Decision-Theoretic Design Space Exploration of Multiprocessor Platforms

Giovanni Beltrame, *Member, IEEE,* Luca Fossati, and Donatella Sciuto, *Member, IEEE*

***Abstract*—This paper presents an efficient technique to perform design space exploration of a multiprocessor platform that minimizes the number of simulations needed to identify a Pareto curve with metrics like energy and delay. Instead of using semi-random search algorithms (like simulated annealing, tabu search, genetic algorithms, etc.), we use the domain knowledge derived from the platform architecture to set-up the exploration as a discrete-space *Markov decision process*. The system walks the design space changing its parameters, performing simulations only when probabilistic information becomes insufficient for a decision. A learning algorithm updates the probabilities of decision outcomes as simulations are performed. The proposed technique has been tested with two multimedia industrial applications, namely the `ffmpeg` transcoder and the parallel `pigz` compression algorithm. Results show that the exploration can be performed with 5% of the simulations necessary for the most used algorithms (Pareto simulated annealing, nondominated sorting genetic algorithm, etc.), increasing the exploration speed by more than one order of magnitude.**

***Index Terms*—Decision theory, design space exploration (DSE), multiprocessor, system-level design, system-on-chip (SoC).**

I. Introduction

THE CONTINUOUS increase of transistor density on a single die is leading toward the production of more and more complex systems on a single chip, with an increasing number of integrated components and processing units. This brought to the introduction of the system-on-chip (SoC), that integrates on a single medium all the components of a full system. The design and development of such systems raises new challenges due to the large design space and tight design and time-to-market constraints [1].

These parametrized embedded SoC architectures must be optimally tuned (i.e., their configuration parameters must be appropriately chosen) to find the best trade-off in terms of the selected figures of merit (e.g., energy and delay) for a given class of applications. This tuning process is called design space exploration (DSE) [2].

In general, this optimization problem involves the minimization (or maximization) of *multiple objectives*, making the definition of optimality not unique [3]. In particular, the solution of *multiobjective* optimization problems consists of finding the points of the Pareto curve [4], [5], i.e., all the points which are better than all the others for at least one objective. However, a Pareto curve for a specific platform is available only when all the points in the design space have been evaluated and characterized in terms of objective functions. This full search approach is often unfeasible due to the high-cardinality of the design space and to the long simulation time needed for evaluating the objective functions.

In the past, multiprocessor platforms have been explored using either classical heuristic algorithms (such as tabu search, simulated annealing, etc.) [6] or pruning techniques that try to reduce the size of the design space [7]. Both classes of techniques rely on simulation as a means for evaluating the system-level metrics corresponding to a newly found configuration. If system-level simulation can be performed in a reasonable time, these algorithms provide good results. This is generally not true for multiprocessor systems-on-chip, for which simulations can be rather lengthy and time consuming [8]. This paper addresses the platform configuration optimization problem by exploiting the domain knowledge provided by the definition of a design platform. The idea is to move the DSE complexity from simulation to probabilistic analysis of parameter transformations. Exploration is modeled as a Markov decision process (MDP) [9], and the solution to such MDP corresponds to the sequence of parameter transformations to be applied to the platform to maximize (or minimize) the desired value function. The proposed approach requires to simulate the system only in particular cases of uncertainty, massively reducing the simulation time needed to perform the exploration of a system, while maintaining near-optimality of the results.

Overall, this paper provides three main contributions:

1) the development of an efficient exploration methodology that reduces the number of simulations required for DSE;
2) the formalization of platform domain knowledge for MPSoC;
3) the definition of an efficient solution algorithm for the exploration problem.

The algorithm has been tested on two widely used industrial applications, `ffmpeg` (a video encoder/decoder) and a parallel implementation of the `gzip` compression algorithm, and on a small benchmark consisting of an implementation of Bailey's 6-step fast Fourier transform (FFT) algorithm. Results show

Manuscript received October 8, 2009; revised January 12, 2010. Date of current version June 18, 2010. This paper was recommended by Associate Editor D. Atienza.

G. Beltrame is with the the École Polytechnique de Montréal, Montreal, QC H3T 1J4, Canada (e mail: giovanni.beltrame@polymtl.ca).

Luca Fossati is with the European Space Agency's ESTEC, Noordwijk, The Netherlands (e-mail: luca.fossati@esa.int).

Donatella Sciuto is with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milan 20133, Italy (e-mail: sciuto@elet.polimi.it).

Digital Object Identifier 10.1109/TCAD.2010.2049053

that the number of simulations required by our algorithm is one order of magnitude smaller than what required by standard techniques with a comparable level of accuracy.

DSE using MDP is based on the ideas presented in [10]; with respect to that work, the exploration strategy has been refined and optimized and a more extensive set of experiments executed. We have also compared our approach with ten state-of-the-art multiobjective optimization algorithms.

This paper is structured as follows: Section II summarizes classical exploration algorithms for multivariate analysis; Section III introduces the decision model and its solution algorithm, while the application of the methodology to two industrial benchmarks is described in Section IV; finally, Section V draws some concluding remarks.

II. Related Work

Three main classes of techniques have been proposed in literature for aiding the DSE of system architectures:

1) techniques that try to reduce the design space size [4], [11], [12];
2) techniques that provide exploration heuristics [6], [13];
3) techniques based on statistical analysis aimed at guiding the exploration to specific regions of the design space [14], [15].

Once the candidate solutions are identified, two mechanisms (or a combination of both) can be used for the evaluation: detailed simulation [16] or estimation of performance using predictive models [15].

Class (1) techniques limit the exponential increase of design space size by eliminating those configurations that are certainly nonoptimal. In [4], the design space is divided into partitions, and exhaustive search is performed inside each partition. Then the Pareto-optimal configurations of each partition are combined to determine the global curve.

A different technique, proposed in [11], aims at reducing the number of configurations from the product of to the sum of the number of parameters. This strategy orders applicable parameter transformations according to their effect on the system metrics, then applies transformations in that order. Transformation effects are evaluated through exhaustive simulation on a set of benchmarks and, as such, they are valid only for the same class of applications as the benchmarks. Although complexity is highly reduced, the exploration results remain in general sub-optimal. A similar sensitivity analysis is also performed in [12]: the design of experiments (DoE) [17] technique is employed to characterize the impact of the parameters on the system performance. Once sensitivity analysis is performed, heuristics are used to modify parameters and determine the optimal system configuration. It is worth noting how the technique is limited in the number of system configurations that it can take into account.

DoE is also employed by Palermo *et al.* [18] to generate an initial set of experiments, creating a coarse view of the target design space. Response surface modeling [19] is then used to refine the exploration; this process is iterated to cover the design space. Our approach does not need the initial set of experiments, it requires a smaller number of iterations to update probabilities (learning is performed per action and not globally), and better supports nonuniform response surfaces (due to their implicit modeling).

Binary decision diagrams are used by Mohanty *et al.* [7] to reduce the design space: first unfeasible configurations are eliminated, then the remaining space is evaluated through system-wide performance estimation models, further pruning it. In the end, the identified solutions are simulated. The approach is somewhat similar to ours in that we also use estimation models (called *movement vectors* in the following) to identify the nonoptimal configurations, thus avoiding their simulation. We differ in that the estimation models are not simply applied to standard optimization algorithms (such as GAs), but they are tightly integrated with the exploration framework, consistently reducing the number of simulations. [20] uses decision trees to perform DSE of memory management, but without applying decision theory to obtain automated exploration.

