Copyright Notice

The document is provided by the contributing author(s) as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. This is the author's version of the work. The final version can be found on the publisher's webpage.

This document is made available only for personal use and must abide to copyrights of the publisher. Permission to make digital or hard copies of part or all of these works for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage. This works may not be reposted without the explicit permission of the copyright holder.

Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the corresponding copyright holders. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each copyright holder.

IEEE papers: © IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The final publication is available at http://ieeexplore.ieee.org

ACM papers: © ACM. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The final publication is available at http://dl.acm.org/

Springer papers: © Springer. Pre-prints are provided only for personal use. The final publication is available at link.springer.com

QoS Manager for Energy Efficient Many-Core Operating Systems

Simon Holmbacka, Dag Ågren, Sébastien Lafond, Johan Lilius
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 FIN-20520 Turku
Email: firstname.lastname@abo.fi

***Abstract*—The oncoming many-core platforms is a hot topic these days, and this next generation hardware sets new focus on energy and thermal awareness. With a more and more dense packing of transistors, the system must be made energy aware to not suffer from overheating and energy waste. As a step towards increased energy efficiency, we intend to add the notion of QoS handling to the OS level and to applications. We suggest the design of a QoS manager as a plug-in OS extension capable of providing applications with the necessary resources leading to better energy efficiency.**

***Keywords*-QoS, Distributed Operating Systems, Many-Core Systems, Energy Efficiency**

I. Introduction

Pollack's rule [1] describes the performance increase of a CPU as an increase proportional to the square root of the increase in core complexity. As a result of this rule, chips with less complex but more cores are becoming popular. Twice the core complexity will, according to Pollack's rule, result in only about 40% performance speed-up, while using the same amount of transistors for adding more available cores increases the performance potentially by 70-80% [1]. While processing power can be increased by simply adding more cores, developing the software for many-core chips utilizing the parallelism is not trivial. Scalability issues can arise from both performance bottlenecks and new types of power constraints these chips introduce. In this paper we tackle the energy and thermal issues present in many-core chips due to the difficulty of managing the power dissipation efficiently.

Performance is usually maximized by spreading out tasks evenly on the chip, which also results in less thermal hotspots. On the other hand if tasks are scheduled to only a few cores, the idle cores could be shut down and as a result the total energy consumption decreases. A problem arises from this dynamic management and from optimization for energy efficiency without introducing performance degradation. Applications in computer systems usually strive towards high performance; an aim in the opposite direction to lowering the energy consumption. The compromise is to lower the energy consumption as much as possible while still provide the necessary processing power. For this cause we will investigate how to allocate the right amount of resources to the applications at the right time in order to provide sufficient Quality-of-Service (QoS).

QoS is a metric describing the *level of performance compared to a stated specification* [2]. By introducing QoS awareness into the applications, resource allocation can become more energy efficient because applications can deliberately ask for only a defined amount of processing power. The notion of QoS extends applications' influence over resources, and expands energy scalability by enabling control of the resource distribution. A QoS manager is therefore vital to a system level point of QoS control.

This paper presents the design of a QoS manager capable of regulating performance of many-core operating systems. The suggested manager is an OS service to which applications and other OS services can connect and establish the information flow necessary for QoS control. Besides the QoS manager, this paper also focuses on the declaration language the applications use to express their requirements. The contributions of the paper are:

- The QoS manager: a standardized link between applications and resources
- The possibility of applications to hint their resource need to the OS

II. Notion of Performance and QoS

QoS is a term used in real-time systems [3], usually in order to describe the relation to soft deadlines. It is also a term used in cloud computing [4], to enable the *selling* of a bundle of processing power to the user with a certain quality. In both cases, QoS is used to describe the average feasibility of a system without looking at sharp deadlines – a feature we intend to extend the OS with. This notion will enable us to create a more energy efficient system.

Energy is consumed as cores dissipate power over time and by the cooling infrastructure required for actively leading the heat away. Power is required for executing tasks on the processing elements, which in turn create the waste heat. In order to create an energy efficient system, the tasks should: a) execute on the appropriate execution unit and b) be only allocated the necessary amount of resources. For this, the notion of performance is an important measurement for deriving QoS values and how well an application is able to satisfy the user.

The QoS value for an application is determined by comparing the performance requirement in the specification with the actual measured performance. The ratio between these two values is the drop in QoS. If the QoS drop is more than allowed by the specification, the system must

control some actors giving the application more resources and thus higher QoS. Similarly, if the performance is too high, the system should decrease the amount of resources to the corresponding application in order to reduce energy waste. For this reason we suggest a new single entity – the QoS manager – controlling the QoS for the applications.

