Application-specific multi-core architectures are usually designed by using a configurable platform in which a set of parameters can be tuned to find the best tradeoff in terms of the selected figures of merit (such as energy, delay and area). This multi-objective optimization phase is called design space exploration (DSE). Among the design-time (hardware) configurable parameters we can find the memory sub-system configuration (such as cache size and associativity) and other architectural parameters such as the instruction-level parallelism of the system processors. Among the run-time (software) configurable parameters we can find the degree of task-level parallelism associated with each application running on the platform.

The contribution of this paper is two-fold; first, we introduce an evolutionary (NSGA-II based) methodology for identifying a hardware configuration which is robust with respect to applications and corresponding data-sets. Second, we introduce a novel run-time heuristic that exploits design-time identified operating points to provide guaranteed throughput to each application. Experimental results show that the design-time/run-time combined approach improves the run-time performance of the system with respect to existing reference techniques, while meeting the overall power budget.

This research is partially supported by the Hasler Stiftung under the EMME project (grant no. 11096), and by the European Community under the projects: FP7-ICT-248716-2PARMA, FP7-ICT-247999-COMPLEX and ARTEMIS-100230-SMECY.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee.

© YYYY ACM 1539-9087/YYYY/01-ARTA $10.00 DOI 10.1145/000000.000000 http://doi.acm.org/10.1145/000000.000000
The problem is even worse when considering the run-time behavior of the system concerning the simultaneous execution of multiple applications.

This paper proposes a technique for determining a design-time configuration of the multi-core architecture with increased robustness with respect to applications and corresponding data-sets, i.e., whose performance and power consumption is likely to remain optimal across a range of different workloads. Besides, the design-time exploration is able to generate a set of guidelines (or operating points) that the run-time OS layer will utilize to determine the best parallelism to be used for each running application. In particular, we introduced a novel run-time heuristic to meet the target throughput for each application.

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 ecosystems where a somewhat known set of services or applications is running permanently. This scenario is more realistic than one may 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. In particular, we consider the case in which a sub-set of processors (or cores) can be assigned to a single, multi-threaded application. We exclude from our analysis other parameters such as the frequency and the voltage associated with each system processor [Isci et al. 2006; Bergamaschi et al. 2008] because: a) they are orthogonal to our methodology, b) our focus is on the actual performance and power tradeoffs which can be achieved with pure task-level parallelism.

This work presents an evolution of the approach we proposed in [Mariani et al. 2009]; in particular, we highlight the following contributions:

— We extended the run-time heuristic by decreasing the frequency yet preserving its efficiency.
— We addressed and improved the validation methodology of the Response Surface Models used in the design-time exploration by consolidating the set of statistical tools with a fine grained analysis of the prediction error.
— We provided an accurate analysis of the costs and benefits related to the proposed design-time and run-time methodologies by introducing an estimation of the run-time layer overheads.
— We introduced a case study taken from an automotive cognitive safety system.

The paper is organized as follows. Section 2 presents the state-of-the-art techniques related to the addressed problem. Section 3 provides an overview of the design-time/run-time optimization flow while sections 4 and 5 provide a detailed description of the design-time approach and the run-time approach respectively. Section 6 presents an experimental evaluation of the proposed approach (and a comparison with existing techniques) while Section 7 introduces a case study. Finally Section 8 outlines some conclusions for the work.

2. BACKGROUND

Conventional Design Space Exploration (DSE) is used to tune the system parameters of a platform to achieve a desired power/performance tradeoff. It generally consists of a multi-objective optimization problem whose domain is composed of parameters of a microprocessor-based platform (and all integrated IPs) while the objective functions are, typically, the overall performance and power consumption [Keutzer et al. 2000]. The solution of the optimization problem is a set of candidate points termed Pareto set [Hwang, C. L. and Masud, A. S. M. 1979].

The emphasis on the efficiency of the DSE algorithms is due to the fact that in computer architecture research and development, simulation still represents the main tool to predict performance of alternative architectural design points. If we consider a full cycle-accurate system-level simulation, even the most simple workload requires a significant time to be analyzed, thus a comprehensive exploration of the design alternatives can exceed practical limits [Palermo et al. 2009]. The overall goal is to minimize the number of simulations to be executed during the exploration phase. A common approach to solve this problem is to use Design of Experiments (DoEs) and Response
Surface Modeling (RSM) techniques [Santner et al. 2003] combined with suitable multi-objective minimization or maximization techniques.

The DoE is typically used to identify an initial plan of simulations to provide the designer a coarse grained view of the architectural design space; consolidated approaches are either based on random sampling or more sophisticated techniques like Box-Behnken and Latin Hypercube designs [Palermo et al. 2009; Joseph et al. 2006b]. RSM techniques are usually introduced after the initial experimental design to analyze the system response of unknown configurations without incurring into additional delay due to simulation. They can be used to derive main and interaction effect analysis [Joseph et al. 2006a; Lee and Brooks 2006] or to extend traditional optimization (as in meta-model assisted optimization). In particular, they can be used in iterative refinement approaches [Palermo et al. 2009], as a filtering criterion which excludes from the optimization the worst configurations [Emmerich et al. 2006] and even by identifying the best experiments to be performed to improve the model itself [Knowles 2006].

Analytical RSM models can vary from simple or piecewise linear regression [Joseph et al. 2006a; Lee and Brooks 2006], radial basis functions [Joseph et al. 2006b; Palermo et al. 2009], artificial neural networks [Palermo et al. 2009; Ipek et al. 2006], Kriging interpolation [Mariani et al. 2010] or fuzzy algebra [Ascia et al. 2007]. The selection of a specific type of RSM (model selection) has already been addressed by means of data mining [Li et al. 2009], genetically evolved neural networks [Castillo et al. 2008] and genetically evolved linear models [Cook and Skadron 2008].

Run-time resource management. Design-time decisions might obviously be sub-optimal with respect to the actual deployment scenario. It is a common procedure to adjust the platform resource usage and/or allocation at run-time (such as the platform processors) to meet the actual application needs. Heuristics have been developed to achieve sustained performance by keeping power consumption under a pre-determined constraint [Nollet et al. 2010]. On-line RSMs have been used to improve run-time management by either using piecewise linear regression [Banerjee et al. 2008], artificial neural networks and Q-learning [Martinez and Ipek 2009] or other predictive/adaptive models [Curtis-Maury et al. 2008]. Design-time exploration is increasingly used as an alternative technique to provide simple guidelines for run-time management for small eco-systems [Ykman-Couvreur et al. 2011; Mariani et al. 2010] or to support more complex and versatile composite approaches [Shojaei et al. 2009].

This paper proposes a holistic approach for bridging design-time exploration with run-time resource management of multi-cores architectures. Design-time exploration is performed to converge to a static platform-configuration which is robust with respect to different applications and corresponding data-sets, i.e., whose performance and power are likely optimal over a wide range of run-time scenarios. At the same time, it provides a novel run-time heuristic, strictly driven by the information gathered during design-time exploration, that allows to meet the target throughput for each application.

3. OVERVIEW OF THE OPTIMIZATION FLOW

Customizable multi-processors are widely adopted for application-specific solutions and represent a promising line of investigation. They represent the best compromise in terms of a stable hardware platform which is configurable at design-time while being software programmable (thus upgradable and extensible to some extent).

