ARTE: An Application-specific Run-Time management framework for multi-cores based on queuing models

Giovanni Mariani \(^a\), Gianluca Palermo \(^b\), Vittorio Zaccaria \(^b\), Cristina Silvano \(^b\)

\(^a\) ALaRI – Università della Svizzera Italiana, Lugano, Switzerland
\(^b\) Politecnico di Milano, Milano, Italy

Abstract

This paper presents an Application-specific Run-Time management (ARTE) framework to tackle the problem of managing computational resources in an application specific multi-core system. The ARTE framework run-time goal is to minimize applications’ response times while meeting the applications’ computational demands and fitting within the available power budget.

The approach addresses application specific embedded systems assuming that the set of target applications is known at design-time. In addition, it considers run-time scenarios that are unpredictable due to variable user activity and/or interaction with the external environment. ARTE takes decisions about resource distribution to the active applications at run-time once the system state is known.

ARTE leverages an analytical queuing model at run-time to predict the applications’ response times. The accuracy of this model is enhanced by accounting for contention overhead on the resources shared among the active applications. The analytical nature of the queuing model allows an estimation of the system performance with negligible overhead.

Finally, we compared the proposed ARTE framework to state-of-the-art techniques to assess its benefits in terms of systems performance and run-time overhead for the selected set of parallel benchmarks.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Nowadays Chip Multi Processors (CMPs) and Multi-Processors Systems on Chip (MPSoCs) have become de facto standard architectures in both the general purpose and embedded computing arenas.

The availability of many processors (or computing elements) on a single chip brought several new challenges to system designers; in particular, efficient power management has become a primary factor for product success with equal (if not even more) impact on the value of the system perceived by the consumer. This is evident for portable devices but has subtle yet important consequences on reliability and cooling costs for non-portable systems.

Traditional techniques for power management in multi-core systems consider switching off or slowing down the frequency of computational elements which are underutilized [5,11,29,3]. In fact, if the switching overhead is negligible and the performance is not saturated, it is possible to meet performance requirements with fewer active resources and lower power consumption. In this context a Run-time Resource Management (RRM) system, as part of the OS, acts by identifying a suitable frequency and voltage to be assigned to each power island. Practically speaking, the RRM tunes a set of run-time...
knobs (or parameters) by employing a wide range of heuristics to minimize the power consumption and, ideally, maximize the system performance. Overall, state-of-the-art RRM frameworks maximize the instantaneous throughput measured either in terms of instructions per second [11,5] or, from a higher level point of view, in terms of tasks completed (or jobs) per second [8].

We point out that portable devices dedicated to interactive multimedia services might benefit up to a certain point from the above approaches since the performance is actually a minimum requirement which varies with the Quality of Service expected by the user (e.g., a smooth reproduction of video). Once performance requirements are met, the system response time becomes a more malleable function to be traded off to meet a pre-determined instantaneous power budget while not sacrificing performance on the long term (i.e., making other threads starve). In this new perspective, our solution carefully relaxes the average time between the instant a job arrives (e.g., a new video frame has to be decompressed) and the instant in which the job is adequately served (e.g., the frame has been actually decompressed in raw pixels stored in the frame-buffer) to satisfy the requirements on power. Restating the problem in terms of queuing theory, response time allows us to adopt policies for minimizing the overall queue length associated with the execution of each application.

The target execution context of our techniques is composed of a multiprogrammed system where multiple applications compete for a sub-set of the available processors. The run-time tunable parameters handled by the RRM are the number of processors assigned to each application at run-time. As observed in [34], optimal processor assignment is very workload dependent. In this work, we assume that the set of target applications running on the system is known a priori. While this may be problematic from the general purpose computing standpoint, we address relatively small eco-systems populated by a somewhat known set of services or applications. This scenario is more realistic than one might think, as at least one example of modern embedded operating systems (such as Apple’s iOS 4) explicitly avoids an arbitrary number of competing tasks in favor of controlled tasking to contain energy consumption. The proposed approach (named Application-specific Run-Time mangEment framework, or ARTE [21]) aims at determining the relationships between power consumption, system performance and application parallelization during a Design Space Exploration (DSE) phase carried out at design time. The information gathered at design-time is then used at run-time to enable a queuing model-based processor allocation policy.

Dynamic voltage and frequency scaling (DVFS) is somewhat orthogonal to our methodology [11,5]. Our focus is to explore the actual impact of application parallelization on the Quality of Service and power consumption, hence DVFS has been omitted from the analysis performed in this paper.

The remainder of this paper is organized as follows. Section 2 introduces a motivating example that illustrates the complexity of the problem of processor assignment to applications in a multi-core system. Section 3 summarizes comparable approaches to the problem of run-time resource management for multi-core systems, while Section 4 introduces our mixed design-time/run-time optimization flow. In Section 5 we present in detail the proposed RRM named ARTE, while Section 6 introduces experimental data supporting the efficiency of our approach with respect to other state-of-the-art techniques. Finally, in Section 7 we present considerations on the generality of the proposed approach and in Section 8 we draw our conclusions. In addition, Appendix A provides the glossary of terms used in this work.

2. Motivating example

Let us consider that the user of a mobile device requires the reproduction of a video stream. It is well known that the human eye perceives the typical film motion as being fluid at about 25 frames per second (fps), because of its blurring. In a typical scenario, it is not worth increasing the frame rate of the encoded stream and, consequently, the throughput of the video decoder due to intrinsic psycho-visual limits of the human brain. Generally speaking, there are many applications (such as, audio processing or cellular-network communication) that have specific minimum throughput requirements (due either to features of the human brain or the limits of the communication medium) but additional throughput does not necessarily improve user experience.

The fundamental problem to be solved by the RRM is to assign the available computing elements to a set of running applications given the actual arrival rate of requests, and the allocated system-level power budget.