III. The QoS Manager

The presented QoS manager is implemented as an OS extension. The manager is able to measure QoS values from the applications (referred to as sensors), and with the obtained information control the resource allocation on system level. The structure of the manager is shown in Figure 1. It contains the manager, applications and actuators interconnected. The system is built from a *sensor-controller-actuator* structure shown in Figure 1, and is described below.

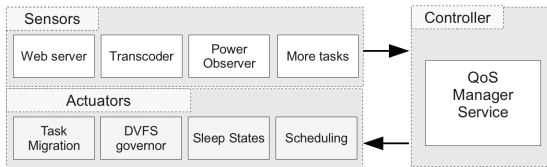

Figure 1. Structure of the OS containing the QoS Manager

Sensor: A sensor is a unit capable of connecting to the manager, expressing QoS and sending measurements to the QoS manager, hence the name *sensor*. Initially, the sensor registers its own QoS requirements and type of resources to use to the QoS manager with a declaration language described in Section IV. Afterwards, the sensor measures its own performance periodically with an implementation specific mechanism in the sensor itself. The performance values are sent to the QoS manager for QoS evaluation. As long as the values are within the specified QoS range, the task of the QoS manager is simply monitoring. In Figure 1 three sensors are connected to the controller: *web server*, *transcoder*, and *power observer*.

Controller: The controller is the part managing the link between sensors and the resources. It contains a database over all established sensor connections and the control unit. Furthermore, the controller handles the resource allocation if the measured QoS from the sensors is too low. Since the sensors are able to hint what kind of resource they lack in such a situation, the controller functions as a plug-in system connecting application directly to the right part of the hardware. All control theoretical implementations are put into the control unit. In this paper we used a simple P-controller (proportional controller) due to its simplicity. The P-controller determines how *much* more/less resources should be allocated to a certain sensor depending on its QoS value. In future work, we intend to investigate more advanced control methods.

Actuator: Actuators are units capable of indirectly altering the sensors' performance by regulating some resources (hardware or software). The way performance increases is sensor dependent, which means that such an actuators must be available, that the requests from the sensor can be fulfilled. A common actuator is the DVFS governor capable of setting the CPU voltage and frequency of a processor. Other actuators could handle sleep states, or migrate tasks from core to core to adjust the level of parallelism. Specific hardware related actuators could shut down memory banks on demand to decrease power dissipation. Even fan controllers can be used to set the fan speed for energy efficient cooling. Sensor choose which actuators to use with a declaration language describe in the following section.

IV. Declaration Language

A simple language has been derived to let the programmer determine QoS requirements for sensors and what actuators are connected to the sensors.

Overview: The declaration language is used during the implementation of the applications, and is compiled to c-code used for the registration and transmission of measurements used in the sensors. Rules and measurements are sent to the QoS manager during runtime. The sensors should therefore use the language to describe required QoS. A template for using the language is shown in Listing 1 and explained below.

```
QoS  MyTemplate {
     requirements{
        boundary:   <condition1>:   <value>;
        boundary:   <condition2>:   <value>;
        ... }
     priority   <value>
     control{
        actuator:    <Actuator candidate1>   <sign>;
        actuator:    <Actuator candidate2>   <sign>;
        ... } }
```

Listing 1. Template for specifying QoS

The declaration language used in the sensors is divided into fields for expressing the performance and QoS. A field contains an entity needed to specify what is intended from the system upon a measurement.

Requirements: The requirements field describes the actual limits for determining QoS boundaries. QoS requirements in form of a performance description is therefore inserted in this field and is compared against a selected `setpoint`. By setting a setpoint, the QoS manager can relate the performance measurements to what would be considered too low (poor performance) or too high (energy waste). In order to specify the accepted range of performance the user must also specify a QoS `limit`, which gives the lower bound of what is considered acceptable. For example the programmer of a webserver can choose a `setpoint` of 500 *requests/sec* and the QoS `limit` of 450 *requests/sec*.

Priority: The priority field determines which (if exists many) of the connected sensors have the highest weight. Situations can occur in which two different sensors' requirements completely conflict each other. In these cases the priority selects how much is weighted from which client. The priority from a thermal guard would for example be prioritized higher than a performance request from a web server if physical damage is imminent because of heat issues. Currently the weighing system is implemented to discard lower prioritized measurements in favor of measurements with higher priority. A more comprehensive way of expressing priorities is part of future work.