Techniques in *Class (2)* apply heuristics to the exploration problem. In this context, genetic algorithms have often been used for architectural exploration [6], [13]. Although they reduce the time needed for exploration by 80% when compared to exhaustive search, their effectiveness is bound to large sample populations. This means that even if the performance gain increases with the size of the population, simulation times still remain excessive for complex applications. Pareto tabu search and Pareto simulated annealing (PSA) belong to this category of function approximation heuristics, and in [16] they are applied to architectural exploration. Both techniques are much more efficient than exhaustive search, but to reach an acceptable accuracy they require a high-number of simulations. A more detailed description on the use of heuristic algorithms in the context of multiobjective optimization is given in [21].

Other approaches apply multiagent exploration algorithms such as ant colony optimization [22] and particle swarm optimization [23]. Although these are very efficient for some class of optimization problems, they require a large number of evaluations (several for each agent).

Algorithms of *Class (3)* also aim at pruning the design space and at directing the search to the most promising regions. Genetic algorithms and exact methods are combined in [14]: the DSE problem is formalized as a multiobjective 0–1 integer linear programming problem. A pseudo-boolean solver is used to force the genetic algorithm to stay in the feasible search space. Since all our search space is feasible, applying such approach to our system would simply result in the application of a heuristic exploration algorithm.

Ascia *et al.* [15] tackle the problem of DSE reducing the number of evaluations and the time required to evaluate (i.e., simulate) a system configuration. They propose the use of evolutionary algorithms (EA) as an optimization technique and fuzzy systems (FS) for the estimation of the objective function: the EA evolves normally while the FS learns from simulation, until it becomes expert and it can be used for estimating system performance. The main drawback of the approach consists of the high-number of simulations necessary to train the FS.

The algorithms described so far approach the problem as a black-box: they do not take into account the peculiar features, the design constraints and the a-priori knowledge of the target platform. For instance, platform-based design [24] methodologies provide a useful amount of knowledge that can be used to define better exploration strategies, consistently reducing the number of required simulations. This paper provides the theoretical infrastructure to exploit this information (referred to as *domain knowledge*), introducing an exploration algorithm based on decision theory.

III. Proposed Methodology

This paper proposes a new methodology for the DSE of platform-based computer systems [25]. During the initial phases of the design, defining the optimal micro-architecture configuration of a system can be very challenging, especially when the effects of varying the configuration parameters are unknown. DSE of a platform can be seen as the path of two points in two spaces: the parameter space P (e.g., the cache size, the number of processors, etc.) and the metrics space M (e.g., the power consumption, the execution speed, etc.). The modification of a parameter (e.g., changing the cache size) is referred to as an *action*, and determines a movement in the parameter space that modifies the current system configuration. This movement is deterministic and discrete and produces a corresponding movement in the metrics space (e.g., a change in the execution time) defining the results of the action.

Therefore, the exploration of the design space of an application platform can be seen as a path that leads an initial configuration to a destination configuration with "better" characteristics.

The results of actions are generally not known a-priori, meaning that they can only be accurately determined through simulation. We propose to substitute simulation with a probabilistic estimation of the effects of an action, modeling the problem as an MDP [26].

Definition 1: An MDP is a tuple $\langle S, A, T, R\rangle$ where:

1) S is the set of possible system states;
2) A is a set of possible actions, i.e., modifications of the platform parameters;
3) $TP_D : S \times A \longrightarrow \Pi(S)$, the state transition function, is the probability density function for every state-action pair: $TP(s, a, s')$ is the probability to get in state s' applying action a from state s;
4) $R : S \times A \times S' \longrightarrow \mathbb{R}$ is the expected reward for each state-action pair, i.e., $R(s, a, s')$ is the expected reward for executing action a from state s and ending in state s' (for example in our domain the reward can be the speed gained from the *action* of doubling the cache size).

An MDP is also characterized by the Markov property [9]: the state transition function TP_D depends only on the current state and the chosen action.

In general, an MDP is represented by a graph G whose nodes have a one-to-one correspondence with the system states S and whose edges are composed of the actions A. In the following, we will refer to G as either a graph or a tree since, as detailed in Section III-G, we do not allow backward actions, thus generating a tree.

Solving an MDP means finding the strategy (i.e., the best action to be performed from each state in the system) that maximizes the expected reward R. This means that the exploration starting in state s will consist of a series of *steps*, trying to maximize a given metric. In our case, we start from a random platform configuration and we walk through the different configurations until we reach the optimal one.

A. Platform Exploration as an MDP

The platform exploration problem can be modeled as an MDP.

1) S is the set of all possible configuration-metrics tuples.
2) A is the set of applicable actions, i.e., modifications to the platform parameters.
3) TP is the probability density over the metrics space; before simulation we just have an estimation of the metric values, and a change in the parameters (e.g., doubling the cache size) corresponds to an interval in the metric space (e.g., an interval in the execution time) and, after metric discretization, to more than one state. TP defines the probability of ending in each one of those states.
4) R is the reward associated with a movement in the parameter space, i.e., the difference between the metric in the starting state s and in the destination state s'. Using the cache example, R is the speedup obtained by modifying the cache size.

The Markov property holds: the effect of a transformation depends only on the configuration to which it is applied and not on past actions that brought to the given configuration.

In general, the *a priori* uncertainty about the effects of actions makes simulation the only way to determine the position of the current configuration in the metrics space. However, platform-based design limits the number of parameters and it partially provides the knowledge to *estimate the effects of actions*. In particular, given a transformation in the parameters space, a corresponding subspace in the metrics space can be determined. For example, it is possible to estimate that doubling the cache yields an execution time in the interval $I = \left(T_{\text{low}}, T_{\text{high}}\right)$; this restricts the use of simulation only for uncertainty conditions, i.e., when the resulting intervals from available actions do not provide sufficient information to make a decision about the best action to be applied.

Algorithm 1 contains a high-level description of the exploration algorithm.

Our approach starts by identifying the configuration parameters $\overrightarrow{p}$ (e.g., cache size, etc.) and the functions $\overrightarrow{f} : P \rightarrow M$, which describe how the parameters affect the system's metrics. The generation of the graph G follows; this graph encodes the exploration problem, representing it as an MDP which is then solved by executing the *Value Iteration Algorithm*, as described more in depth by Algorithm 2. The result are the actions (e.g., cache size increase, etc.) that should be applied to take the system to an optimal configuration. These actions

Algorithm 1: Overall Exploration Strategy

```
Identify the configuration paramters p⃗
Define the movement vectors f⃗
repeat
    Pick a random configuration s_0
5   Initialize the set of states to be examined s̄ = s_0
6   repeat Creation of the Graph G
        forall s_i in s̄ do
            s̄ = s̄ − s_i
            forall a applicable in s_i do
                apply a creating child nodes s_k
                partition s_k metrics
                update transition function T according
                to s_k
                s̄ = s̄ + s_k
            end
        end
    until s̄ = ∅ or depth == l
17  Strategy = Value Iteration Algorithm (G)
    s_curr = s_0, steps = 0
19  repeat Strategy Application
        NS = Strategy(s_curr)
        if ∃ s_i, s_j ∈ NS : Strategy(s_i) ≠ Strategy(s_j)
        with i ≠ j then II kind uncertainty
            s_next = Simulation Results
            if s_next ∉ NS then
24              Error ⟶ Restart All
            end
        else
            s_next = NS
        end
        s_curr = s_next
        check convergence
        steps++
    until convergence or steps >= l
    Simulate last obtained state s_curr
    num_runs++
35  s_0 = s_curr
until num_runs < MAXRUNS or convergence
37 if convergence then
    s_0 = s_curr
    enable forbidden actions
    Restart from line 5
end
Change Utility Function
Restart All
```

are applied to the system and simulation is used to solve uncertainty situations (as shown in Fig. 4). By varying the objective function [27], it is possible to find a set of optimal points, hence covering the Pareto curve [3].