The customization of next generation multi-processor Systems-on-Chip is problematic due to the following challenges:

--- Problem complexity. The design space associated with future configurable micro-processor IP blocks and the available software-level optimization techniques is huge and it grows exponentially with the degree of parallelism offered by the architecture. In fact, as the technology enables more multi-processing capabilities, manual identification of optimal system configurations becomes unfeasible.

— **Adaptation to dynamic environments.** Workloads of computing platforms are characterized by heavy interaction with the user (and/or the environment). The user can load on the platform different applications and his/her requirements can change dynamically in an unpredictable way. The user activity issues at run-time the execution of some software components and resources should be allocated to guarantee that constraints (either power, temperature or execution time) are met.

The methodology proposed in this paper has two main goals: *a*) finding a robust design-time tradeoff in terms of the system figures of merit (e.g., power consumption and throughput) and *b*) deriving an efficient run-time policy to be integrated into the OS to optimally assign the computing resources (or cores) to run the applications. The overall design-time/run-time optimization flow is shown in Figure 1 and discussed hereafter.

![Design-time/run-time optimization flow](image)

**3.1. Design-time methodology**

The design-time exploration flow takes into account the target applications running on the system. We assume that, although the arrival and departure times of each application are unknown, the complete set of applications is known a-priori (a reasonable assumption for an application-specific processor).

The exploration engine elaborates the architecture template which consists of a set of hardware parameters to be configured at design-time and a set of software parameters which consists of possible resource assignments to each application. In this paper, we will consider as a software parameter the number of homogeneous computing elements \( \pi \) allocated to a single application. The optimization engine generates a final hardware configuration which is robust with respect to the applications and corresponding data-sets. Moreover, it generates, for each application, a set of guidelines or operating points which consists of an estimate of the target throughput and power consumption for each value of \( \pi \). As an example, Figure 2 shows the operating points associated with the task-level parallelism \( \pi \) of an application whose scalability decreases as the task-level parallelism increases (due to the overhead of task synchronization).

**3.2. Run-time methodology**

The proposed run-time methodology introduces a run-time processor assignment policy to maximize the performance while meeting a specified power budget. In our framework, each application execution is considered atomic for a specific data set (or job). The sequence of jobs associated with an application is called *burst*; each burst has an arrival time and a completion time. Although the
task-level parallelization is considered fixed for the execution of a single job, it can be changed between the execution of two distinct jobs. In this paper, we consider code versioning [Ykman-Couvreur et al. 2006] as the main enabling technology to change the parallelization of the application. However, our approach is agnostic with respect to the technology used to parameterize the parallelism. Depending on how the parallelism is exposed, other mechanisms to manipulate it can be considered (e.g. choosing the number of threads at run-time, such as in OpenMP, or by applying specific transformations to the program structure such as in StreamIt. [Gordon et al. 2006; Gordon et al. 2002]).

Figure 3 shows the example behavior of the task-level parallelization chosen by the run-time manager for a scenario with three applications running concurrently. Application \(\alpha_1\) executes a burst of 5 jobs, whose starting time is indicated with a '+', while applications \(\alpha_2\) and \(\alpha_3\) start later. Initially, the run-time manager assigns 8 cores to \(\alpha_1\), reserving 4 cores to \(\alpha_3\) and 2 cores to \(\alpha_2\). However, when \(\alpha_1\)'s burst finishes, the run-time policy assigns 8 cores to \(\alpha_2\) which increases its job throughput, while \(\alpha_3\) remains with the same parallelization. The decision is based on the operating points identified during the design-time exploration and on the current power budget assigned by the user and/or other QoS system policies.

When application \(\alpha\) is processing jobs, its average throughput is defined as:

\[
T_{\alpha}(\pi) = \frac{1}{J_{\alpha}(\pi)}
\]

(1)

where \(\pi\) is the task-level parallelism associated with \(\alpha\) while \(J_{\alpha}(\pi)\) is the average job execution time. The average power consumed by application \(\alpha\) while processing jobs is defined as:

\[
P_{\alpha}(\pi) = \frac{E_{\alpha}(\pi)}{J_{\alpha}(\pi)}
\]

(2)

where \(E_{\alpha}(\pi)\) is the average energy consumption per job of application \(\alpha\).

Equations 1 and 2 represent the average throughput and power consumption of each applications during the processing phase. In some cases, platform applications can be characterized by some idle time between two consecutive job arrivals.
4. DESIGN-TIME DESIGN SPACE EXPLORATION

The proposed design-time methodology aims at finding an optimal hardware system configuration \( \mathbf{x} \) minimizing both the overall power consumption and job execution time of a given set of applications \( \mathcal{A} = \{ \alpha_1 \ldots \alpha_k \} \):

\[
\min_{\mathbf{x}} \Gamma(\mathbf{x}), \quad \Gamma(\mathbf{x}) = \left[ \begin{array}{c} P_\alpha(\mathbf{x}, \pi) \\ J_\alpha(\mathbf{x}, \pi) \end{array} \right], \quad \forall \alpha \in \mathcal{A} \land \forall \pi \in \Pi
\]

where \( \Pi \) is the possible set of task-level parallelizing options. Generally, finding a unique configuration \( \mathbf{x} \) to solve this minimization problem is not possible since power consumption and execution time are two contradictory objectives. The problem is solved in two consecutive steps. First a multi-objective optimization phase identifies a set of Pareto optimal configurations representing the best tradeoffs in terms of the selected figures of merit (see Section 4.2). Then, a selection criterion is used to choose the final configuration to implement (see Section 4.3).

Considering the problem defined in Equation 3, whenever we fix a specific application \( \alpha \) and architectural configuration \( \mathbf{x} \), the remaining free variable \( \pi \) is the only parameter that can change the power \( P_\alpha(\mathbf{x}, \pi) \) and latency \( J_\alpha(\mathbf{x}, \pi) \) associated with the program execution. Figure 2 may be thought of as an example of power consumption \( P_\alpha(\mathbf{x}, \pi) \) and throughput \( 1/(J_\alpha(\mathbf{x}, \pi)) \) variation due to the parallelization chosen for a fixed application \( \alpha \) and a fixed architectural configuration \( \mathbf{x} \).

According to [Kugele et al. 2007], the problem expressed by Equation 3 is generally unsolvable and it is negotiated with less constrained problems such as the robust counterpart problem or the minimization of the average value of \( \Gamma(\mathbf{x}) \) [Palermo et al. 2008]. In this paper, we will follow the second option, by re-formulating Equation 3 into the following robust architectural optimization problem: find the optimal design-time configuration and application parallelization averaged over each single scenario of uncertainty \( \alpha \):

\[
\min_{\mathbf{x}, \pi} \Omega(\mathbf{x}, \pi), \quad \Omega(\mathbf{x}, \pi) = \left[ \begin{array}{c} P_\ast(\mathbf{x}, \pi) \\ J_\ast(\mathbf{x}, \pi) \end{array} \right]
\]

where \( P_\ast(\mathbf{x}, \pi) \) and \( J_\ast(\mathbf{x}, \pi) \) are the geometric averages of the power consumption and the job execution time over the application set \( \mathcal{A} \). Formulating the problem as a geometric average can guarantee robustness with respect to different applications (and data-sets) since the goal is to minimize the average loss (e.g. \( P_\ast(\mathbf{x}, \pi) \)) instead of considering separately each single scenario \( \alpha \in \mathcal{A} \) (e.g. \( P_\alpha(\mathbf{x}, \pi) \)), see for example [Kugele et al. 2007].