Control: The control field is used to describe what actuators should be used by the sensors. Actuators are chosen name wise based on available actuators in the controller database. All actuators are related to a control sign (+ or -). The sign determines in which direction the actuator should aim its output signal for the specific sensor. An example shown in Listing 2 describes is a power observer which strives to minimize the power dissipation of the system.

```
control{
    actuator:   CPU_freq,   −;
    actuator:   CPU_nr,     −;
    actuator:   Parallelize,  −;
}
```

Listing 2. Example of control for a power observer

This sensor has a negative signs on CPU frequency and number of active CPU cores as it aims to shut down and scale down cores in order to reach low power dissipation. It also tries to parallelize as little as possible in order to enable the shut down of cores. A webserver or transcoder could, on the other hand, use positive signs to request more resources if the QoS drops too low.

V. EVALUATION

We evaluated a simple system with two sensors: a JPEG decoder which decodes JPEG images in an infinite loop and a power observer which is used to keep the dissipated power under a certain value in order to act as an on-demand power saving mode. The applications were run on top of FreeRTOS [5]. The system was mapped on the Versatile Express board equipped with an ARM Cortex-A9 based CoreTile 9x4 quad-core chip running at 400 MHz with 1 GB of DDR2 memory. Our FreeRTOS port is available at [6]. Figure 2 shows four CPU cores each running one separate instance of FreeRTOS. Our system consist of one master core running the QoS manager and three worker cores. Each core has one JPEG decoder task running. Each decoder task runs completely independent of the other decoders. In this architecture, we are therefore able to decode up to four pictures in parallel.

Measurement data describe how many pictures per second (`p/s`) a core is able to decode, and is sent to the master. The total sum of `p/s` of all four separate JPEG decoding

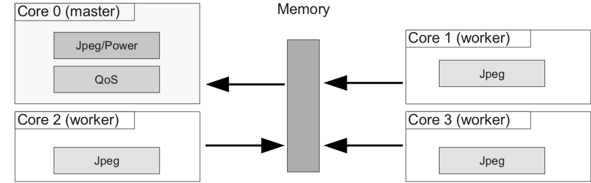

Figure 2. Mapping of the QoS manager on an ARM Cortex-A9 quad-core

instances gives the final `p/s` number for the whole system. Furthermore, the master core implements a power observer, which is used as a on-demand power saving feature. A sleep state mechanism was implemented as actuator; giving the master core the opportunity to shut down individual worker cores in order to lower the power dissipation. Experiments were conducted to show how the energy consumption behaves according to what performance requirements are set in the QoS manager.

Without power requirements: The first set of experiments were conducted without power requirements. Table I shows the requirements for the JPEG decoder in five different tests. The first test (1) has a performance requirement of 7.5 `p/s` with a QoS of 93.3% etc.

Table I
REQUIREMENTS FOR THE FIRST EXPERIMENT

Test nr.	1	2	3	4	5
Setpoint [p/s]	7.5	5.5	4.0	3.8	2.0
QoS [%]	93.3	90.9	97.5	84.2	75.0

Results from the first run is shown in Figure 3. Figure 3(A) shows the picture rate of each test run. From the figure it is clear that the cases with a steady curve are successfully provided with the demanded resources most of the execution time. The oscillating curve is a result of demanding such a picture rate that 2.5 active cores are required. To avoid the oscillations, additional actuators such as DVFS could scale down one core in order close the gap between setpoint and requirement more exact. The power dissipation was also measured during the same experiments. Figure 3(B) shows the power output from the same use cases as in Table I, with the oscillating case (test 3) removed for illustrative reasons.

Figure 4 shows the final energy consumption for a 5 minute run on the Cortex-A9 for all mentioned test cases and four additional configurations. It shows clearly how the energy consumption increases steadily as the performance requirements (`p/s`) increase. The result is a nearly linear relationship between performance and energy consumption due to the QoS manager with the sleep state actuator.

With power requirements: The next set of experiments included power requirements to give a more realistic situation with multiple parameters to match. Similarly to the first experiment, a JPEG decoder was used as application with the same range of performance requirements. A power observer application was added to the system. The power observer measures the power dissipation of the chip. Ideally