In this paragraph, we would like to show that a greedy optimization of the instantaneous throughput might lead to a sub-optimal ‘average response time’ on the long-term. To emphasize this point, let us consider a CMP composed of 16 out-of-order MIPS processors which communicate via a shared bus.

We consider a system with two running applications (called $x_1$, $x_2$) that process a certain amount of data in chunks (or jobs) whose rate depends on the interaction between the user and the system. We consider that neither of the two applications has strict real time constraints. The task-parallelization (i.e., the number of cores assigned to each application for executing a single job) cannot be changed within the execution of the job itself but only between two distinct jobs. If an application is busy elaborating a job, all the successive jobs are stored in a FIFO structure (we will call it queue in the remaining sections of the paper). Let us consider that each application queue is initially set up with 15 jobs and that the allocated system-level power budget is 38 W.

Let us consider two case studies, Case 1 (C1) and Case 2 (C2). In both cases, we use a state-of-the-art greedy heuristic RRM that allocates processors such as to maximize the instantaneous throughput sum while fitting within the given power budget [20]. The only difference between the two use-cases is that the power budget of C1 is 43 W while the power budget of C2 is 38 W.
Effects of the greedy heuristic. In case C1 (Fig. 1(a)), the RRM greedy heuristic assigns one processor to \( a_2 \) and 8 processors to \( a_1 \), serving, in the initial phase, 56.2 Job/s with 42.4 W instantaneous power. In case C2 (Fig. 1(b)), the simultaneous throughput is maximized assigning 4 processors to both applications, serving 48.8 Job/s and consuming 37.6 W of instantaneous power.

Reducing the power budget in C2 decreases the overall instantaneous throughput. However, the case C2 has a 20% shorter overall completion time than C1. As said above, C1 presents a maximum throughput sum of 56.2 Job/s when \( a_1 \) is given 8 cores but slows down the consumption of jobs from the \( a_2 \) queue. This is a known starvation effect [6] that might arise when much faster applications compete for the resources with a priority that is somewhat inversely correlated to their job execution time.

The problem with the above scenario is that maximizing a function of the instantaneous throughput does not take into account how many jobs in the queues are waiting to be served. To deal with this problem we propose to model the run-time behavior using queuing models. We show that it is possible to use these models at run-time to estimate the application response time for different resource distributions and hence driving the RRM towards better solutions.

3. Related work

In a multiprogrammed multi-core scenario, different applications compete to access the system resources making the overall platform behavior very complex and even counter-intuitive. For example, the authors of [34] introduced experimental evidence about performance gains when some of the system processors are left idle to address unpredictable workload instead of being assigned to the active applications. The authors observed that the efficiency of different RRM schemes is strongly dependent on application specific features such as the scaling of the performance with respect to the number of allocated resources.

Among other works, the authors of [1] observed that, for multiprocessor workloads, the Instructions Per Cycle (IPC) is a poor performance metric and does not automatically reflect performance as a whole. It has also been noted that for multiprogrammed multi-core scenarios, system level performance metrics such as throughput measured in terms of Job/s or other user-oriented performance metrics are more suitable for maximizing performance [8].

Multiprogrammed workload analysis. When dealing with multiprogrammed multi-core scenarios program interactions on the shared resources are very complex to analyze [4,15]. To cope with this problem, high level system models of program interactions have been proposed [4,15,36].

Authors in [4] suggest to model contentions for shared resources by using statistical regression techniques. The works in [15,36] focus their attention on the execution of multiple single-threaded programs. Authors in [15] use a composable regression model to compute the contentions on the shared resources, while in [36] programs’ interactions on the shared cache are estimated by iteratively adjusting programs’ execution times based on the cache access model presented in [7].

RRM heuristics. So far, several works appeared in the literature assume that the applications set is known at design-time (either in the virtual prototyping stage or earlier). In [39,31], the authors introduce an RRM that solves a multi-dimension multiple-choice knapsack problem at run-time. In particular, at design-time, different operating configurations (e.g., allocated processors) are simulated for each application and several metrics such as average throughput and power consumption.
are saved. At run-time, once the set of active applications is given, the RRM heuristically chooses for each application one of the previously identified operating configurations. The authors of [38] generalize the methodology presented in [39] for heterogeneous multi-core systems and apply it to an industrial MPSoC scenario. In [20], a joint design-time/run-time DSE methodology has been proposed where the run-time operating configurations are chosen together with an optimal tuning of architectural parameters [32].

Other approaches found in the literature address unknown relationships between resource allocation and system performance. These relationships are learned at run-time by fitting an analytical model to observed system-level indices. In [2] a piecewise linear regression model is used for modeling the relationships between resource allocation, application performance and energy consumption while Martínez and Ipek [22] proposed machine learning approaches such as Artificial Neural Networks and Q-learning. In [18] a comparison of several RRM techniques is carried out. Among others, control theoretic approaches as the one presented in [17] are compared to machine learning techniques similar to the ones proposed in [22]. The work in [18,17] addresses a single application scenario where the goal is to minimize the power consumption while fitting in a given performance requirement. In [16] the approaches presented in [18] are also applied to multiprogrammed scenario.

We point out that, although literature presents a wide range of approaches to RRM under dynamic workloads [39,31,38,18,17,16,19,22,41] they rarely, if not at all, consider response times in favor of instantaneous figures of merit. A particularly interesting case study can be found in [33]. The authors show that the response time can be traded off with power consumption for a multimedia platform but they address only the management of DVFS on a single core platform without considering the concurrent execution of multiple applications.

In this work we propose to use queuing models to predict application response times. The results in the experimental section demonstrate that system performance is improved when the RRM goal is formulated as the minimization of application response time rather than the maximization of the instantaneous throughput.