To solve such a multi-objective problem, we propose an exploration methodology based on the NSGA-II evolutionary strategy [Deb et al. 2002]. The strategy applies operators such as crossover and mutation to process a population of configurations by evaluating the fitness of the associated chromosome \( (\mathbf{x}, \pi) \).

The chromosome structure used in this paper is shown in Figure 4. The first part is dedicated to represent the hardware configuration \( \mathbf{x} \), i.e., the parameters that should be statically tuned (e.g. size and associativity of the cache memory and/or type and number of processors to be integrated on the multi-core platform). Each hardware parameter is encoded with an integer level which is associated with feasible configurations. The second part of the chromosome is dedicated to express the task-level parallelism associated with the applications by using an integer number.

The main difficulties encountered when using the NSGA-II algorithm is the very long simulation time required to evaluate the system-level objective function \( \Omega(\mathbf{x}, \pi) \) and, in turn, the fitness of each chromosome since it depends on actual simulations of the target architecture.

![Fig. 4. Chromosome structure used for the NSGA-II algorithm.](image-url)
For this reason, we introduce an analytic meta-model for approximating the target objective function:

\[ \hat{\Omega}(x, \pi) \sim \Omega(x, \pi) \]  

such that the components of \( \hat{\Omega}(x, \pi) \) (i.e., \( \hat{P}_s(x, \pi) \) and \( \hat{J}_s(x, \pi) \)) are efficiently evaluated and represent a reasonable approximation with respect to the metrics measured by an actual simulation of the system. The model is constructed and updated with simulation data collected during optimization and it is a very effective tool for analytically predicting the behavior of the system platform without resorting to a system simulation. In principle, the analytical model might be implemented using linear regression techniques [Joseph et al. 2006a; Lee and Brooks 2006]. However, different research works have demonstrated that Artificial Neural Networks (ANNs) are more suitable to model the complex behavior of multi-core computing architectures [Palermo et al. 2009; Ozisikyilmaz et al. 2008].

In order to choose which model should be used (either the simulator or the ANN) during the fitness evaluation, we introduce an evolution control strategy. In particular, we leverage the results from statistical modeling of platform-based architectures, by identifying the uncertainty estimation of architectural configurations. The goal is to simulate only those configurations that are promising from the point of view of either execution time or power consumption.

In the following, we will describe in detail the ANN based model and the proposed evolution control strategy.

### 4.1. Execution time and power models

ANNs represent a powerful and flexible method for generalized response surface modeling [Bishop 2002]. The representation power of ANNs is rich enough to express complex interactions among variables; any function can be approximated with an arbitrary precision by a three-layer ANN like the one proposed in Figure 5. In this configuration, the neural network consists of an input, an output and a hidden layers. Input values (in our case the hardware and software parameters in the chromosome) are presented at the input layer; predictions are obtained from the output layer.

![Fig. 5. A fully connected, feed-forward ANN with one hidden layer, three input parameters and output variable \( y \).](image)

The ANN model used in this paper is a scalar approximation function. Thus, we need two neural networks, one for predicting \( P_s(x, \pi) \) and one for predicting \( J_s(x, \pi) \). Since, in this description, the same considerations apply to both \( P_s(x, \pi) \) and \( J_s(x, \pi) \), we will use the symbol \( 'y' \) as a synonym for the two figures of merit.

The ANN model we adopt is the three-layers perceptron network, i.e. a fully connected feed-forward ANN. In this ANN, weighted edges connect every unit in each layer to all units in the next layer, communicating the outputs to next units. In practice, both hardware and software parameters \((x, \pi)\) appearing in the input layer are weighted and passed to the hidden layer. In the hidden layer, every unit transforms the input data by using a sigmoid function. Finally the network output \( \hat{y} \) (i.e.,
the predicted figure of merit) is computed as a linear combination of the different hidden layer outputs.

The ANN is treated as a function of the input configuration \((x, \pi)\) and it is parameterized by using a vector of the weights of the links connecting the neurons. Generally, the parameter vector is fixed during a training phase by using a set of training observations in order to minimize the prediction error \(\epsilon\) on the observed data.

In this paper, we use a back-propagation algorithm to perform a least square minimization of the error \(\epsilon\) by iterating over the set of observed simulation results (we call this set the learning set). To avoid over-fitting (i.e., lowering the generalization power of the model), we apply the early stopping criterion [Masters 1993]. In particular, we use 20% of the learning set as a validation set and we iterate the learning algorithm over the remaining set (that we call training set) as long as the error on the validation set decreases.

4.2. Evolutionary control based on prediction confidence interval

As outlined before, the adopted NSGA-II has been modified to be assisted by the ANN during the evaluation of the fitness of each configuration \((x, \pi)\) (see Figure 6 for a block diagram of the optimization flow). Our NSGA-II works by generating a set of candidate solutions \(X = \{(x_1, \pi_1), \ldots, (x_n, \pi_n)\}\) from the current population by using a permutation and cross-over block (named GEN). The set of configurations \(X\) is then fed to the neural network to create a prediction of the objective function \(\hat{\Omega}(x, \pi)\). The output of the neural network drives a refinement block that selects the configurations that should be refined with the simulator (to derive the actual value of \(\Omega(x, \pi)\)). The refinement policy is called evolution control strategy.

\[
X_S = \eta(X), \quad X_{NS} = X - X_S
\]

At the end of the simulations, both \(\Omega(X_S)\) and \(\hat{\Omega}(X_{NS})\) are fed to the selection block of the NSGA-II to get the next generation. Besides, \(\Omega(X_S)\) is used for improving, by means of an on-line training algorithm, the prediction accuracy of the ANN.

Fig. 6. The proposed ANN model-assisted NSGA-II exploration algorithm (Safe Dominance Control Strategy, SDCS).
Typically, IPE filters select a fixed percentage of $X$, by giving priority to the configurations $(x, \pi)$ which can improve the identified Pareto front considering only the prediction mean. This approach represents the state of the art of model based optimization [Jin et al. 2000; 2001] and it is known as Best Control Strategy (BCS). The characteristic function:

$$\rho_{BCS} = \frac{\eta(X)}{|X|}$$

is known as the permeability of the BCS filter.

A problem of traditional filtering is that permeability $\rho$ is not dependent on any particular feature of the population, being user-selected. This characteristic can introduce some additional simulation-time and/or sub-optimality if the permeability is not correctly set by the user.

In this paper, we propose to avoid this problem by introducing an alternative IPE filter $\eta$ which exploits the information on the prediction confidence interval of each configuration $(x, \pi) \in X$. Basically, if the ANN prediction $\hat{\Omega}(x, \pi)$ is such that $(x, \pi)$ is Pareto dominated within a reasonable prediction range by another configuration, the filter excludes it from the configurations to be simulated. We call this approach Safe Dominance Control Strategy (SDCS). The proposed filtering, being based on specific properties of meta-model used, promises to reduce the problems associated with traditional filtering such as BCS discussed above.