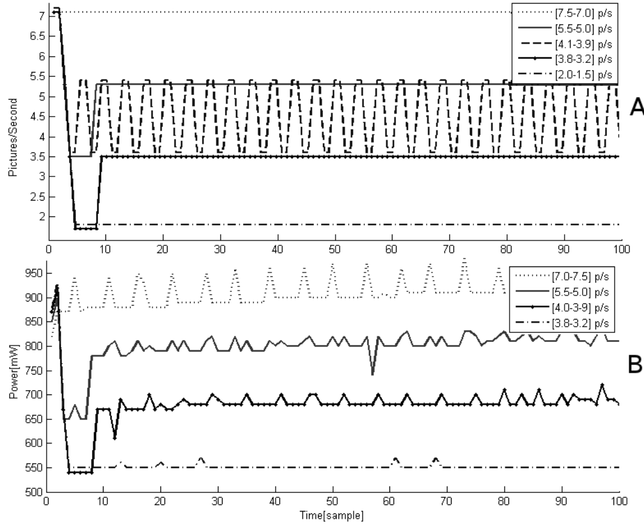

Figure 3. Power dissipation with different performance settings

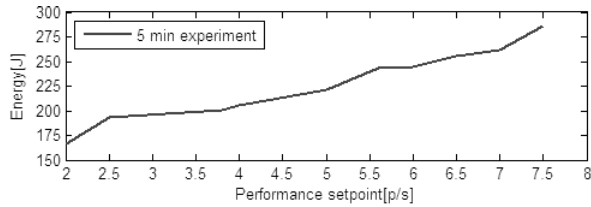

Figure 4. Energy consumption with different performance settings

it requires 650 mW of dissipated power, but accepts power dissipation up to 700 mW. The power observer uses also a higher priority than the JPEG decoder sensor to function as a power saving and heat protection feature.

With these settings experiments were run for the first settings in Table I. Without power constraints the system activated four cores during the whole test and dissipated roughly 900mW on average.

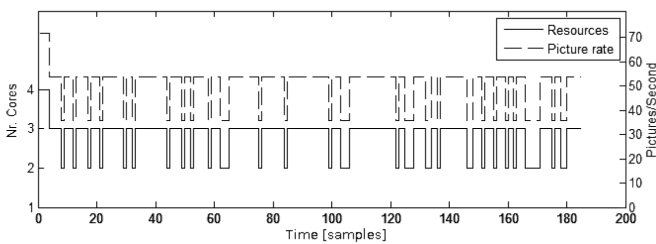

Figure 5. Results from experiment with power constraints ([7.5-7.0]$\mathrm{p/s}$)

By adding power constraints, the system is forced to shut down some cores in order to meet the higher prioritized requirement from the power observer. Figure 5 shows how the system is forced to operate mostly with three active cores with occasional usage of only two cores. This experiment shows that the system is able to override requirements on demand by higher prioritized sensors in order to obtain power saving feature etc.

VI. Related Work

QoS management and monitoring exist in different areas; from cloud infrastructures and web servers [4], [7] to OS level on single computers and real-time systems [3], [8], [9] etc. Language constructs for injecting QoS support has also earlier been presented. Aagedal presented in his PhD thesis [2] CQML; a language having the property of describing QoS requirements. In this work, applications specify what performance is to be expected from it and what is considered as performance in context of the application. Applications also monitor own performance and signal this value to the QoS manager periodically.

We use similar notations inspired by the languages to describe QoS in applications, with more focus on the OS-level support. Our manager will be implemented as an OS extension capable of system level control many-core systems. Furthermore we have added the control output, by which applications can choose which action needs to be taken if the desired QoS is not achieved.

Design choices for a run-time manager was presented in [10], consisting of a resource manager and a quality (QoS) manager. The task of the QoS manager was to optimize an operation point such that the system is maximally utilized. Utilization is controlled by adjusting quality points in the applications i.e. selecting one of many performance levels an application specifies. Video resolutions or frame rate for a transcoder are examples of such performance levels. Similarly in [11], a system PowerDial is used to insert configuration parameters (knobs) into applications and tune their values to achieve the best accuracy vs. performance trade-off. Complementary to Hoffman's work [11], his application tuning knobs can be used as a single actuator in our model, which forms an application to be both a sensor and actuator at the same time. Instead of controlling the applications, our manager is intended to only monitor the applications which indirectly influence the resources.

The managers in [9] and [10] require the application programmer to specify required processing power, memory and communication capabilities. We intend to simplify requirement notation by only requiring an abstract *quality* value freely defined by the programmer. The programmer does not need to modify the application or analyze performance points in order to use the presented QoS manager.