This paper extends our previous work [21] by extending the ARTE approach to account for run-time overheads related to the contention of shared resources usage that are occurring due to the execution of multiple applications. Moreover this paper presents extensive experimental results that prove the advantages of ARTE with respect to different state-of-the-art techniques such as an advanced machine learning based approach [22,18,16] and a simple resource balancing methodology [11]. Finally, this paper discusses the implementation costs and the portability of the methodology to different multiprogrammed systems.

4. Overview

The proposed Application-specific Run-Time managEment (ARTE) framework aims at providing a suitable run-time policy for assigning processors to applications to minimize the response time while meeting a specified constraint on the power consumption. In this paper, we use code versioning [40] to change the parallelization of the application and thus we assume that the number of cores assigned to each application can be changed only between two distinct jobs. However, other mechanisms for manipulating the program representation to exploit available processors can be considered with their additional overhead (e.g., stream program fusion [9,10]).

In the proposed framework, the RRM decides how to allocate processors to active applications based on the following information:

- The power budget, which we assume to be set either at design-time or at run-time by the OS [11].
- The user activity, which generates, at run-time, a sequence of jobs to be served by the system with a set of minimum throughput requirements for each application. The throughput offered by the system should at least match the requirement expected by the user. As an example, if the user is re-producing a video, he expects a smooth reproduction at 25 frames per second.
- An application characterization is carried out at design-time by using a design space exploration heuristic. The application characterization consists of generating a set of operating points that specifies, for a number of processor allocation configurations, the associated figures of merit.

The overall framework provides support for the design-time and the run-time optimization phases as shown in Fig. 2.

At design-time, given the target application set $A$, an exploration algorithm is used to identify a set of operating points that represent, for each application $a \in A$, a tuple containing the parallelization of the application combined with its associated figures of merit. In this work, the considered figures of merit are: the power consumption $\psi$, the average job execution time $\tau$, and the access rate generated by communication request to the shared memory at the highest hierarchical level $\sigma$. This access rate (or equivalently the communication bandwidth) will be used to evaluate the program interactions.

The figures of metric reported in the application characterization represent the standalone execution behavior of an application’s operating point when no other applications are running.

Given the operating points gathered at design time (i.e., the application characterization), the RRM’s main task is to select the number of processors to be allocated to each application to meet the power budget and to offer the required throughput. We point out that, given the finite amount of memory used for storing jobs in the system, job queues should be consumed at
an average rate which is at least equal to the throughput required by the user (otherwise the system dynamic behavior is not stable). Whenever the required throughput cannot be sustained for the long-term (because of a significant divergence with respect to the design-time assumptions), corrective actions should be taken (e.g. dropping jobs) to maintain stability. This might be perceived by the user as sluggishness in the user interaction or multimedia reproduction.

**Modeling the run-time contention overheads.** In this work we present ARTE, an extension of the run-time resource management approach proposed in our previous work [21]. The main limitation identified in our previous work was the lack of modeling application interactions. Typically, when different applications are concurrently executed on the same multi-core system, they interfere with one another when accessing shared resources (e.g. the shared memory or shared communication medium).

In the present work, the ARTE methodology published in [21] is complemented with an analytical prediction model of the contentions on the shared resources. In this way, ARTE adapts its behavior to the estimated interferences.

To model application interactions at a high level, we use a regression model similar to the one proposed in [4]. In the considered architectures, the shared resources are part of the memory system (e.g. the level 2 cache and the main memory). The access rates to the shared memory at the highest level of abstraction are saved in the application characterization for each application \( a \). Given the applications \( a_1, \ldots, a_n \) that are running concurrently and their access rates \( r_a \), the regression model returns for each application \( a \) the average contention delay per single access \( \delta_a \). In particular, the contention delay is computed as a function of three access attributes [4]:

- \( n \), the number of applications that are concurrently executed;
- \( r_a \), the access rate of the application \( a \);
- \( \omega_a \), the overall access rate of the other active applications. This is computed as the sum of the access rates \( r_a \) for each application \( a \in A \), \( a \neq a \).

An analytical model \( f_a \) is generated at design-time for each application:

\[
\delta_a = f_a(n, r_a, \omega_a)
\]  \hspace{1cm} (1)

To generate the analytical model, we use linear regression. The regression coefficients are fit to the delay per access \( \delta_a \) observed from some experimental simulations run on a cycle accurate simulator. The applications to run concurrently during the experiments and their parallelizations are selected using a Design of Experiment (DoE) technique called **simplex lattice DoE** [27]. This DoE is thought to explore the space of a mixture of elements (in our case an application mix). The percentages of the elements in the mixture represent the percentages of computing resources allocated to the applications. We adopt the **simplex lattice DoE** since it distributes uniformly the experiments over the constrained space under study (i.e. using this DoE, the allocated computing resources are constrained so that they never exceed the available ones). The **simplex lattice DoE** includes an experiment for each possible pair of applications \( (a_i, a_j) \). During each of these experiments, 50% of computing resources are assigned to each application.
resources are allocated to each application in the pair.\footnote{2} To improve the model accuracy \cite{23}, we augment the simplex lattice DoE with the centroid experiment (i.e. we run all applications \( x \in A \) with a uniform distribution of computing resources).

During the experiments we measure for each application the value of \( \delta_x \). The analytical model \( f_x \) is then generated for each application \( x \) using linear regression. In order to improve the model quality, the measured \( \delta_x \) is first pre-processed by using the Box-Cox transformation functions \cite{13}. For our case studies we noted that the best model is obtained when using the logarithmic transformation. In practice, linear regression is used to estimate the coefficients \( a_x, b_x \) and \( c_x \) of the following function:

\[
\log(\delta_x) = a_x \times n + b_x \times \sigma_x + c_x \times \omega_x
\]  

(2)

Once the model is constructed and validated in terms of its error rate, it can be used to approximate the contention overheads at run-time by ARTE. The execution time \( \tau_x \) of a job of the application \( x \) can be approximated as:

\[
\tau_x = \tau_x + (\sigma_x \times \tau_x) \times \delta_x(n, \sigma_x, \omega_x)
\]  

(3)

For instance, \( \tau_x \) is the execution time considering the standalone application execution (i.e. \( \tau_x \)) with the additional contention delay. The contention delay is given by the number of accesses (i.e. \( \sigma_x \times \tau_x \)) times the delay per access.