The prediction confidence interval is a range around the prediction $\hat{\Omega}(x, \pi)$ for an unobserved configuration $(x, \pi)$. The higher the uncertainty $\sigma^2$ related to the ANN models used to predict the target metrics, larger the interval is.

A confidence parameter $\chi$ characterizes the confidence interval around $\hat{\Omega}(x, \pi)$ indicating the probability that the value $\Omega(x, \pi)$ falls within it. Confidence intervals can be computed for any desired confidence $\chi$ assuming that $\epsilon/\sigma$ follows a normal distribution with zero mean and unitary variance.

Considering a multi-objective minimization problem, let us define a best case $b_\chi(x, \pi)$ and a worst case $w_\chi(x, \pi)$ which are respectively the lower and upper bound vector values of the prediction confidence intervals of all objectives for the configuration $(x, \pi)$.

In the proposed SDCS strategy, we define the operator $\eta$ such that

$$X_S = \eta(X) = \{ x \in X \mid \exists x' \in X \land w_\chi(x', \pi') \prec b_\chi(x, \pi) \}$$

where the operator $\prec$ denotes Pareto dominance.

In practice, the SDCS strategy simulates only the points for which the best case is not Pareto dominated by the worst case of any other point, given a confidence value of $\chi$. Figure 7(a) shows the case when configuration $c_1$ is not simulated, having its best case $b_1$ dominated by the worst case $w_2$ of configuration $c_2$ (the prediction interval is indicated as a box around the values of $P_\ast(x, \pi)$.)

Fig. 7. System configurations and prediction intervals. On the left, according to the proposed SDCS strategy, configuration $c_1$ is not simulated. However, the case on the right forces configuration $c_1$ to be simulated.
and $J_\alpha(x, \pi)$ objective functions). Figure 7(b) shows the case when configuration $c_1$ is chosen to be simulated because its best case is not dominated by the worst case of configuration $c_2$. Finally, it is worth noting that the main difference between BCS and SDCS is that BCS represents the case when no confidence interval is used.

4.3. Selection of the final hardware configuration

The result of the NSGA-II based strategy is a Pareto set $X$ of configurations $(\bar{x}, \bar{\pi})$. Each configuration $(\bar{x}, \bar{\pi})$ is associated with a different tradeoff between execution time $J_\alpha$ and power consumption $P_\alpha$. However, from the multitude of candidate hardware configurations $X$ we need to derive a single, final configuration $\bar{x}$ to be implemented. Then we measure, for each $\bar{x} \in X$, both execution time $\bar{J}_\alpha$ and power consumption $\bar{P}_\alpha$ associated with all the available parallelizations $\pi \in \Pi$.

To select the final hardware configuration, we rank the quality of the solution $\bar{x}$ by using a geometric average:

$$Q(\bar{x}) = \prod_{\forall \pi} J_\alpha(\bar{x}, \pi) \times P_\alpha(\bar{x}, \pi)$$

(9)

The ranking function $Q(\bar{x})$ introduces a unique ordering among the static hardware solutions, by favoring those showing the best power-delay tradeoffs among all the possible parallelizations. We then select, as the final static configuration $x_\alpha$, the one which minimizes $Q(\bar{x})$.

The above delay-power product metric is used to identify the best candidate hardware configuration. Once this is determined, run-time operating points can be derived to characterize the throughput and power for each parallelization by using simulations. This information enables the throughput oriented optimization at run-time.

In particular, once the hardware configuration $x_\alpha$ is chosen, the data associated with the operating points are made accessible at run-time in a configuration file read by the run-time resource manager. Data include the throughput measure $T_\alpha = 1/\bar{J}_\alpha(x_\alpha, \pi)$ and the power consumption $P_\alpha(x_\alpha, \pi)$ related to each application $\alpha$ for all parallelizations $\pi \in \Pi$.

4.4. A practical example

As a practical example, let us consider a 4-core, configurable architecture whose design space is composed of the following parameters: $a)$ issue width (IW, with range 1 - 2), and, $b)$ data cache size (DCS, with range 1KB - 2KB). A generic hardware configuration $x$ is thus represented by the parameters IW and DCS (overall 2 $\times$ 2 combinations) while the parallelism $\pi$ for each application can vary within the range \{1,2,4\}. For simplicity, we assume to have a single application, $\alpha_1$, running on the architecture. Thus, in this case, $J_\alpha(x, \pi)$ and $P_\alpha(x, \pi)$ represent the latency and power of the system estimated by simulating $\alpha_1$ ($J_\alpha = J_{\alpha_1}$ and $P_\alpha = P_{\alpha_1}$).

Let us assume that our SDCS-based NSGA-II algorithm identified the set $X$ of Pareto hardware configurations shown in Table I. To identify the final configuration $x_\alpha$, our design time algorithm proceeds as follows:

— It completes Table I in order to cover all the possible parallelizations $\pi$, for each identified unique combination of DCS and IW. The complete list of configurations is shown in Table II.

— It derives, for each hardware configuration, the value of the rank index $Q$ from both $J_\alpha$ and $P_\alpha$ (we recall that a single value of $Q$ exists for each static configuration, given that it is the grand product over all the possible parallelizations).

— It selects, as a candidate hardware configuration $x_\alpha$, the configuration of DCS and IW with the lowest rank $Q$ (shown in bold font in Table II)

The final list of operating points for $\alpha_1$ is shown in Table III; it is derived directly from the configuration $x_\alpha$ marked in bold in Table II. This list is then used by the run-time manager to select the appropriate parallelization for $\alpha_1$ at run-time, given the requirements on its throughput. In multi-
Table I. Example output of the NSGA-II algorithm

<table>
<thead>
<tr>
<th>IW</th>
<th>DCS</th>
<th>π</th>
<th>J ∗ (s/job)</th>
<th>P ∗ (W)</th>
<th>T ∗ (job/s)</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1KB</td>
<td>1</td>
<td>0.33</td>
<td>0.5</td>
<td>3.03</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1KB</td>
<td>4</td>
<td>0.14</td>
<td>3.0</td>
<td>7.14</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1KB</td>
<td>1</td>
<td>0.31</td>
<td>0.6</td>
<td>3.23</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1KB</td>
<td>2</td>
<td>0.18</td>
<td>1.1</td>
<td>5.56</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>2KB</td>
<td>2</td>
<td>0.17</td>
<td>2.2</td>
<td>5.88</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>2KB</td>
<td>4</td>
<td>0.11</td>
<td>4.0</td>
<td>9.09</td>
<td>-</td>
</tr>
</tbody>
</table>

Table II. Selection of the final hardware configuration x∗ (bold)