The different steps of the exploration algorithm are described in higher detail in the following paragraphs.

B. Parameters and Metrics Spaces

Let us consider the set $P = \{p_1, p_2, ..., p_n\}$ of platform parameters, where p_k can be, for example, the number of processors, the cache size, the number of threads, etc. Each p_k can be modified by a transformation τ that changes its value as in [11]. For example, given $p_k = cache\ size$, τ can consist of doubling it.

Definition 2: Given a platform configuration $P = \{p_1, \ldots p_k, \ldots p_n\}$, a transformation $\tau(p_k, \Delta)$ produces a configuration $P' = \{p_1, \ldots p'_k, \ldots p_n\}$ where $p'_k = p_k + \Delta$ with $\Delta \in \mathbb{Q}$.

To evaluate the effects of a transformation on the architecture, we estimate the variations induced on a set of reference metrics $M\{m_1, .., m_i\}$ as intervals; we call such intervals "movement vectors." In the following, we consider the $M \triangleq \{T, E\}$ metric space, where T is the execution time of an application and E its energy consumption.

C. Movement Vectors

Definition 3: A movement vector is an interval in the metrics space corresponding to a transformation vector in the parameter space, and it is defined as

$$\langle f_1(\tau(p_k, \Delta)),\ f_2(\tau(p_k, \Delta)),\ ...,\ f_i(\tau(p_k, \Delta))\rangle$$

where $i = |M|$, and $\overrightarrow{f} = f_1, f_2, ..., f_i$ functions that determine the effect of the transformation τ on each metric $m_j \in M$.

For example, considering $M = \{T, E\}$ and the cache size as the system parameter p, f_1 determines how a variation in the cache size affects execution time and f_2 how it affects energy consumption.

To determine the movement vectors $\overrightarrow{f}$ for a given parameter p, it is sufficient to define, for each transformation τ that can be applied to that parameter, the corresponding minimum and maximum resulting values in the metrics space. We use intervals because the exact value of the metrics is not known and it can only be determined through simulation. Considering the execution time T, f will not produce T_{exact} but values $T_{\min}$ and $T_{\max}$ such that $T_{\min} \leq T_{\text{exact}} \leq T_{\max}$ with a certain probability density function. To include effects due to the interaction of platform parameters that are difficult to predict, each interval is extended by 10%. With this setup, in our experiments simulation values have rarely been outside the estimation intervals. However, even if the estimations are not correct (so if $T_{\text{exact}} \leq T_{\min}$ or $T_{\text{exact}} \geq T_{\max}$), the algorithm still leads to the correct result, but it requires a higher number of simulations to produce the Pareto curve.

In the following, we present three actions that can be applied to a complex electronic system and the corresponding intervals in the metric space, in particular we change the number of processors and their cache size. In our experiments, we used many more movement vectors that are not presented here in detail for the sake of brevity.

Notice how all movement vectors should be bound by designer's defined limits. The reason for this is to remain within the technological limits for the system, e.g., it makes no sense to use an infinite number of processors or an infinitely large memory.

Let us now define and explain in detail the movement vectors related to the number of processing elements and the cache size.

1) *Processor Number:* It is possible to evaluate the execution time T and energy E for a given transformation.

- Add δ to the total number of processors np

$$\tau(np) : np' = np + \delta.$$

1) *Lower bounds*

$$T(np') = T(np)\frac{np}{np'}$$
$$E(np') = E(np)$$

this means that in the best case the speedup is proportional to the amount of processing power added to the system, and E remains unchanged.

2) *Upper bounds*

$$T(np') = T(np)$$
$$E(np') = E(np)\frac{np'}{np} + E_{\text{standby}}(np' - np)$$

meaning that no speedup is observed in the worst case, i.e., the added processors were not used. The energy of the system is increased proportionally to the amount of computational power introduced in the system (e.g., going from one to three processors would triplicate system energy) plus the minimal standby energy that would be used by the newly added processors. $E_{standby}$ is defined as the energy associated with the execution of a processor in idle for $T(np')$.

- Remove δ from the total number of processors np—$\tau(np) : np' = np - \delta$. For T, this is the reverse of the previous case, with the upper and lower bound exchanged, i.e., in the worst case the slowdown time is proportional to the amount of processing power removed from the system. For E, the bounds are defined analogously.

1) *Lower bound*

$$E(np') = E(np)\frac{np'}{np}$$

meaning that at best, the energy is reduced proportionally to the number of processors removed from the system.

2) *Upper bound*

$$E(np') = E(np)$$

the energy of the system remains unchanged.

2) *Cache Size:* The movement vectors associated with the cache size can be easily assigned a bound considering that in the best possible case (a perfect cache) the cache will generate zero misses, and in the worst case the cache will always miss. As we are considering a symmetrical parallel system, we evaluate the cumulative amount of cache hits and misses for all processors for the sake of simplicity. This is not a limiting factor for our approach: in case a nonsymmetrical system is considered, one can use separate movement vectors for each cache in the system.

It is worth noting that the energy required by a single cache access strongly depends on its geometry and it is not necessarily proportional to the cache size. We used eCACTI [28] to determine the energy used by caches in our system, calculating both dynamic and leakage power. The former is computed as a cost per access, while the latter depends only on the selected cache and the estimated simulation time.

Given these considerations, with c_h and c_m the number of cache hits and misses, respectively, increasing the cache size dc by a power of two (i.e., $\tau(dc) : dc' = dc \cdot 2$) gives the following.

1) *Lower bound*

$$T(np') = T(np)\frac{c_h}{c_h + c_m}$$
$$E(np') = \begin{cases} E(np)\dfrac{c_h + c_m}{c_h} & \text{if } E_{\text{best}} > E_{\text{cache}} \\ E(np)\dfrac{E_{\text{best}}}{E_{\text{cache}}} & \text{otherwise} \end{cases}$$

meaning that in the best case the cache is perfect and has no miss, and the overall speed of the system is increased proportionally to the amount of cache misses removed from the system. Similarly, the energy of the system remains proportional to the ratio between the energy of the new cache size with no misses (E_{best}) and the previous cache energy.[1] In case, the new cache requires higher energy even without misses, only the system level impact is considered, reducing the energy proportionally to the amount of misses removed from the system.

2) *Upper bound*

$$T(np') = T(np)$$
$$E(np') = \begin{cases} E(np)\dfrac{E_{\text{best}}}{E_{\text{cache}}} & \text{if } E_{\text{best}} > E_{\text{cache}} \\ E(np) & \text{otherwise} \end{cases}$$

meaning that in the worst case, the increased cache had no effect on T and E is either unmodified (if the new cache size has lower energy for the same amount of misses) or proportional to the increased energy use of the cache.

The movement vectors for decreasing the cache size can be determined similarly, as it can be done for several other parameters. It is worth noting that these bounds are very loose, but they are sufficient for good operation of the algorithm.

In general, configuration parameters of complex hardware systems are inter-dependent: it is not possible to study the effect of one of them independently from the others. This is taken into account by the previous equations: as long as the global effect produced by a transformation does not exceed the intervals defined by the upper and lower bounds the algorithm operates efficiently. In theory, there might be situations in which transformations produce unforeseen effects (such as a small change in the cache size consistently slows down the system because of the increased bus contention): in those cases the algorithm detects that the bounds were not valid at the first simulation and restarts from the newly found point. The result is that the algorithm still finds the Pareto curve, but performing more simulations.