Note that \( \tau_x, \sigma_x, \) and \( \omega_x \) depend on the applications’ versions loaded on the platform. When deciding the resource allocation, ARTE interrogates the analytical model several times and predicts the expected service rate \( \mu_x \) for different operating configurations as follows:

\[
\mu_x = 1/\tau_x
\]  

(4)

5. ARTE

ARTE addresses the maximization of quality of service (QoS) given the instantaneous power budget constraint. For each application \( x \), we define the QoS \( Q_x \) as the inverse of the average response time \( R_x \), i.e., the average time measured between the submission of a job to the application and the time the job is completed (thus, including the average waiting time in the queue). We define the overall platform QoS \( Q \) as the geometric mean over the set \( A \) of the active applications:

\[
Q = \left( \prod_{x \in A} Q_x \right)^{\frac{1}{n}} = \left( \prod_{x \in A} R_x^{-1} \right)^{\frac{1}{n}}
\]  

(5)

The geometric mean, unlike the arithmetic mean, tends to dampen the effects of very high or low values, which might bias the mean if a straight average (arithmetic mean) were calculated \cite{12}. Indeed this is our case since we deal with a set of heterogeneous applications with very different workloads and throughput ranges. The actual response time \( R_x \) is known only after the application execution is completed. To enable run-time management, ARTE maintains and evaluates at run-time a prediction model of the expected average response time \( \bar{R}_x \).

*Prediction of the response time by using queuing models.* In this paper, we propose to model \( \bar{R}_x \) by using fundamental results coming from the theory of queuing networks \cite{35}. The model is based on the following assumptions:

1. The job arrival rate \( \lambda_x \) of an application \( x \) is the throughput required by the user. It is measured in Job/s and it depends on the user activity to be monitored at run-time. An application is considered active if \( \lambda_x > 0 \). This assumption simply states that all jobs issued by the user must be processed.
2. The job arrival rates \( \lambda_x \) of different applications are mutually independent. For instance, we assume that there are no interferences between throughput requirements. Interferences might only impact the job execution time.
3. The job arrival times can be modeled as continuous time Markov process \cite{33}, i.e. they are mutually independent. In particular job inter-arrival times are exponentially distributed with mean \( 1/\lambda_x \).
4. The average job execution time \( \tau_x \) for an application can be approximated with the analytical model presented in Eq. 3, as in \cite{4}.
5. For a given application, before starting the execution of a new job, all previous jobs should be completed.

\footnote{2} The simplex lattice DoE would also require an experiment to run each application \( x \) standalone using all the computing resources. We do not run these experiments because we consider contentions arising from the execution of different applications running concurrently.

Please cite this article in press as: G. Mariani et al., ARTE: An Application-specific Run-Time management framework for multi-cores based on queuing models, Parallel Comput. (2013), http://dx.doi.org/10.1016/j.parco.2013.04.002
The queuing model used in this paper is composed of queues and servers. The queue, in our case, is a structure residing in the system memory in which the jobs are stored waiting to be executed while the server is the set of system resources required for processing the jobs (i.e., the processors allocated to an application). Given that the jobs should be executed in order (Assumption 5, in the list above) and that arrival rates are mutually independent among applications (Assumption 2), we model each application with an independent queuing system with one server and one queue (Fig. 3).

We assume that the arrival of jobs can be modeled as a Markov process (Assumption 3) with a deterministic service time (M/D/1, i.e., Markov/Deterministic/1 server). The assumption on the deterministic service time is in line with the applications considered in our case study. However, it is rather easy to model stochastic execution time depending on the actual dataset as well (Section 7 reports additional considerations on this topic).

In the M/D/1 model [35], the expected number of jobs in the system (either waiting in queue or being served) in the steady state is given by:

$$N_s(\mu_s, \lambda_s) = \rho_s(\mu_s, \lambda_s) + \frac{[\rho_s(\mu_s, \lambda_s)]^2}{2[1 - \rho_s(\mu_s, \lambda_s)]},$$

(6)

$$\rho_s(\mu_s, \lambda_s) = \frac{\lambda_s}{\mu_s}$$

(7)

where $\rho_s$ is the server utilization, i.e., the fraction of time the server is busy. Given Eqs. 6 and 7, we can build a prediction model for $R_s$ by using Little’s Law [35]:

$$R_s(\mu_s, \lambda_s) = \frac{N_s(\mu_s, \lambda_s)}{\lambda_s}$$

(8)

We can note that $R_s$ depends only on the estimated service rate $\mu_s$ and the arrival rate $\lambda_s$. At run-time, ARTE profiles the arrival rate $\lambda_s$ and estimates the service rate $\mu_s$ by using the analytical model in Eq. 4.

Formalization of the run-time management problem. For each application $x$ and each parallelization $\pi$, we define an operating point $v^x_\pi$ as the following tuple:

$$v^x_\pi = (\psi^x_\pi, \mu^x_\pi, \sigma^x_\pi)$$

(9)

Where $\psi^x_\pi$ is the average power consumption of the application $x$, the $\mu^x_\pi$ is the standalone application throughput for the given parallelization $\pi$, and $\sigma^x_\pi$ is the access rate to the shared memory. Each operating point is fully characterized at design time.

We assume that each application $x$ has a set $\Pi_x$ of code-versions [40], each version providing a different parallelization $\pi_x \in \Pi_x$. We consider an operating point $v^x_\pi$ for each code-version. We call $V_x$ the set of all possible operating points for the application $x$. For each parallelization $\pi_x$ of the application $x$, the values of $\psi^x_\pi, \mu^x_\pi$ and $\sigma^x_\pi$ are characterized at design time via simulation.