<table>
<thead>
<tr>
<th>IW</th>
<th>DCS</th>
<th>π</th>
<th>J ∗ (s/job)</th>
<th>P ∗ (W)</th>
<th>T ∗ (job/s)</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1KB</td>
<td>1</td>
<td>0.33</td>
<td>0.5</td>
<td>3.03</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1KB</td>
<td>2</td>
<td>0.20</td>
<td>1.0</td>
<td>5.00</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1KB</td>
<td>4</td>
<td>0.14</td>
<td>3.0</td>
<td>7.14</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1KB</td>
<td>1</td>
<td>0.31</td>
<td>0.6</td>
<td>3.23</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1KB</td>
<td>2</td>
<td>0.18</td>
<td>1.1</td>
<td>5.56</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1KB</td>
<td>4</td>
<td>0.12</td>
<td>3.3</td>
<td>8.33</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>2KB</td>
<td>1</td>
<td>0.28</td>
<td>1.0</td>
<td>3.57</td>
<td>0.0139</td>
</tr>
<tr>
<td>2</td>
<td>2KB</td>
<td>2</td>
<td>0.17</td>
<td>2.2</td>
<td>5.88</td>
<td>0.0461</td>
</tr>
<tr>
<td>2</td>
<td>2KB</td>
<td>4</td>
<td>0.11</td>
<td>4.0</td>
<td>9.09</td>
<td>-</td>
</tr>
</tbody>
</table>

Table III. List of operating points for α

<table>
<thead>
<tr>
<th>π</th>
<th>J α (s/job)</th>
<th>P α (W)</th>
<th>T α (job/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.33</td>
<td>0.5</td>
<td>3.03</td>
</tr>
<tr>
<td>2</td>
<td>0.20</td>
<td>1.0</td>
<td>5.00</td>
</tr>
<tr>
<td>4</td>
<td>0.14</td>
<td>3.0</td>
<td>7.14</td>
</tr>
</tbody>
</table>

application scenarios, the algorithm generates a list of operating points for each application running on the eco-system.

5. RUN-TIME RESOURCE MANAGEMENT: THE MAXTR POLICY

In our typical run-time scenario, the user can dynamically require the execution of different applications. We assume that, for each active application, the number of jobs to be served is dynamically dependent either on the activity of the user interacting with the platform or on events/conditions coming from the environment. Our RRM dynamically identifies a distribution of the available cores to the active applications such that a given power budget is met and the system performance is maximized. We are assuming that the power budget is assigned by other layers of the underlying operating system by observing, for example, the actual battery charge level. The MAXTR run-time management policy proposed in this paper represents an extension of the MAXT policy [Mariani et al. 2009] that allows managing constraints on the individual application throughput on a window-based, periodic basis. This extension allows to limit the consumption of memory resources allocated to the application job queues by accounting for each application need and to improve the response time.

The basic idea of the MAXTR policy is that resource allocation is done on a periodic-basis, i.e., allocated resources are reserved (MAXTR = MAXT with resource Reservation) to their applications within a window of size τ and they are not reallocated to other applications within window τ. This may imply: a) invoking fewer times the RRM policy and thus significantly lowering the overall overhead and b) serving with a potentially reduced overhead the jobs bursts that are adjacent in time.

In this context, we assume that even if job arrivals are unknown at design-time they can be monitored at run-time. Under this assumption, we profile dynamically the job arrival rates λ_i(n) for each
application $i$ within a given time window $n$ and we map processors to the applications for the next window $n+1$ based on the assumption $\hat{\lambda}_i(n+1) = \lambda_i(n)$.

More formally, being $\langle \lambda_1...\lambda_k \rangle$ the arrival rates measured for the $k$ applications during the period $\tau$ associated with window $n$, the parallelism for each application $\pi_\alpha_i$, for the window $n+1$ is selected in order to maximize the overall throughput $T_O$:

$$T_O(\pi_{\alpha_1}, \ldots, \pi_{\alpha_k}) = T_{\alpha_1}(x_o, \pi_{\alpha_1}) + \ldots + T_{\alpha_k}(x_o, \pi_{\alpha_k})$$  \hspace{1cm} (10)

by constraining the individual throughputs:

$$\forall i, \ T_{\alpha_i}(x_o, \pi_{\alpha_i}) \geq \hat{\lambda}_i(n+1)$$  \hspace{1cm} (11)

and

$$\sum_i P_{\alpha_i}(x_o, \pi) \leq P_B.$$  \hspace{1cm} (12)

where $T_{\alpha_i}$ and $P_{\alpha_i}$ are the operating points derived at design-time while $P_B$ is the target power budget.

The above constraint on throughput ensures that the parallelism $\pi_{\alpha_i}$ of each application $\alpha_i$ is chosen such that the arrival rate $\lambda_i(n+1)$ is adequately served, with evident impact on the response time of the applications. Additionally, the search space is reduced by quickly removing the operating configurations whose throughput $T_{\alpha_i}$ is not adequate to serve the arrival rate $\lambda_i(n+1)$. The reduction of the search-space improves the run-time selection overhead.

To give a practical example, let us consider two applications $\alpha_1$ and $\alpha_2$ characterized by the list of operating points shown in Table III. Let us assume that the arrival rates are $\hat{\lambda}_1 = 1.5$ (job/s) and $\hat{\lambda}_2 = 3.5$ (job/s). In this case, the search space is composed of 3 operating points ($\alpha_1$) times 2 operating points ($\alpha_2$); in fact, parallelization 1 is automatically excluded for $\alpha_2$ given the predicted throughput in the list of operating points, while $\alpha_1$ can exploit all the parallelizations available since they all adequately serve the arrival rate. Thus, we have 6 combinations to find the maximum throughput instead of the full list of 9 combinations (33% reduction of the search space). The advantage of operating point reduction may grow with the number of applications running.

Similarly to \textsc{maxt}, after selecting the parallelism for each application $\alpha_i$, the RRM communicates the desired parallelization level to each application. Applications independently perform the operating points switch once the execution of the job currently processed is completed. The switch is allowed only if the power budget and resource constraints are met. Otherwise, the applications switch to the nearest operating point such that the previous constraints are met.

6. EXPERIMENTAL RESULTS

In this paper, we address the design optimization of an embedded multi-core platform composed of a general purpose processor (\textit{host} processor) and an application-specific computing fabric (consisting of a homogeneous set of \textit{computing elements}). The general purpose processor provides flexibility to run a relatively common Operating System (OS), while the other computing elements serve as relatively simple (with few or none superscalar features) processors to run the tasks of the target applications. The proposed RRM policy 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 \textit{host} processor acts as a Fabric Controller. We consider here a computing fabric which is composed of 16 MIPS-like processors (Figure 8) with a centralized shared memory architecture but the approach is general enough to be applied to a relatively large computing fabric with a distributed shared memory configuration.

The computing fabric inter-processor communication is based on a high-bandwidth split transaction bus supporting a write-invalidate write-back snoop-based MESI coherence protocol acting
directly between L2 caches. To ensure the coherency of shared data in the memory hierarchy, this protocol generates invalidate/write requests between L1 and L2 caches.

To estimate system-level metrics, we leveraged the SESC simulation tool [Renau et al. 2005], a fast MIPS instruction set simulator for CMPs providing dynamic power $P_{\alpha}(x, \pi)$ and execution time $J_{\alpha}(x, \pi)$ associated to the processing of a job of a user-selected application. To evaluate the system metrics we consider an operating frequency of 300MHz.

To provide a fairly broad evaluation of the proposed approach, we leverage the SPLASH-2 parallel benchmark suite [Woo et al. 1995]. In particular we focus our attention on the following application kernels: Complex 1D FFT ($\text{fft}$), Integer Radix Sort ($\text{radix}$), Ocean Simulation ($\text{ocean}$) and Blocked LU Decomposition ($\text{lu}$). For brevity, we will use the symbols $\alpha_1...\alpha_4$ to indicate each application.