D. States

Each state of the system graph G encoding the MDP is identified by the tuple $\langle p, m \rangle$, where p is a point in the

[1] $E(np)$ is the system energy, E_{cache} is the energy used by the cache only

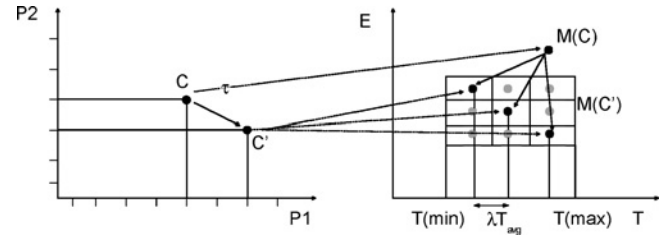

Fig. 1. Automatic partitioning of a 2-D metric space and parameter space.

parameter space P and m a point in the metric space M. There can be the situation where a set X of states share the same p (i.e., the corresponding systems have the same cache size, number of processors, etc.) but different values for m (e.g., execution time). This is due to the fact that m is obtained through the application of the movement vectors, thus it is only an estimation of the real value m'; in particular it is an interval, as explained in Section III-C. Such interval is then partitioned to obtain the states $\langle p, m'_1\rangle$, $\langle p, m'_2\rangle$, etc.

Given the subspace $M(C')$ in the metric space identified by the movement vector associated with a transformation τ, we partition it according to a given accuracy λ and consider the centroid of each partition for the states of the MDP, as shown in Fig. 1. We identify each partition as $d_{i,a,s}$: the i-th partition of the subspace is obtained by applying action a from state s; in Fig. 1 we have three partitions which are identified as $d_{1,\tau,C}$, $d_{2,\tau,C}$, and $d_{3,\tau,C}$.

This partitioning introduces an error given by the difference between the present value of the metrics (still unknown, it can only be determined through simulation) and the centroid of the partition that best approximates it. By increasing the number of partitions we can reduce the error, but it increases the number of states, leading to an exponential increase in the solution time of the MDP [9]. Two mechanisms have been identified to control the trade-off between algorithm accuracy and number of states.

1) We define λ, the *distance* encoding the precision of the proposed algorithm. Each partition d can have a maximum width of $\lambda \cdot | M_{avg} |$: increasing λ reduces the number of partitions (states), but it increases the approximation error. In previous formula, M_{avg} is the average of the considered metric: $M_{avg} = \frac{M_{\min}+M_{\max}}{2}$.
2) We consider an event horizon l, that represents the maximum depth of the decision tree G; this bounds the number of steps considered in the value-iteration algorithm (Algorithm 2), determining the maximum number $|A|^{l+1}$ of examined states, where $|A|$ is the number of modifications which can be performed on the platform's parameters (*actions*). Tuning l implies a trade-off between accuracy and exploration speed since: the smaller is l the more simulations are executed.

E. Actions

The number of actions is one of the dominant factors determining the complexity of MDPs [9]; for this reason we need to reduce as much as possible the number of applicable actions in each state. In our formulation, each action is a transformation of the platform parameters and, in the following, these two concepts are considered interchangeable.

Each parameter can be modified by dual transformations

$$\tau^+ = \tau(k, \Delta)$$
$$\tau^- = \tau(k, -\Delta).$$

The execution of τ^+ is made void by the execution of τ^-, so a subsequent execution of the two actions should be avoided: we already know that we would end up in an already examined state. For this reason, we keep a list of executed actions which we use to forbid dual actions. Dual actions are allowed again only at the end of algorithm execution to make sure that we are not trapped in a local optimum.

F. Transition Functions

As the metrics space is partitioned, $TP(s, a, s')$ represents the probability of reaching a partition with a centroid s' from state s by applying action a. This information cannot be easily determined before simulation, and therefore the initial probability distribution $TP_D(s, a)$ is uniform for all the states s' that can be reached from s by applying action a.

During the generation of the decision tree we can, anyway, correct the probability function. Assuming that we know $\overline{s}$, the actual destination point of a, we can correct the probability distribution $TP_D(s, a)$ by increasing the probability $TP(s, a, \overline{s})$ of ending in $\overline{s}$; this is achieved by using "virtual sample numbers w."

Definition 4: For each partition $d_{s',a,s}$ associated with a state-action tuple $\langle s, a\rangle$, we define the *virtual sample number* $w_{s'}$ as the number of times $\langle s, a\rangle$ ends in $d_{s',a,s}$ when traversing a decision tree from root to leaf.

Transition functions $TP_D(s, a)$ for each state-action pair are refined by adding $c \geq 1$ virtual samples to each partition $d_{s',a,s}$ during the building of the decision tree associated with the MDP. Fig. 2 shows an example of this process: for simplicity we assume that it is possible to execute only one action a (e.g., doubling cache size) and that the metric space is composed only of the execution time.

1) From s action a is applied; the execution time is estimated with the interval $(T_{\max} - T_{\min})$, with $T_{\max} - T_{\min} = 3\lambda T_{avg}$, so we generate three states S_1, S_2, and S_3 with metrics $\left(T_{\min}, T_{\min} + \lambda T_{avg}\right)$, $\left(T_{\min} + \lambda T_{avg}, T_{\min} + 2\lambda T_{avg}\right)$, and $(T_{\min} + 2\lambda, T_{\max})$. The probability density TP_D of ending in any of those states is uniform.
2) A breadth first exploration is applied on the newly generated states; we first examine s_1, corresponding to the best case of action a; we add w_1 virtual samples to the best case of the probability function associated with the edges of action a exiting from the new state s_1, $TP_D(s_1, a)$. w_1 is a user-defined value which determines how quickly the probability density functions should be updated according to the expected results of taken actions. In our experiments, with the particular domain used, we kept $w_1 = 1$, as other values didn't produce any significantly different result (but might in other domains).

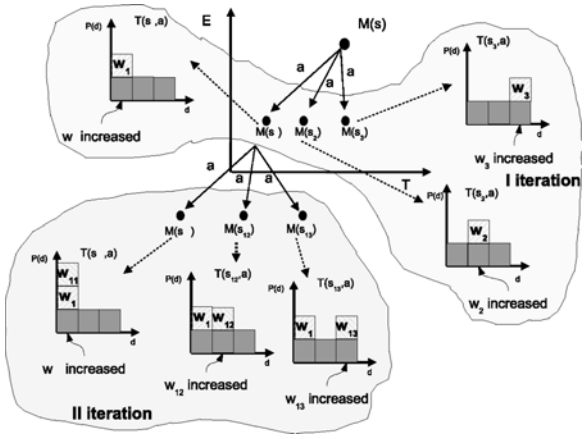

Fig. 2. Evolution of $TP(s_i, a)$ as the decision tree is built: virtual samples change the probability of ending in a partition d_i. Depending on the chosen partition (d_1, d_2, or d_3), we increment the corresponding w_i and update the probability distribution.

3) The same concept is applied to the average and worst case, thus generating $TP_D(s_2, a)$ and $TP_D(s_3, a)$.
4) Using the newly created probability functions we proceed in the exploration this time starting from s_1, s_2, and s_3; new states are generated by applying action a and their probability function TP_D is updated. When the depth of the tree reaches l, the event horizon, the exploration terminates.

This defines a more realistic TP as the algorithm goes on with the exploration, while maintaining the Markov property. To simplify the presentation, the proposed example has the same number of edges exiting from every node; the concept can be easily extended to the general case of a variable number of edges by appropriately redistributing the virtual samples.

G. Exploration Algorithm

As described in Algorithm 1, the exploration process is composed of four steps: generation of the decision tree (line 6), solution of the MDP (lines 6–17), application of the transformations (lines 19–35), restart and optimality verification (line 37).