In the rest of the paper, we will indicate with boldface the vectors that contain all the arrival rates and parallelizations used for each application running on the system ($\lambda$ and $\pi$). As an example, given two permanently running applications in the system ($x_1$ and $x_2$), the vectors:

$$\pi = \begin{bmatrix} \pi_{x_1} \\ \pi_{x_2} \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}, \quad \lambda = \begin{bmatrix} 12.2 \\ 29.8 \end{bmatrix}$$

(10)

indicate that $x_1$ is given 4 processors with estimated job arrival rate equal to 12.2 Job/s while $x_2$ is given 2 processors with an estimated arrival rate of 29.8 Job/s.

The estimated QoS (Eq. 5) is a function of the response time $R_s$ of each application. Please note that we have introduced a set of assumptions under which the $R_s$ depends on $\mu_s$ and on the arrival rates $\lambda_s$. In turn, $\mu_s$ depends on the throughput offered by the current parallelization $\mu^x_\pi$ and on the access rate $\sigma^x_\pi$ associated with the application parallelizations in $\pi$ (Eqs. 3 and 4).

The parameters that can vary at run-time are the arrival rates $\lambda$ and the parallelizations $\pi$. The arrival rates $\lambda$ should be dynamically profiled and the parallelizations $\pi$ should be set by the RRM. We can rewrite the QoS as a function of these parameters:

$$Q \sim Q(\lambda, \pi)$$

(11)

We can formalize the run-time management problem as finding the optimal $\pi$ that maximizes $Q(\lambda, \pi)$, given an instantaneous set of arrival rates $\lambda$ and a set of constraints on the maximum power consumption and the number of processors available. The problem can be stated as follows:

Find $\pi$ such that $Q(\lambda, \pi)$ is maximum,

$$\sum_{x \in A} \psi(v^x_\pi) \leq P,$$

(12)

$$\sum_{x \in A} \pi_x \leq \Lambda$$

(13)

$$\sum_{x \in A} \pi_x \leq \Lambda$$

(14)
Eq. 12 represents a constrained single objective combinatorial optimization problem which is instantiated once the arrival rates $\lambda$ are known. In particular, we assume that $\lambda$ is unknown at design-time and it is inferred at run-time from the user activity. To maximize the QoS, we profile the arrival rates at run-time and we periodically solve the optimization problem across time windows. Given the randomness in the arrival time distribution, a straightforward profiling of $\lambda$ might be subject to some drastic variations among time windows (especially when the window is small). We thus introduced a moving-average filter to smooth this effect.

In the next section, we will describe how the optimization problem above can be efficiently solved at run-time by ARTE.

Run-time resource management heuristic for ARTE. Algorithm 1 shows the resource allocation routine used by ARTE to solve the problem presented in Eq. 12. The algorithm is activated periodically (considerations on the period size will be presented later) in order to fit with the dynamic user activity profiled in terms of average job requests per time unit i.e. $\lambda$. For each application (steps 3–13), an initial minimal parallelization is identified to keep the throughput satisfying the arrival rates $\lambda$. The heuristic then tries to improve the QoS by exploiting additional processing elements and the available power budget (steps 14–21). Finally each application parallelization $\pi_a$ is set at step 22.

Algorithm 1. Resource allocation routine for ARTE

1: for each $a \in A$ do
2: Update $\lambda_a$
3: $\pi_a = 1$
4: while $\mu_a < \lambda_a$ do
5: Increase $\pi_a$ to the next value in ascending order of $\Pi_a$
6: end while
7: end for
8: for each $a \in A$ do
9: Compute $\mu_a$ given the current parallelization vector $\pi$
10: if $\mu_a < \lambda_a$ then
11: Increase $\pi_a$ to the next value in ascending order of $\Pi_a$
12: end if
13: end for
14: while $\exists \exists A, \phi(x)$ do
15: $A_f = \{a \in A | \phi(x)\}$
16: $\pi = \text{argmax}_{a \in A_f} \gamma_a$ 
17: Increase $\pi\pi$ to the next value in ascending order of $\Pi_{\pi}$
18: for each $a \in A$ do
19: Compute $\mu_a$ given the current parallelization vector $\pi$
20: end for
21: end while
22: Load target parallelizations

The run-time heuristic (steps 14–21) is a gradient-based search that iteratively increases the parallelism $\pi$ allocated to the running applications. The search iterates until an application $a$ is found whose parallelization $\pi_a \in \Pi_a$ can be increased without resulting in an unfeasible solution. The predicate $\phi(x)$ verifies that the solution is feasible, i.e., the overall power and resource usage do not exceed neither the power budget $P$ nor the available resources $\Lambda$:

$$\phi(x) = \left[ \left( \psi(v^x_a) + \sum_{b \in A(x)} \psi(v^b_a) \right) < P \right] \cap \left[ \left( \pi_a + \sum_{b \in A(x)} \pi_b \right) < \Lambda \right]$$

The objective function of the search is the gain per unitary resource $\gamma_a$:

$$\gamma_a = \frac{\hat{Q}(\pi_a) - \hat{Q}(\pi_a)}{\pi_a - \pi_a}$$

where $\gamma_a$ quantifies the positive benefits, for application $a$, to switch to a greater parallelization $\pi_a$. Step 14 considers the maximum improvement achievable for any application. The application $\pi$ that offers the maximum improvement is selected (step 16) and its parallelism increased (step 17).

6. Experimental results

In this paper, we address an embedded CMP platform composed of a general purpose processor (host processor) and an application-specific computing fabric consisting of a set of processing elements. The host processor provides flexibility to run...
a common operating system, while the computing elements serve as relatively simple processors to run the parallel applications. The ARTE algorithm is meant to be integrated in the general purpose OS to dispatch application tasks over the available processors of the computing fabric. In this sense, the host processor acts as a Fabric Controller. We consider a computing fabric composed of 16 MIPS-like processors (Fig. 4) with a shared memory architecture.