The hardware parameters considered during the design-time exploration are the cache sizes and associativities as well as the instruction issue width of the processors (the range of these parameters are reported in Table IV). The run-time tunable parameter is the task-level parallelization of the applications. Each application has been task-parallelized manually [Mignolet et al. 2009] for a number (power of 2, from 1 to 16) of processors.

The chromosome structure adopted during the NSGA-II based exploration encodes both hardware and software parameters $(x, \pi)$. The overall design space explored by the proposed design-time methodology is composed of $2^{15} \times 5$ points, where $2^{15}$ and 5 respectively represent the total number of hardware and software configurations.

To evaluate the benefits of the proposed design-time methodology, we compared the proposed SDCS-based NSGA-II with a conventional NSGA-II and a BCS-based NSGA-II. Since both BCS and SDCS are based on the selected model of ANN (Section 4.1), we present some experimental evidence regarding its validation for the considered use case.

### Table IV. Range of hardware design parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Issue Width</td>
<td>1 - 8</td>
</tr>
<tr>
<td>L1 Instruction Cache Size</td>
<td>2KB - 16KB</td>
</tr>
<tr>
<td>L1 Data Cache Size</td>
<td>2KB - 16KB</td>
</tr>
<tr>
<td>L2 Private Cache Size</td>
<td>32KB - 256KB</td>
</tr>
<tr>
<td>L1 Instruction Cache Associativity</td>
<td>1w - 8w</td>
</tr>
<tr>
<td>L1 Data Cache Associativity</td>
<td>1w - 8w</td>
</tr>
<tr>
<td>L2 Private Cache Associativity</td>
<td>1w - 8w</td>
</tr>
<tr>
<td>I/D/L2 Block Size</td>
<td>16B - 32B</td>
</tr>
</tbody>
</table>

The chromosome structure adopted during the NSGA-II based exploration encodes both hardware and software parameters $(x, \pi)$. The overall design space explored by the proposed design-time methodology is composed of $2^{15} \times 5$ points, where $2^{15}$ and 5 respectively represent the total number of hardware and software configurations.

To evaluate the benefits of the proposed design-time methodology, we compared the proposed SDCS-based NSGA-II with a conventional NSGA-II and a BCS-based NSGA-II. Since both BCS and SDCS are based on the selected model of ANN (Section 4.1), we present some experimental evidence regarding its validation for the considered use case.
6.1. Validation of the neural network model

Common neural modeling practices [Blum 1992] suggest to use a number of hidden units which is in the range of input and output parameters of our target design space. So we selected a configuration of the network whose hidden layer size is almost half of the architecture parameters (i.e., 5).

To assess the quality of the neural model, we analyzed the following properties:

— Mean relative error $|y - \hat{y}|/y$ of the trained network for the learning set.
— Mean relative error $|y - \hat{y}|/y$ of the trained network for a set of design configurations not involved in the learning phase. We call test set this set of configurations.
— Normality of the residuals $\epsilon$ for the training set. This is evaluated by comparing the distribution of the standardized residuals (SR) $\epsilon/\hat{\sigma}^2$ with an ideal normal distribution by means of a Quantile-Quantile (QQ) plot. The rationale behind this kind of plot is that, by construction, all the observations with a normal distribution are likely to be laid on a straight line.

Fig. 9. QQ plots of the standardized residuals for the LU decomposition benchmark.
To evaluate the mean relative error and to generate the QQ plots, for each of the available benchmarks, the following steps have been considered:

— Run a set of simulations corresponding to 0.1% of the entire design space (this is the *learning set*).
— Run a second set of simulations non overlapping with the first set and corresponding to 0.1% of the entire design space (this set is used as *test set*).
— Pre-processing data with a Box-Cox transform with $\lambda = 0$; this step enabled us to reduce the variance of the observations and to improve the model accuracy [Palermo et al. 2009].
— Training the ANN model with the collected data by using the back-propagation algorithm.
— Evaluate the residuals $\epsilon$ for the *learning* and the *test sets*.

Considering the *learning set*, the neural model presented a mean relative error of 4% on energy consumption $P_\ast(x, \pi)$ and 8.5% on execution time $J_\ast(x, \pi)$. For the *test set*, the mean relative error is of 14.4% and 15.8% for the energy consumption and execution time respectively. Figures 9(a) and 9(b) show the distribution of the standardized residuals respectively for the execution time and the energy consumption for the *LU decomposition* benchmark (one of the most used in digital signal processing systems). As can be seen, the residuals of the trained ANN model are well approximated by an almost straight line, thus confirming the hypothesis of normal distribution. This behavior has also been observed for the remaining benchmarks in the selected set. Therefore we can deduce that the selected ANN model represents a suitable model for our methodology.

### 6.2. Evaluation of the design-time methodology

Our design-time experimental campaign has the following goals:

— Identify the best values of the IPE filter parameters $\chi$ and $\rho_{BCS}$ which maximize respectively the performance of SDCS-based and BCS-based optimization heuristics.
— Compare the proposed SDCS-NSGA-II with a conventional NSGA-II and a BCS-NSGA-II.

For the sake of clarity, since all the heuristics are NSGA-II based, in the remaining part of the paper we will use the terms SDCS for SDCS-NSGA-II and a BCS-NSGA-II respectively.

In our experiments, the initial DoE is characterized by 50 randomly sampled configurations $(x, \pi)$. The genetic algorithms have been iterated for 100 generations. Concerning SDCS and BCS heuristics, the DoE is evaluated via simulations and the results are used to construct the first ANN models. The ANN models are then retrained every 50 simulation samples.

In this paper, we will use the *Average Distance from Reference Set* (ADRS) [Czyzak P. 1998] as a measure of the distance of the heuristic Pareto front with respect to the exact Pareto front. The ADRS is usually measured in terms of percentage; the higher the ADRS, the worst is the heuristic solution.

![Comparisons of SDCS and BCS in terms of simulations required to obtain the desired ADRS value of 2.5%](image-url)

**Fig. 10.** Comparisons of SDCS and BCS in terms of simulations required to obtain the desired ADRS value of 2.5% by varying the permeability for BCS and the confidence for SDCS.
Given that, the optimization results are dependent on the initial random DoE and also considering the randomness introduced by the genetic operators, all heuristics have been executed repeatedly until the observed ADRS for each heuristic is stabilized. The tuning of parameters $\chi$ and $\rho_{BCS}$ has been carried out respectively for the SDCS and the BCS to minimize the number of simulations required to obtain a desired ADRS. In particular, the target ADRS value is 2.5% (same order of magnitude of those used by [Palermo et al. 2009]). The considered values for $\rho_{BCS}$ are 50%, 40% and 30% [Jin et al. 2001], while the considered confidence values $\chi$ are 75%, 50%, 25%.

Figure 10 shows results in terms of the overall number of simulations required to reach the desired ADRS. The simulation count for the BCS algorithm is minimized to 246 design configurations when $\rho = 40\%$. The best value of the confidence parameter $\chi$ for the SDCS algorithm is $\chi = 50\%$, which minimizes the simulation count to a similar value (i.e. 244).

Even if the algorithms have similar performance when targeting an ADRS of 2.5%, the SDCS permeability is adapted to the uncertainty of the ANN models generation after generation. At the beginning of the optimization, the ANNs are characterized by a high uncertainty, the confidence intervals are large and hence more design configurations are simulated. This phenomenon is reported in Figure 11, which shows the trend of SDCS permeability for a specific algorithm execution with $\chi = 50\%$.