The *first step* consists of the generation of the graph $G = \{S, E\}$ that describes the behavior of the MDP. A node $s \in S$ in the graph represents a state of the system, and in particular a tuple $\langle P, M \rangle$, where P is a point in the parameters space and M is a point in the metrics space. The fact that dual actions are not enabled, frees G from cyclic structures, simplifying the problem. An edge $e \in E$ is defined as

$$e = \langle s_i, s_j, a, TP(s_i, a, s_j) \rangle$$

and it represents the transition probability (given by the transition function $TP(\cdot)$) from state s_i to state s_j when applying action a.

At the beginning of the exploration, the initial values of the parameters P are chosen randomly and the corresponding metrics M are determined via simulation, building the root node s_0. Starting from s_0, all possible actions are applied, generating the configurations that differ from s_0 by one parameter. For each configuration, movement vectors identify the destination subspace in the metric space (in our case they identify the estimated range for power consumption and execution speed). This is partitioned according to the maximum accuracy λ defined for the exploration. The centroid associated with each partition, together with the corresponding parameter configuration P, generates a new child node, and the difference between parent and child metric defines the reward R of that particular action. Last, for each state s_i, the probability $TP_i(s_i, a, s_k)$ is generated according to the procedure described in Section III-F. The decision tree is progressively built by iterating on the newly generated nodes breadth-first, until either it is not possible to apply any other action on the leaf nodes or the l-th level (our event horizon) is reached. A sample graph is shown in Fig. 3: each action produces an interval in the metrics and various states are generated by partitioning such intervals according to the desired accuracy λ.

Algorithm 2: The strategy evaluation algorithm

```
initialize V(s) = 0
repeat
    forall s ∈ S do
        forall a ∈ A do
            Q^π(s,a) := Σ_{s'∈S} TP(s,a,s') [R(s,a,s') + γV^π(s')]
        end
        V^π'(s) := max_a Q^π(s,a)
    end
until strategy converges
```

The *second step* solves the MDP using the value iteration algorithm [9], as described by Algorithm 2. This phase ends when, for each state s the difference between two successive cumulative returns $V^{\pi'}(s)$ is smaller than an arbitrary small value ϵ, so when there is no substantial improvement of the return. At the end the set of $k < state, action >$ pairs (i.e., the strategy π) to be performed from the initial state in order to obtain an optimal system configuration is identified. Note how k is bound by the event horizon l: there cannot be more actions than the maximum depth of the decision tree [9]. At this point of the algorithm we are evaluating the utility function $T^{1-\alpha}\dot{E}^{\alpha}$, with a given α; as such, *optimal configuration* means the configuration corresponding to the minimum of the utility function.

In the *third step*, the actions identified in the previous stage are applied to the system and the obtained metrics are evaluated after each of those actions. Simulation is used for the evaluation only when strictly necessary: it will be performed only if an action brings the system to *a set* of states (as opposed to a single state) *and* the strategy maps different actions to each of those states. Remember that an action is a modification to the platform parameters; the application of the action generates, in general, an interval in the metrics space: if this interval is smaller than the desired accuracy we have a single state, otherwise the action takes us into multiple states.

In other words, if the strategy suggests action a for a given state $\langle P, M \rangle$, three conditions are possible, as shown in Fig. 4.

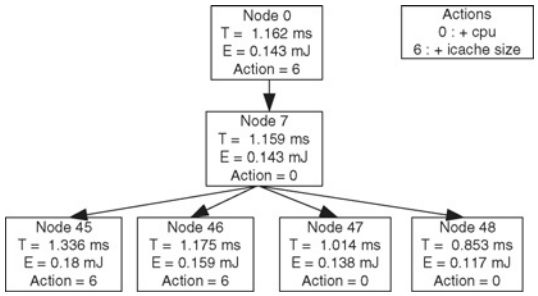

Fig. 3. Sample of decision tree generated by the proposed algorithm: the metric values indicated in the states represent the center of the metric intervals. Action 6 generates only one state as the produced interval is smaller than the accuracy expressed by λ, while Action 0 generated a much larger interval which needs to be spread over four states.

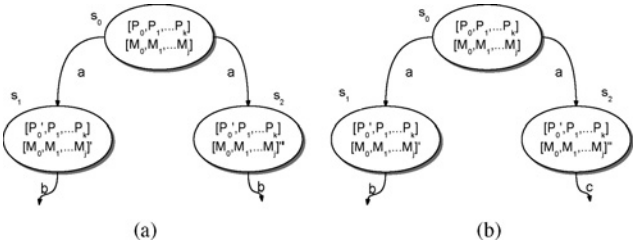

Fig. 4. (a) Uncertainty of I kind. (b) Uncertainty of II kind.

1) a can lead to one state only (i.e., to an interval in the metrics space smaller than λ). The action is deterministic within the chosen accuracy λ and no simulation is necessary.
2) a leads to a set of states (e.g., s_1 and s_2), and the computed strategy maps the same action to all of them (i.e., the action b should be executed from both s_1 and s_2). This means that whichever state the system will end into, the same next action will be chosen, and therefore it is not useful to verify it by simulation. This is called an *uncertainty of the first kind* and it is shown in Fig. 4(a).
3) a produces more than one state, and they are mapped to two or more different actions. Only in this case it is important to discriminate which state the system will actually reach, thus a simulation is performed. This is called an *uncertainty of the second kind*; in Fig. 4(b), the strategy maps actions b and c to the states s_1 and s_2 obtained by the application of action a: simulation is then needed to determine if we are in s_1 (thus action b should be applied) or we are in s_2 and action c should be used.

Every time an uncertainty condition of the second kind arises, the algorithm is stopped and a simulation is performed, fixing its position in the metrics space. It might be that, in case movement vectors are not properly defined, no estimated interval corresponds to the true values of the metrics obtained through simulation. In this case, the correspondence between the simulated configuration p_s and the obtained metric m_s is cached and the whole exploration algorithm restarted. When the algorithm reencounters p_s estimation is not employed but, instead, the true metric value m_s is used, and the exploration can proceed correctly.

Note that when an uncertainty of the first kind arises, exploration continues not on a single state, but on a set $S = \{s', s'', \text{etc.}\}$; the same considerations still hold, keeping in mind that applying the strategy to the set means applying the strategy to each state contained in the set.

The application of the strategy goes on until either all actions are applied or an optimal configuration is found (when the algorithm converges, so when no action gives a gain greater than a user defined value ϵ).

Steps one, two, and three are repeated until either convergence has been reached during the strategy application or the maximum number of runs has been performed.

In the *fourth step*, an additional restart is performed to check that the exploration is not trapped in a local minimum: a new graph G_{new} is created considering the last optimal configuration as the initial node and with forbidden actions enabled: actions that were previously applied can be reverted (e.g., if cache size was increased during exploration, now it can be decreased). The MDP encoded by G_{new} is then solved.

H. Pareto Search

The algorithm works with utility functions [29] that have the utility property [9] (i.e., functions that map states to real numbers); in the context of multiobjective optimization such functions are also called *scalarizing functions*. To have the algorithm generate an approximate Pareto curve, as opposed to generating a single point, it must be possible to perform the exploration in a multivariate environment. This is obtained by repeated applications of the algorithm, changing the scalarizing function at every application so that it covers the whole span of the metric space of interest; this technique, called *parameter variation* is often used to generate the Pareto curve of multiobjective optimization problems [27]. For instance, in the experiments detailed in the following, the design space was explored, similarly to what described in [11], using the scalarizing function $T^{1-\alpha} \cdot E^{\alpha}$ with $0 \leq \alpha \leq 1$. We build a set A of n points uniformly distributed in [0, 1] (e.g., $A = 0, 0.5, 1$), each of which is used as the α value for one run of the algorithm. This means that $|A| = 6$ will require six consecutive runs of the algorithm to determine a Pareto curve. The optimal $|A|$ to be used has been determined experimentally, as detailed in Section IV.