The computing fabric inter-processor communication is based on an high-bandwidth split transaction bus supporting a write-invalidate snoop-based MESI coherence protocol. Each processor has private L1 instruction and data caches. L2 cache and main memory are shared resources.

To estimate system-level metrics, we leveraged the SESC simulation tool [30], a well-known MIPS instruction set simulator for CMPs providing dynamic power \( \psi_s \) and performance \( \mu_s \) and access statistics to the shared memory \( \sigma_s \) associated to the execution of a job of a user-selected application \( \pi \) with thread-level parallelism \( \pi_s \). Relevant architectural parameters are listed in Table 2.

To provide a fairly broad evaluation of the proposed approach, we leverage the SPLASH-2 parallel benchmark suite [37]. In particular we focused our attention on the following application kernels: Complex 1D FFT (\textit{fft}), Integer Radix Sort (\textit{radix}), Ocean Simulation (\textit{ocean}) and Blocked LU Decomposition (\textit{lu}). For brevity, we will use the symbols \( \alpha_1 \ldots \alpha_4 \) to indicate each application.

Each application has been task-parallelized manually for a number of processors varying as a power of 2 (from 1 to 16) [25].

Evaluation of the run-time model. ARTE is based on the assumption that contention overheads on the shared resources can be approximated by the analytical model presented in Eq. 3. We validated empirically the contention model as follows. First the model is constructed to approximate the contentions measured during an experimental simulation campaign. Then we run a set of additional experiments to collect contention data not included during the model training (we call this set the test set). Finally we evaluate the model error on the test set.

The simulation experiments to generate the contention model are represented by an augmented simplex lattice DoE. We simulate using SESC the execution of each application pair \( (\alpha_i, \alpha_j) \) assigning 50% of the available computing resources (i.e. 8 cores) to each application. In addition, we augmented the simplex lattice DoE with the centroid experiment that represents the concurrent execution of all 4 applications by using 4 cores each.

For the test set of experiments we run 20 different simulations drawing at random the resource distribution among the 4 applications.

Table 3 reports the estimated regression coefficients (Eq. 2) as well as the relative error in the job execution time when using the analytical contention model (Eq. 3). The reference relative error considers no interactions during concurrent application execution.
Overall, the coefficient of determination $R^2$ measured on the estimated contentions is 81.71%. The coefficient of determination represents the variability of the measured contentions that is explained by the analytical model. We considered these results satisfactory for our needs.

Comparison results. The proposed ARTE methodology has been compared against three state-of-the-art approaches:

- The first approach consists of adopting a Reinforcement Learning algorithm (RL) to decide about resource distribution \cite{16,18}. As suggested in \cite{18} we adopted an implementation of the Q-Learning called SARSA. Given the system state, an artificial agent iteratively allocates resources to the applications and collects a reward that represents the inverse of the response time. Doing so the agent learns which are the best resource distributions to use in the different system states. Before carrying out the evaluation and comparing RL performance to the other approaches, a training phase was run for several input sequences.
- The second approach consisted of the maximization of the actual throughput ($\maxT$) as presented in \cite{20}. The heuristic maximizes the actual throughput $\mu = \sum \mu(v^i)$ considering the constraints in terms of total resources and power budget.
- The third approach is derived from the pull-high, push-low (PHPL) heuristic as presented in \cite{11}. Basically, the PHPL policy periodically verifies the power consumption of the different applications and decides how to allocate resources by decreasing the number of allocated processors of application that consume the highest power whenever the power budget is not met.

To evaluate the performance of ARTE, RL, maxT and PHPL, we generated 100 different input sequences of job arrivals by modulating the job arrival rates for each application by using a Markov random process.

For the given input sequences, we investigated the behavior of the four RRM schemes (ARTE, RL, maxT and PHPL) by varying the power budget constraint (from 40 W up to 70 W). Fig. 5 presents the average system performance in terms of response times and queue lengths (average and maximum) by varying the power budget constraint.

We note that ARTE and RL provide better response time than maxT and PHPL approaches (Fig. 5(a)). The response time in reference to maxT is reduced between 1.1% and 68.5% and between 33.2% and 52.4% in reference to PHPL.

---

### Table 3

<table>
<thead>
<tr>
<th>Application</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Relative error [%]</th>
<th>Reference relative error [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>0.143</td>
<td>31.188</td>
<td>−41.999</td>
<td>4.03</td>
<td>9.28</td>
</tr>
<tr>
<td>radix</td>
<td>0.292</td>
<td>27.935</td>
<td>−48.208</td>
<td>8.08</td>
<td>14.39</td>
</tr>
<tr>
<td>ocean</td>
<td>0.449</td>
<td>3.4758</td>
<td>−20.083</td>
<td>4.15</td>
<td>7.19</td>
</tr>
<tr>
<td>lu</td>
<td>−0.898</td>
<td>28.22</td>
<td>−135.38</td>
<td>2.88</td>
<td>2.92</td>
</tr>
</tbody>
</table>

Fig. 5. System performance in terms of response time (a), average (b) and maximum (c) queue length as a function of the power budget constraint.
After the initial learning phase (carried out before the evaluation reported in Fig. 5), RL is capable of providing performance similar to ARTE. We used the Kruskall–Wallis test to verify if the response times of RL and ARTE are statistically different. The results up to 60 W showed that there is no statistical evidence on the fact that RL and ARTE differ in terms of response time. For a power budget larger than 60 W, the analysis is in favor of ARTE showing that the proposed methodology can be considered better also from a statistical point of view.

The differences between ARTE and RL are not in the performance but in their implementation cost and run-time overhead, as well as in their portability to the optimization of different application sets. This will be discussed in detail in Section 7.