The SDCS policy, during the initial phases, runs more simulations than the BCS (permeability above the 40%). Indeed, these simulations are used to improve the ANN quality. The SDCS permeability generally decreases after this initial phase since the model uncertainty decreases.

![Fig. 11. IPE filter permeability for an SDCS execution with $\chi = 50\%$.](image)

![Fig. 12. Mean Relative Error (MRE) of the ANN models when samples are taken from the design space exploration. The learning set size is expressed as a percentage of the whole design space.](image)
Figure 12 shows the mean relative error of the considered ANN, for both the execution time and power consumption predictions.

For estimating the overall optimization speed up, SDCS and BCS are compared with the conventional version of the NSGA-II. To complete the validation, we vary the value of the desired ADRS and we run both SDCS and BCS until the desired ADRS are obtained (we can compute the actual ADRS since we have performed an off-line full-search analysis of the entire design space).

As reported in Figure 13(a), the simulation time savings of BCS with respect to the NSGA-II execution is in the range from 19.1% to 31.8%, while for SDCS savings are from 25.1% up to 32.3%.

When compared to the conventional NSGA-II, the execution time of SDCS and BCS heuristics include the overhead related to the use of ANNs. However, Figure 13(a) considers only the time savings related to actual simulations since it represents the largest part of the overall execution time of the algorithms. Our Matlab implementation of SDCS introduces a 1.2 seconds overhead for each generation of the underlying evolutionary algorithm. Part of this overhead (60%) is due to the back-propagation algorithm used to train the ANN (10 inputs, 5 hidden units and 1 output unit). ANN training is done by using the early stopping criterion by limiting the number of training iterations to 200. This setting provides a reasonable tradeoff between training time and model quality for the target applications.

To conclude with the analysis of advantages and disadvantages resulting from a meta-model assisted optimization, Figure 13(b) shows the probability of identifying a solution with a given ADRS at the end of the optimization. Typically, introducing analytical models in the exploration flow is problematic since they might drive the optimization towards false optima (i.e., solutions that are optimal for the analytical model but not for the actual system response). However, results in Figure 13(b) show that for both model assisted heuristics (BCS and SDCS) the desired accuracy (ADRS) is obtained more than 97.5% of the times. This confirms our assumptions that, even though models introduce approximations, these are negligible for the sake of the final optimization results.

6.3. Selection of the robust candidate configuration

To select a hardware configuration for the target platform, we apply the procedure presented in Section 4.3. We focus on the exploration results obtained with SDCS. The selected best architecture configuration is reported in Table V.

Once the hardware configuration is selected, we save the operating points for each application $\alpha_1...4$ which are shown in Figure 14.
Fig. 14. Application operating points found with the proposed methodology.

Table V. Best architecture configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Issue Width</td>
<td>1</td>
</tr>
<tr>
<td>L1 Instruction Cache Size</td>
<td>16KB</td>
</tr>
<tr>
<td>L1 Data Cache Size</td>
<td>8KB</td>
</tr>
<tr>
<td>L2 Private Cache Size</td>
<td>256KB</td>
</tr>
<tr>
<td>L1 Instruction Cache Associativity</td>
<td>1w</td>
</tr>
<tr>
<td>L1 Data Cache Associativity</td>
<td>8w</td>
</tr>
<tr>
<td>L2 Private Cache Associativity</td>
<td>4w</td>
</tr>
<tr>
<td>I/D/L2 Block Size</td>
<td>32B</td>
</tr>
</tbody>
</table>

6.4. Analysis of the MAXTR policy

In this section, we evaluate the MAXTR policy by comparing to PHPL [Isci et al. 2006] and MAXT [Mariani et al. 2009] policies. Since the given heuristics work with a fixed power budget, we set up an evaluation in which it is varied as a fraction of 12W (which is the overall power consumption of the considered benchmarks).

In the actual scenario, application execution requires the processing of a limited number of jobs. To maximize the system performance when applications become idle, one can either a) reallocate computing resources every time an application becomes idle (such as in MAXT) or b) reserve the computing resources to the application within a certain time-window in order to quickly restart the execution if new jobs for the same application arrives within a specific time window $\tau$ (such as in MAXTR).

For our run-time validation, we generated a set of scenarios by using a Markov process where job inter-arrival times are exponentially distributed (i.e. a Poisson process). Note that the proposed methodology does not make any assumption on the specific input arrival process. To generate the test case, a Markov arrival process was chosen since it is widely adopted for modeling different systems. Among other, [Simunic et al. 2001] adopts this arrival process to model job arrivals for an MPEG decoder application deployed on a mobile embedded system. During the generation of the scenarios, we consider that arrival rates are changing smoothly over time. Besides, the time window size used for MAXTR is equal to 0.5 s. Figure 15 shows the average performance indices obtained in these scenarios for the analyzed policies (power consumption and average job response time). The response time of a job is defined as the time the job spends in the system from its arrival to its completion (the total time spent waiting and on service). The response time is a particularly valuable metric for interactive and/or real-time applications.

First of all, we can observe that all policies are able to meet the power budget constraint (Figure 15(a)). We can also observe that the proposed MAXTR technique always overcomes PHPL in terms of average job response time.

When the power budget is relatively high, the RRM policy providing the best performance is MAXT (as shown in Figure 15(b)). In fact, a higher power budget allows the RRM to allocate more
resources, the platform can provide a higher throughput and the average number of jobs resident in the system is lower (Figure 16). On the other hand, with a lower power budget, the MAXTR policy shows a better response time.

To provide more insight in the run-time system behavior, Figure 17 shows the system performance trend for a specific scenario considering a power budget of 9 W.

First of all, we can observe that the power budget, even if it is met on average, it is rarely exceeded (Figure 17(a)). This effect is due to the variability in the power consumption between the execution of different jobs that were not predictable at design-time. We can also observe that for what concerns the average system throughput, the three policies behave similarly (Figure 17(b)). In fact, the average throughput is bounded by the job arrival rate (the system cannot elaborate more jobs than the ones arrived so far) and all RRM policies are serving it without significant penalties. However, a significant difference can be found in the average job response time (Figure 17(c)), where the MAXTR policy provides the lower value overall.

**Overhead analysis** The RRM algorithms for the different policies have been implemented in C and overheads have been estimated via simulation of the host processor.

Figure 18(a) shows for each of the RRM policies the minimum, mean and maximum overheads resulting from a single RRM invocation together with the cumulative RRM overhead due to the invocations during the 60s period; specifically, a single invocation of the RRM policies to select the application parallelization takes always less than 3ms. Mean execution time overheads for a single RRM invocation are: 0.39ms for MAXTR, 0.45ms for MAXT and 2.5µs for PHPL. Mean overheads are higher for the policies MAXT and MAXTR because they perform a complete search for the optimal combination of the software parameters among the operating points identified at run-time.
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.

The *cumulative* overhead of MA XTR is about one order of magnitude smaller than MA XT because MA XTR is invoked only at the end of time windows (\(\tau=0.5s\)). During our trial 60s simulations (see Figure 18(b)), MA XTR and PHPL are invoked significantly less frequently than MA XT. Overall, the *cumulative* overheads during the simulation period (60 sec) for the three policies are: 47ms (MA XTR), 256ms (MA XT) and 0.3ms (PHPL) respectively.