During the exploration, all points simulated by the algorithm are added to the nondominated set if and only if they are not Pareto-covered by other points. At the end of the scan of the design space, the nondominated set constitutes the estimation of the Pareto curve.

IV. Experimental Results

The proposed methodology has been validated using two large applications and a small benchmark for which exhaustive search was possible. The two applications are *ffmpeg*, a video transcoder used to convert a small clip from MPEG-1 to MPEG-4, and *pigz*, a parallel compression algorithm. The small benchmark consists of an implementation of Bailey's 6-step FFT algorithm (*fft6*).

Our experiments have been planned to:

1) determine the number of simulations required by the algorithm to obtain an approximate Pareto-set;

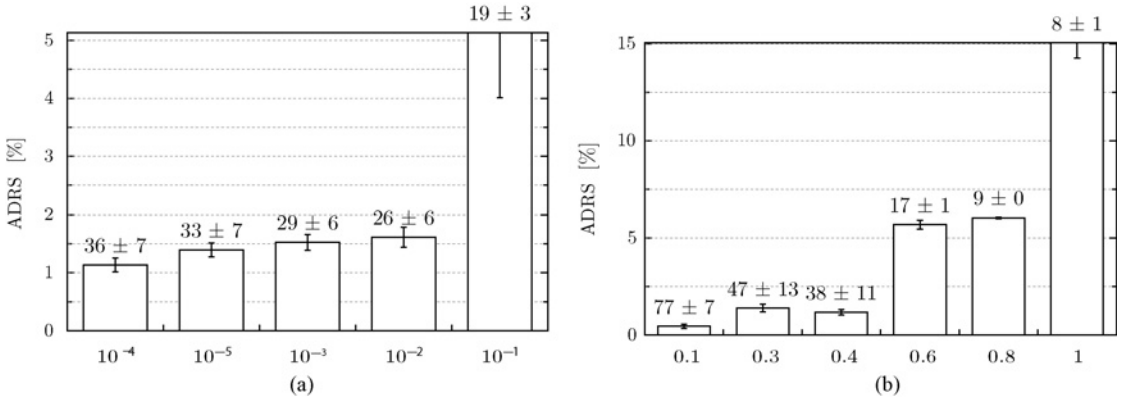

Fig. 5. Average accuracy (ADRS) with varying values of (a) ϵ and (b) λ. The numbers are the average number of evaluations performed for the exploration.

TABLE I
DESIGN SPACE FOR THE EXPERIMENTAL PLATFORM

Parameter	From	To
Number of PEs	1	8
PE frequency	100 MHz	500 MHz
L1 cache size	1 K	32 K
Bus Latency	10 ns	100 ns
Memory latency	10 ns	100 ns
L1 cache policy	LRU, LRR, RANDOM	

2) quantify the quality of the Pareto-set found by the algorithm;
3) compare the results with state-of-the-art optimization algorithms;
4) identify and respond to possible objections to the algorithm performance.

For this purpose, the ReSP [30] open-source simulation environment was used to perform the exploration of the aforementioned applications on a chip-multiprocessor platform. The platform has been explored using the parameters listed in Table I, with a resulting design space of 8640 points.[2] Although the design space is not extremely large, it is comparable with similar works (e.g., [12] uses 6144 points) and the exhaustive exploration of any medium/large application would require an unfeasibly long simulation time (e.g., roughly two months for *ffmpeg*) in any case. Even the full exploration of the simple *fft6* benchmark required six days of uninterrupted simulation. To gather sufficient data for a statistical analysis the *fft6* benchmark was run 10 times for each exploration algorithm. This number was reduced for *ffmpeg* and *pigz* due to the unfeasibly long simulation times, ending with a number of runs between two and ten for each algorithm on all benchmarks.

The proposed algorithm was compared with ten state-of-the-art multiobjective optimization algorithms: pareto simulated annealing (PSA), multiple objective simulated annealing (MOSA), Serafini's multiple objective simulated annealing (SMOSA), multiple objective genetic local search (MOGLS), Ishibuchi's and Murata's multiple objective genetic local search (IMMOGLS), Pareto memetic algorithm (PMA), multiple objective multiple start local search, strength Pareto evolutionary algorithm (SPEA), nondominated sorting genetic algorithm (NSGA), and controlled nondominated sorting genetic algorithm (referred to as NSGA2). The MOMHLib++ [31] library was used in this paper as a reference implementation of such algorithms.

[2]It is worth noting that bus and memory latency are not realistic parameters, but they enlarge the design space to better test the proposed algorithm

A. Estimation of the Number of Evaluations

The number of evaluations needed to obtain an approximate Pareto-set for the proposed algorithm depends on the number of available actions, on the size of their bounds, and on the algorithm parameters l (the event horizon), ϵ (the convergence margin), λ (the accuracy factor), and $|A|$ (the number of α values).

Theoretically, the higher l, the lower the number of simulations needed to reach convergence with the current α setting, but the higher the memory requirement. In our implementation, l is dynamically set according to the available host memory and it remained set to three in most of our experiments (going to four in few cases).

To determine the best values for the parameters λ and ϵ and to observe how they affect the performance of the proposed algorithm, we tested the exploration of the *fft6* benchmark with varying values of λ and ϵ, assessing accuracy [using average distance from reference set (ADRS), see Section IV-B] and number of evaluations. Fig. 5(a) shows that lowering ϵ quickly increases the accuracy, with a sub-linear increase in the number of evaluations. It is worth noting that below 10^{-4}, there is no evidence of significant difference between values of ϵ. Fig. 5(b) shows similar results for λ, with no evidence of significant difference for values below 0.4.

Nevertheless, $\epsilon = 10^{-6}$ and $\lambda = 0.3$ were chosen to maximize the algorithm precision and to assess the algorithm performance in worst case conditions. Even in these conditions, as our experiments show in the following, the number of simulations required is much lower than those required by existing alternatives.

A, the optimal set of α values was determined using the *fft6* benchmark and running the *mdp* algorithm with an increasing number of α values evenly spaced between zero and one. Fig. 6 shows two box plots of the number of evaluations and number of pareto points found with varying $|A|$. In the box plots, the dark line is the median, the boxes represent the range between

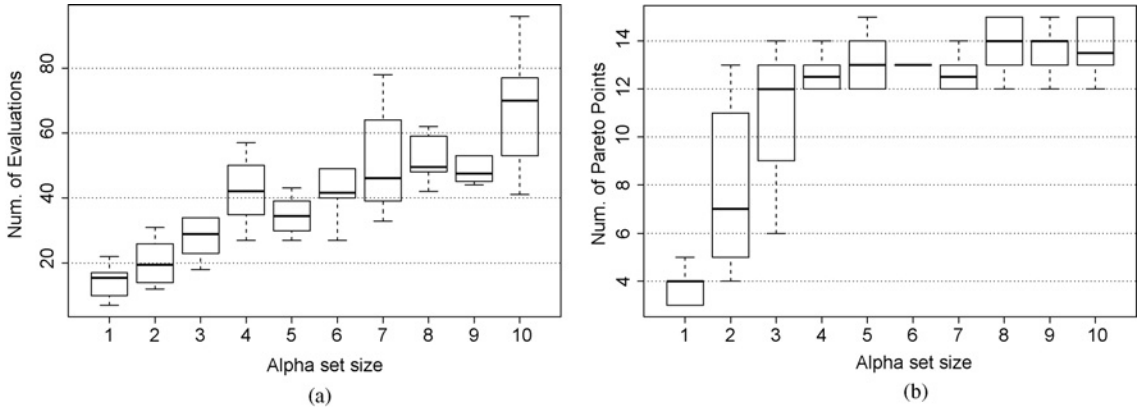

Fig. 6. Boxplots of (a) number of evaluations and (b) number of Pareto points found with varying $|A|$, the α set size. The dark line is the median, the boxes represent the range between the 25th and 75th quartiles and the whiskers are the smallest and largest nonoutlier observations.