It is worth noting that the response time is correlated with the queue length, as can be seen in Fig. 5(b) and (c). ARTE and RL do, in fact, allow for shorter queue lengths with respect to the maxT and PHPL approaches.

We also note that, when the power budget is relatively high, the response time of maxT is very close to the one of ARTE and RL. This happens because a higher power budget can be used to exploit more computational resources. The proposed workload becomes relatively low and fewer jobs are in the system on average. Thus, when a new job arrives it has a higher probability to find an empty queue and can be efficiently allocated using the maxT policy as well.

We might note that maxT allows for performance degradation when increasing the power budget from 40 W to 43 W (and also from 50 W to 53 W, Fig. 5a). This is justified by the fact that maxT tends to create an unbalanced workload by making some low-throughput application starve (as observed in [8,6] and in Section 2).

In Fig. 6 we report relative response time improvement obtained for each application with respect to the reference methodologies. When compared to maxT (Fig. 6(a)), ARTE always provides better performance for a_3 and a_4 and, for low power budget, for a_2 as well. This is because maxT prioritizes the application with the highest throughput that is a_1, letting other applications starve. In reference to PHPL (Fig. 6(b)), ARTE provides, in general, better performance to all applications. Only for few cases ARTE is worse than PHPL with a performance loss lower than 15%. Performance improvement in reference to RL are within ±25% depending on the application and power budget.

RRM behavior. To give more insight about the RRM behavior, we analyzed in detail the execution trace obtained for a specific input sequence when considering a power budget of 60 W. Arrival rates for each application are presented in Fig. 7(a). We analyze the RRM behavior for a variable application mix. In particular a_1 stops its execution at around the 90th second.

For conciseness, only the results for ARTE and RL are shown since maxT and PHPL degrade the overall system performance significantly. Fig. 7(b) shows the average application response time.

For the first 90 s the two policies provide similar response time. Then, when a_1 exits the system, ARTE overcomes the RL policy providing a generally lower response time. This happens because the RL policy has been trained considering all four applications concurrently active. For the last 30 s, RL is in a previously unobserved state and needs to learn how to behave. As discussed in Section 7, it is problematic to use the RL methodology when the application mix can vary. ARTE instead adapts to any application mix composed of design-time characterized applications.

Figs. 7(c) and (d) also show the distribution of system resources over time. Job arrivals have been generated using a Markov process with an average arrival rate reported in Fig. 7(a). Job inter arrival times are not constant but distributed exponentially that leads to actual variations in the profiled arrival rate. Both RL and ARTE switch their operating configurations to adapt with the actual application profile.

RRM overhead. Finally, we evaluated the run-time overheads of the different RRM approaches when executed on the target MIPS based host processor (we assume an operating frequency of 300 MHz). Fig. 8 reports the cumulative overhead resulting from a 2 min system run. The run-time overheads shown in Fig. 8 account for the execution of the RRM routines to decide the application versions to run. Once an operating configuration is selected, the applications might need to switch parallelization. In practice, at the switching point the new application version should be loaded. The overhead introduced by

---

Please cite this article in press as: G. Mariani et al., ARTE: An Application-specific Run-Time management framework for multi-cores based on queuing models, Parallel Comput. (2013), http://dx.doi.org/10.1016/j.parco.2013.04.002
using the Linux operating system for our technology is on the order of magnitude 1–10 ms [24]. The RRM period has been sized to 0.5 s such as to let the run-time overhead related to the operating point switch to be around 2%.

Fig. 8 shows that the lowest overheads of the decision making routines are associated with the PHPL policy and then ARTE, while the highest overheads are associated to the maxT and the RL policies.

The cumulative decision making overhead is distributed almost homogeneously over the considered 2 min. The ARTE, RL and PHPL policies are invoked periodically and their execution overhead does not depend on the number of jobs processed within an RRM period. On average, the overheads introduced within an RRM period are 1.71 µs, 60.6 µs, 1,582 µs and 4,039 µs respectively for PHPL, ARTE, maxT and RL.
The PHPL policy is very fast because it is a straightforward heuristic. An invocation of PHPL simply consists of verifying if
the power budget is exceeded and, in this case, reducing the resources allocated to the most power hungry application. ARTE
is slower than PHPL because it needs to iterate over the analytical models of application response times as presented in Algo-
rithm 1. Nevertheless, ARTE outperforms PHPL in terms of system-level performance (Figs. 5 and 6(b)).

The maxT policy has a higher overhead than ARTE for two reasons. First, as proposed in [20], we use an exhaustive search
for identifying the best resource distributions. Second, maxT policy is invoked every time an application switches between
the active and the idle states. This implies that maxT will be invoked many more times than ARTE, thus the cumulative con-
tention increases.

The overhead related to the RL policy is higher as well. This is due to the complexity of the modeled problem. In fact,
when the RL has to make a decision about resource distribution it interrogates a table of Q values to identify the best action
to be taken. The action should model the resource distribution to the running applications. Being 4 applications with 5 par-
allelization levels we might have $5^4 = 625$ different actions. We optimized the RL to reduce the number of actions by remov-
ing the ones explicitly suboptimal in terms of response time (i.e. the resource distributions that explicitly underutilize the
power budget by using very few resources). Even so, the RL algorithm has still more than a hundred actions to be considered.
RL techniques are general enough to be applied to the RRM problem as demonstrated also in [22,16]. However they have not
been designed specifically for the RRM problem and lack a problem specific optimization.

Differently, ARTE uses an heuristic that we designed specifically for the RRM problem. ARTE uses a greedy search heuristic
(Algorithm 1, steps 14–21) that iteratively increases the allocated resources. ARTE allocates at least 1 core at each iteration of
the greedy search. Since there are 16 cores in the target architecture, ARTE never performs more than 16 iterations, com-
pared to RL that needs to evaluate more than a hundred different actions.

