering the
    // change of performance requirements
    if (...)
      SetCPSGoal(2.5, 10);
    // ...
}
```

In the example, the application sets a performance goal of 2.5 CPS, and a notification rate of 10 cycles. The library keeps track of the application performance, computing the average CPS value over a (configurable) number of last execution cycles. Whenever the performance gap overcomes a given (configurable) threshold, such a gap value is sent to the resource manager. As a consequence, the amount of assigned resources can be adjusted accordingly. The notification rate is then exploited to bound the application reconfiguration rate, and hence the related overhead. In other words, the application asks the resource manager to send back a reconfiguration request after not less than 10 execution cycles or more.

3 Experimental Scenario

In this section we show results of the resource-aware adaptive execution of *blackscholes* from the PARSEC benchmark suite [4] on a embedded development board that features an ARM Cortex A9 dual-core CPU. The benchmark has been properly modified to fit the Abstract Execution Model. The frequency of the CPU has been set to its maximum value, which is 920 MHz. The full CPU usage, which is shown in Figure 3a, causes the chip temperature to raise over 100°C, thus triggering the thermal throttling response of the operating system.



(a) CPS = 4 (b) CPS = 1

Fig. 3. PARSEC *blackscholes* execution: CPU load, temperature and clock frequency variations according to two performance requirements: a) 4 cycles-per-second; b) 1 cycle-per-second.

A continuous frequency scaling is operated in order to cool down the CPU, with performance variability as a further consequence.

In Figure 3b, the application sets a performance goal of CPS=1. The resource manager takes into account such information shrinking the amount of CPU time assigned. The implicit result is a lower but more stable performance level, along with a reduced thermal stress.

References

1. H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA ’11. New York, NY, USA: ACM, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108
2. P. Bellasi, G. Massari, and W. Fornaciari, “Effective Runtime Resource Management Using Linux Control Groups with the BarbequeRTRM Framework,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 2, p. 39, 2015.
3. G. Massari, E. Paone, P. Bellasi, G. Palermo, V. Zaccaria, W. Fornaciari, and C. Silvano, “Combining application adaptivity and system-wide resource management on multi-core platforms,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on.* IEEE, 2014, pp. 26–33.
4. C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT ’08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128