VII. Conclusions

In this paper we have introduced a QoS manager for improving the energy efficiency of many-core systems. The manager is intended to make the system better utilize the resources of the platform depending on the workload. Applications are referred to as sensors; actors capable of declaring performance and QoS requirements. By introducing the notion of QoS, sensors are able to signal their resource

requirements and, through the QoS manager, allocate the resources. The QoS manager control a set of actuators capable of altering the performance characteristics for the sensors. Sensors also set what type of actuator is required for increasing performance of a certain type of sensor, which gives the programmer opportunities to tailor the resources more exact to the application. It also allows future optimization techniques to be plugged in to the QoS manager and used by any sensor if suitable.

The QoS manager has been evaluated on a quad-core ARM Cortex-A9 with a JPEG decoder and its picture rate as use case. The experiments have shown that the QoS manager is able to scale down the energy consumption of the chip in two different ways. Firstly, the application can by itself relax the performance requirements to a given rate and thereby request less resources. Secondly, other sensors with higher priority can force the system to allocate less resources to lower prioritized sensors.

In contrast to current systems, more awareness on the thermal distribution inside the chip must be made when using many-core systems because of the very dense packing of cores and the spacial locality. Controlling QoS will therefore be an important part of the many-core evolution. By using the system level QoS manager, the distributed many-core system can more easily be optimized for a global maximum since the applications can hint the controller of how resources should be used.

VIII. Future Work

An issue not addressed in this paper is the control theoretical view of the QoS controller. Since this part is the system level of control, methods such as PID or MP or fuzzy control should be tested and evaluated complete with stability analysis and tuning rules etc. As this system uses multiple inputs from sensors and multiple outputs to the actuators, a state spaced-based method could enable the possibility for constructing a more efficient controller. Other alternatives would be to formulate the system as a optimization problem in which the objective function minimizes the power dissipation and QoS requirements are the constraints. This would also improve the current priority model since the system would, with more rigorous methods, determine the lowest total cost (power vs. performance) of the system.

The complexity of the controller is also an important parameters especially in a large many-core system, as the number of inputs/outputs is likely to grow rapidly. As the number of cores grow towards extreme numbers (1000+) a single manager will become a bottleneck for communication even if the complexity is very low. To solve the issue, the manager must be decentralized and function as a distributed system with sub-managers handling certain *islands* of cores eventually grouped into *continents* of cores.

We intend to develop the complete environment for demonstrating the scaling effects of the QoS manager on a true many-core platform such as the SCC [12] or TilePro64 [13] and also construct the necessary actuators needed to control such a system efficiently. For example energy efficient scheduling, task migration and dynamic voltage and frequency scaling are techniques useful to create the required actuators. We also intend to use a more complex mix of applications requesting different actuators with different priorities for a more realistic conclusion.

References

[1] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual DAC*. New York, NY, USA: ACM, 2007, pp. 746–749.

[2] J. Aagedal, "Quality of service support in development of distributed systems," Ph.D. dissertation, University of Oslo, Oslo, Norway, March 2001.

[3] F. Monaco, E. Mamani, M. Nery, and P. Nobile, "A novel qos modeling approach for soft real-time systems with performance guarantees," in *HPCS '09.*, june 2009, pp. 89 –95.

[4] P. Zhang and Z. Yan, "A qos-aware system for mobile cloud computing," in *CCIS, 2011 IEEE International Conference*, sept. 2011, pp. 518 –522.

[5] R. Barry, *FreeRTOS Reference Manual: API functions and Configuration Options*, Real Time Engineers Ltd, 2009.

[6] D. Ågren. Freertos cortex-a9 mpcore port. Åbo Akademi University. [Online]. Available: https://github.com/ESLab/FreeRTOS---ARM-Cortex-A9-VersatileExpress-Quad-Core-port

[7] T. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: A control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, p. 2002, 2001.

[8] B. Li and K. Nahrstedt, "A control-based middleware framework for quality-of-service adaptations," *Selected Areas in Communications, IEEE Journal on*, vol. 17, no. 9, pp. 1632 –1650, sep 1999.

[9] V. Segovia, "Adaptive cpu resource management for multicore platforms," Licentiate Thesis, Lund University, Sep. 2011.

[10] V. Nollet, D. Verkest, and H. Corporaal, "A safari through the mpsoc run-time management jungle," *Journal of Signal Processing Systems*, vol. 60, no. 2, pp. 251–268, 2008.

[11] H. Hoffmann and S. Sidiroglou, "Dynamic knobs for responsive power-aware computing," in *Proceedings of the sixteenth ASPLOS conference*. New York, NY, USA: ACM, 2011, pp. 199–212.

[12] P. Thanarungroj and C. Liu, "Power and energy consumption analysis on intel scc many-core system," in *30th (IPCCC), 2011*, nov. 2011, pp. 1 –2.

[13] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, 2007.