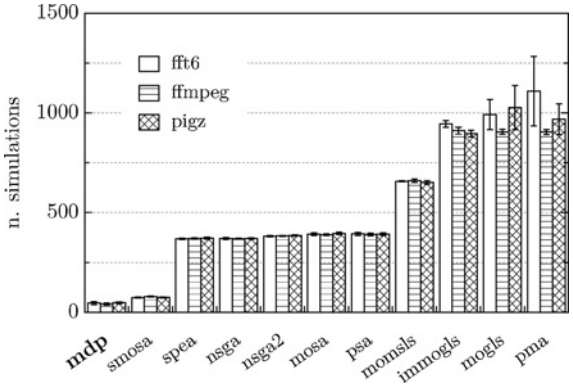

Fig. 7. Average evaluations for algorithm convergence.

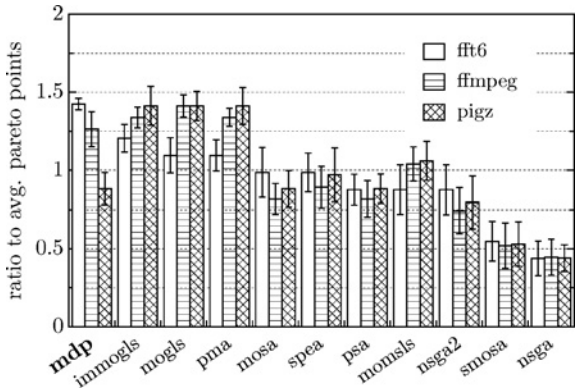

Fig. 8. Number of Pareto points found by each algorithm as a ratio to the global average over all algorithms.

the 25th and 75th quartiles, and the whiskers are the smallest and largest nonoutlier observations.

With $|A| \leq 3$, the number of found Pareto points has high variability, due to the fact that it strongly depends on the starting point of the algorithm. As $|A|$ increases number of found points converges to the maximum value, with no statistically significant difference for $|A| > 5$. Fig. 6(a) that the number of evaluations (i.e., simulations) grows almost logarithmically, but with high variability due to the dependency on the random starting point. This lead to the selection of $|A| = 6$ (i.e $A = \{0, 0.2, 0.4, 0.6, 0.8, 1\}$), which guarantees a median of 40 simulations while finding the maximum number of Pareto points.

Fig. 7 shows how the proposed approach (indicated as *mdp*) compares with other algorithms in terms of number of simulations; note that it is possible to use the number of simulations, and not the actual execution time, as a mean of comparison as the execution times of the state-of-the-art algorithms and of *mdp* are negligible with respect to the simulation time. We tried to our best to find the optimal settings for each algorithm for the design space under consideration, but, still, *mdp* uses ten times fewer evaluations than most algorithms, obtaining comparable, or better, results. The variability of the number of evaluations is very limited, showing stable convergence for all algorithms on all benchmarks.

The number of Pareto points found by *mdp* using $|A| = 6$ is comparable with the one found by other algorithms, running a much higher number of simulations. Fig. 8 shows the number of Pareto points found by each algorithm, normalized by the average found for each benchmark to allow global comparison. *mdp* provides a higher-than-average number of points for *ffmpeg* and *fft6* (being the highest for *fft6* and the second highest for *ffmpeg*) and average for *pigz*. The *best* column of Fig. 8 shows the size of the Pareto-set found over all the executed simulations (which means exhaustive search for the *fft6* benchmark).

In order to verify the effect of the accuracy of the bounds provided, we artificially increased their size 5, 10, and 20%, but results do not show any significant difference in the number of simulations.

Even when comparing *mdp* with recent literature, to the best of the author's knowledge, no results could be found with such a low-number of evaluations and such good estimation of the Pareto curve. The most promising results come from the use of DoE statistical techniques, for example [12] uses a minimum of 34 evaluations to find a single optimal point (maximum performance with a size constraint) and [18] needs $\sim$100 evaluations to determine the Pareto-set in the best case.

B. Quality of the Resulting Approximate Pareto-Set

In addition to obtaining the size of the approximate Pareto-set as in Fig. 8, it is necessary to measure the quality of such