Once an operating configuration is selected, the applications might need to switch parallelization. In this paper, we consider code versioning to enable the switching. In practice, at the switching point the new application version should be loaded. The overhead introduced using the Linux operating system for our technology is of the order of magnitude 1-10 ms [McVoy and Staelin 1996]. The time window of the proposed RRM 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%.

7. AUTOMOTIVE COGNITIVE SAFETY SYSTEM - CASE STUDY

In this Section we present the application of the proposed methodology to the optimization of a multiple-stream MPEG encoding chip dedicated to automotive cognitive safety tasks. In this context, an Automotive Cognitive Safety System (ACSS) is installed on a vehicle equipped with a wide range of sensors (such as cameras and radars). The goal of the ACSS is to keep passengers safer by signaling and/or actuating the following assisted/emergency scenarios:

— Forward collision warning.
— Automatic pre-crash emergency braking.
— Lane departure warning and guidance.
— Lane change assist/blind spot assist.

In this paper, we assume to have a vehicle in urban scenario with three cameras associated with the left, center and right mirrors (see Figure 19). All three cameras are connected to the multi-core system being designed. The chip encodes the streams and sends them to an off-chip Central Safety Unit (CSU), reducing the needed bandwidth on the on-board buses.

Moreover, the driver can enable a live-view of the actual content of the video streams on his dashboard. We assume that the dashboard has a set of displays which are able to reproduce the live-streams coming to the CSU.

The template architecture for the target case study is the same CMP presented in Section 6 and shown in Figure 8. However, in this case study we introduce an area model derived from [Li et al. 2006] to prune the design points that exceed 25mm$^2$ in terms of area (considering a 32-nm process technology). Table VI reports the resulting design space.

In this case study, to evaluate system metrics we consider an operating frequency of 1GHz. As before, the run-time tunable software parameter is the task level parallelization of the target application whose range from 1 to 16 following powers of 2.

7.1. Design-Time Optimization

The design time optimization is carried out by using the SDCS methodology setting the confidence parameter $\chi$ to 50%. Considering that the simulation of the MPEG encoder application for a rea-
Table VI. Range of hardware design parameters for the ACSS case study.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Issue Width</td>
<td>1 - 8</td>
</tr>
<tr>
<td>L1 Instruction Cache Size</td>
<td>2KB - 16KB</td>
</tr>
<tr>
<td>L1 Data Cache Size</td>
<td>2KB - 16KB</td>
</tr>
<tr>
<td>L2 Private Cache Size</td>
<td>32KB - 256KB</td>
</tr>
<tr>
<td>L1 Instruction Cache Associativity</td>
<td>1w - 4w</td>
</tr>
<tr>
<td>L1 Data Cache Associativity</td>
<td>2w - 8w</td>
</tr>
<tr>
<td>L2 Private Cache Associativity</td>
<td>2w - 8w</td>
</tr>
<tr>
<td>I/D/L2 Block Size</td>
<td>16B - 16B</td>
</tr>
</tbody>
</table>

Table VII. Output of the design time optimization for the ACSS case study.

(a) Best architecture configuration.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Heuristic</th>
<th>Throughput [frames/s]</th>
<th>Power Consumption [W]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Issue Width</td>
<td>2</td>
<td>7.4</td>
<td>1.6</td>
</tr>
<tr>
<td>L1 Instruction Cache Size</td>
<td>16KB</td>
<td>14.1</td>
<td>3.4</td>
</tr>
<tr>
<td>L1 Data Cache Size</td>
<td>8KB</td>
<td>26.4</td>
<td>6.5</td>
</tr>
<tr>
<td>L2 Private Cache Size</td>
<td>256KB</td>
<td>50.6</td>
<td>12.6</td>
</tr>
<tr>
<td>L1 Instruction Cache Associativity</td>
<td>1w</td>
<td>86.3</td>
<td>23</td>
</tr>
<tr>
<td>L1 Data Cache Associativity</td>
<td>8w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Private Cache Associativity</td>
<td>4w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/D/L2 Block Size</td>
<td>16B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) MPEG encoder operating points.

The multi-objective optimization has been carried out targeting the minimization of average per frame execution time and average power consumption. The output of the SDCS exploration is a Pareto set of hardware/software configurations. The final hardware configuration found is presented in Table VII(a) and the guideline operating points are shown in Table VII(b). The execution time for the frames considered in our experimental campaign can be modeled with an exponential distribution. When a frame requires a long execution, successive frames can be temporarily stored in the input frame buffer.

7.2. Run-Time Optimization

We assume that the basic requirement needed by the CSU to operate correctly and achieve safety critical operation is 15 frames per second (fps) for each stream. Besides, whenever the driver activates the live-streams on his dashboard, the CSU communicates a new set of frame-rate requirements. This is done whenever: a) a vehicle is detected in proximity by the CSU, or b) the vehicle speed passes a fixed threshold; there is a threshold every 10 km/h for lateral cameras and every 20 km/h for the central camera.

In this case study, we assume that the vehicle starts in a stationary position. Then it accelerates to reach a maximum speed of 60 km/h. After some seconds of constant motion, the vehicle decelerates and stops. During the travel, the vehicle is passed twice and two cars enter and exit the proximity area on the left. Overall, car speed and frame rate requirements for the three cameras are shown in Figure 20.

Here we compare the system behavior for the MAXT and MAXTR run time policies. We first set the power budget to 23 W, which would be enough to run an MPEG encoder on the 16 cores of the computing fabric.

Figure 21 shows the response time related with the three MPEG streams (left, center, right). The response time represents the time elapsed between the instant a frame enters the input frame buffer and the instant the encoded frame is ready to be transmitted on the on-board buses.

In Figure 21(a) we can observe that, when the system is under a heavy load, the response time of the MAXT policy increases significantly. In fact, when a high frame rate is required for the left
camera, this is not served adequately and some frames are waiting in the input buffer for a significant amount of time. When adopting MAXTR (Figure 21(b)), the system monitors the input frame rate and reserves resources accordingly. In this case, when 30 \text{ fps} are required for the left camera, the system reserves to the left MPEG encoder the right of using up to 8 cores. This results in a lower response time never exceeding 500 ms (Figure 21(b)).

8. CONCLUSIONS

In this paper, we exploited a combined design-time/run-time optimization flow to identify \textit{a)} an optimal hardware configuration which is robust with respect to applications and corresponding data-sets and \textit{b)} a run-time management policy based on the guidelines coming from a design-time analysis to improve performance and response time while meeting a specified power-budget.

The proposed design-time exploration methodology leverages analytical approximations of the system response in order to speed up the optimization by skipping some computationally expensive simulations. The analytical model is based on Artificial Neural Networks (ANNs) and we propose
statistical techniques to prevent the design space exploration process from getting stuck in local minima or false optima associated with models’ approximations.

To exploit as guidelines the operating points generated by the design-time exploration, we proposed an efficient RRM policy (MAXTR) to reduce the response time of the target multi-core platform.

REFERENCES


Evolutionary Computation, IEEE Transactions on 10, 1 (Feb.), 50–66.