7. Additional considerations

In the aforementioned result comparison, we observed how ARTE and RL overcome traditional techniques such as PHPL
and maxT for the given RRM problem in terms of system performance (Fig. 5). Those two approaches deeply differ in the
assumptions they take about the underlying system and in their portability to different scenarios.

Considerations about the RL policy. To implement the RL algorithm, one has to specify what are the actions that the arti-
ficial agent can take and what are the possible system states. The RL algorithm performance and overheads strongly depends
on these parameters. After trying several implementations, we reported results of the one providing the best performance (in
terms of response time). In this implementation the actions an agent can take are the parallelization of each application.3 The
state describes the power budget, the arrival rates and the state of the most critical application (i.e. the application with the
longest queue). Overall, considering 4 applications the implemented RL has 5152 different states for each power budget and
up to 136 possible actions. The agent should learn a Q value for each state-action combination and store them in a table. It
is impractical to learn these Q values directly at run-time because it implies to try many suboptimal actions resulting in poor
performance [16]. Thus we run a training phase before exploiting the RL policy. For the problem at hand, considering 4 appli-
cations, each with 5 possible parallelization levels and considering 10 possible power budgets, the RL tables storing the Q values
are composed of about 4 M floating point values. For efficiency reasons, different tables of Q values are generated, one for each
possible power budget value. At run-time, given the power budget we load only one of those tables. Nonetheless, the largest
table stores about 700 K Q values.

This large number of actions and states arises when there are multiple applications running concurrently. The action must
represent the resource allocated to each of these applications. Also the state should report information about multiple appli-
cations. In our implementation we report the minimum parallelization required to meet the current arrival rate. The problem
in [18] can be effectively solved at run-time without a training phase because it considers the management of a single appli-
cation leading to significantly smaller numbers of states and actions.

Given that the actions and the states depend on the applications running concurrently, the trained agent in the RL policy
is capable of solving the RRM problem for the application mix used during the training. If the application mix changes at run-
time, different agents have to be saved/loaded for each possible application mix, exponentially increasing the memory
requirements.

The main limitation for the RL technique is the non-portability to different application mix.

Considerations about the proposed ARTE policy. The proposed ARTE approach is easily portable to any application mix. To
extend the approach by including a new application in the target application set, only requires to supply the application
characterization to the RRM.

The application characterization contains few relevant data about each parallel version of an application such as: the re-
source requirement $p_a$, the supported throughput $\mu_a$, the power consumption $\phi_a$ and the access rate to the shared resources
$\sigma_a$. ARTE needs to store this data and elaborate few additional variables. The memory requirements for ARTE are lower than
the ones of RL, where many Q values have to be stored. In particular, for each application version we have to store four val-
ues. Additionally, regression coefficients of the contention model should be stored (i.e. 3 values per application). The memory

---

3 We excluded from our analysis the actions corresponding to resource distributions clearly suboptimal (i.e. the ones that explicitly undergo the power
budget).
requirements scale linearly with the number of applications rather than exponentially as in the case of the RL policy. In particular, considering 4 applications and 5 versions for each one, ARTE requires to store a total of 92 values.

The main limitation of ARTE consists of its assumptions about the queue model in terms of job arrival and execution time distributions. We would like to recall that similar assumptions are not new to our field [33,26]. Authors in [33] suggest to use an M/M/1 queue model (instead of the M/D/1 proposed herein) for characterizing jobs of an MPEG and an MP3 decoder applications. Other authors in [26] use a Pareto distribution of the job inter arrival times for a mobile device running the Android OS.

It is rather easy to extend ARTE to support different queue models and thus to deal with different arrival and service time distributions. In particular, the Eq. 6 used to estimate the expected number of jobs in the system has to be selected in accordance to the queue model.

For M/M/1 models [33] that consider variable execution time, the following equation should be used:

\[
\hat{N} = \frac{\rho}{1 - \rho}
\]  \hspace{1cm} (17)

Additionally, when considering a Pareto distribution of job inter arrival time [26], the following equation can be used:

\[
\hat{N} = \frac{1}{\mu} \times \frac{1}{1 - q}
\]  \hspace{1cm} (18)

where \(q\) is the geometric parameter of the considered Pareto distribution [14].

8. Conclusions

In this paper we proposed an Application-specific Run-Time managEment framework (ARTE) for embedded multi-core systems. To address long-term performance, ARTE maximizes the QoS measured as the inverse of the response time. This is done by exploiting a queuing model that is characterized by application specific information collected during the design-time analysis. The present paper extends the RRM policy presented in our previous work [21] by including an analysis of program interactions on the shared resources and by comparing it to advanced RRM techniques such as reinforcement learning [18].

ARTE outperforms traditional approaches such as maxT and PHPL [20,11] in terms of run-time system performance by achieving a response time reduction of up to 68.5% and 52.4% respectively. A coarse-grained analysis of the run-time overheads has shown that ARTE’s overhead is still acceptable with respect to simpler greedy approaches, which are less effective in terms of long-term performance.

When compared to advanced RL techniques [18,16], ARTE provides very similar run-time behavior with a reduced run-time overhead (Fig. 8). In particular, ARTE decision making overhead for an RRM period is of the order of 60.6 \(\mu\)s in reference to the 4039 \(\mu\)s of RL. Moreover, we discussed the need to train and/or store an artificial agent for the RL approach for each possible application mix. The ARTE approach, on the other hand, is not affected by this limitation.

Acknowledgements

This work was supported in part by the European Community, under Projects FP7-ICT-248716–2PARMA, FP7-ICT-247999-COMPLEX, and ARTEMIS-100230-SMECY, and by the Swiss National Science Foundation under the Grant No. PBTI2-142333.

Appendix A

To let the reader better follow the description of the proposed methodology, Table A.4 groups together the symbols used all along this work and their meanings.
References