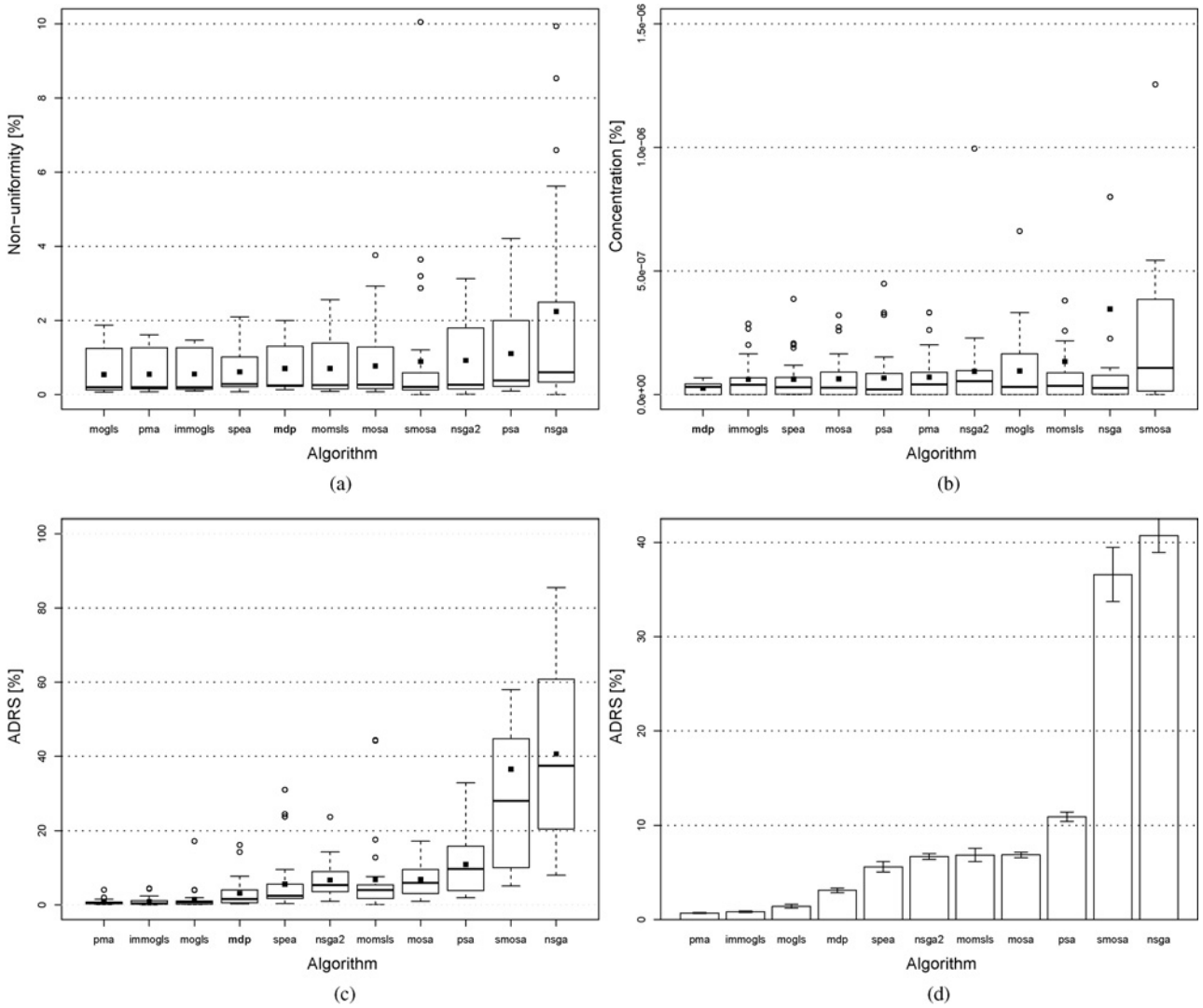

Fig. 9. (a) Non-uniformity, (b) concentration, and (c) ADRS metrics comparing *mdp* with state-of-the-art algorithms. In the boxplots, the black lines are the median, the square dot is the average, the box is the 25–75 interquartile range, the wiskers are the observed maximum and minimum, and the circles are outliers. (d) Average ADRS per algorithm with Studentized 95% confidence interval.

sets. The simplest comparison method among approximate Pareto-sets consists of using the dominance relation presented in [3]. Unfortunately, as it only defines a relative ordering among the approximation-sets, there might be situations in which it is not applicable; moreover, it does not allow quantification of the relative quality of the approximation-sets. In our situation, it reports that the set obtained with *psa* is dominated by *mdp* for the *fft6* and *ffmpeg* benchmarks and that the ones of *nsga* is dominated by *mdp* only for the *fft6* benchamrk; all the other sets are incomparable, they cannot be ordered by the Pareto-dominance relation.

We use three indicators, presented in [32] and [33], to compare the relative quality of the approximate Pareto-set obtained by *mdp* and other state-of-the-art algorithms.

1) *ADRS*—The ADRS [34] is used to compare the approximated Pareto-sets with the best Pareto-set found combining the results of all experiments. This approximates the distance of a considered set from the Pareto-optimal front, and should be minimized.
2) *Non-uniformity*—We measure how solutions are distributed in the design space. Lower non-uniformity means a more evenly-distributed approximate Pareto-set that better estimates the optimal Pareto-set.
3) *Concentration*—We measure the span of each Pareto-set with respect to the range of the objectives. The lower the concentration, the higher the spread of the Pareto-set and the better coverage of the range of objectives.

Fig. 9 shows the comparison of the Pareto-set evaluation metrics using a boxplot. We compared the results of all algorithms using statistical hypothesis testing [17]. In our case, the *null hypothesis* (i.e., what our tests are trying to confute) is that all algorithms behave in the same way on average, and the alternative hypothesis is that the algorithms have a different behavior on average.

Concerning non-uniformity, all algorithms behave similarly on the available benchmarks, as shown in Fig. 9(a), with the exclusion of a few outliers. We performed an analysis of variance (ANOVA) [17] using a 95% confidence interval,

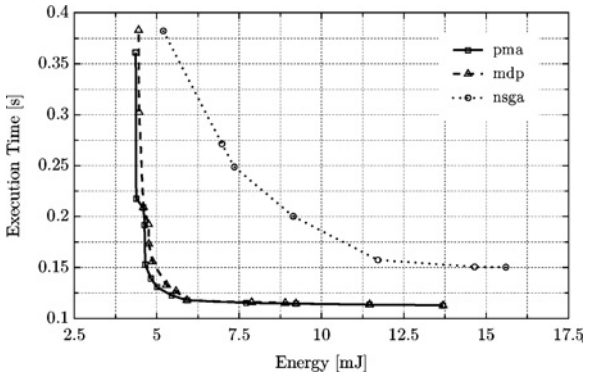

Fig. 10. Average approximate Pareto-set for PMA (the most accurate algorithm), *mdp*, and NSGA (the least accurate) for the fft6 benchmark.

assuming different unknown variances for each algorithm. The F-test (used to determine if any statistically significant difference between averages is present) returns positive with a p-value[3] in the order of 0.001, meaning that the current results are outside the 95% confidence range for the null hypothesis.

Fig. 9(b) shows that concentration has no significant difference between all the algorithms: the F-test reported differences with $p = 0.06$, meaning that there is no substantial evidence at 95% confidence that the algorithms have different average behavior, with the only exception in the very high variability of SMOSA.

The accuracy of *mdp* was measured as the ADRS; the reference set consists of the Pareto-set obtained after collecting all simulation results from all algorithms, or exhaustive simulation in the case of fft6. ANOVA reports that algorithms have different average ADRS with a p-value close to zero, and Fig. 9(c) shows the boxplot of ADRS per algorithm, ordered by average. Using repeated Welch's t-tests [17] between mdp and all the other algorithms, evidence suggests that *mdp* is one of the most accurate algorithms, being surpassed only by PMA, IMMOGLS, and MOGLS, three algorithms requiring a number of simulations two orders of magnitude higher than *mdp*. When considering effect size, the difference between PMA (the best algorithm) and MDP is marginal, between 0.9% and 3.1% ($p = 0.007$).

A more easily-appreciated comparison of all the algorithms is shown in Fig. 9(d), where all algorithms are ordered by increasing average ADRS with 95% confidence interval error bars: if confidence intervals are not intersecting, evidence suggests different average behavior. To exemplify the differences in ADSR, the average approximate Pareto-set for PMA (the most accurate algorithm), *mdp* and NSGA (the least accurate) for the fft6 benchmark are presented in Fig. 10.

In conclusion, although none of the algorithms fully dominates any other on average, *mdp* outperforms most of them for accuracy, with similar concentration and non-uniformity, using one order of magnitude fewer simulations. The three marginally more accurate algorithms require two orders of mangnitude more evaluations.

[3]The p-value is the probability of a set of observations given that the null hypothesis is true, i.e., the probability of the current results if all algorithms had the same behavior on average

C. Possible Objections to the Proposed Algorithm

At first glance, one might come up with a number of possible objections to the proposed algorithms. Deeper analysis, however, shows that these objections do not stand the experimental results of this paper. To anticipate the reader's concern and simplify the understanding of this paper we list here the most common objections and their response.

1) *Parameters cannot be considered separately, the bounds will be too complex to manage*. Although it is true that parameters are interacting, using bounds has exactly the purpose of including every possible interaction within the bounds. Using the simple movement vectors presented in Section III-C still guarantees state-of-the-art accuracy with a fraction of the simulation time. Even if the bounds do not contain the true metric value, *mdp* still operates correctly, even though requiring more simulations.
2) *There is no guarantee the algorithm will scale with larger design spaces*. It is true that the number of nodes in the state graph can grow exponentially with the number of actions, but using the λ parameter one can control the growth, reducing the number of generated nodes, trading-off a larger number of simulations. As long as there is enough memory for an event horizon greater than two, we believe *mdp* will perform better than other pseudo-random algorithms.
3) *There is no clear differentiation between algorithms*. The results we obtained are based on a solid statistical foundation, and provide no clear evidence of a different behavior for most algorithms in terms of accuracy, non-uniformity, and concentration. However, there is a clear difference in terms of simulation time, and this is where the proposed algorithm really shines, using up to two orders of magnitude fewer simulations.

V. Conclusion

This paper presented an application of decision theory, through the use of MDP, to architectural multivariate exploration for platform-based designs. The proposed algorithm exploits the domain knowledge intrinsic in a hardware platform to reduce (more than one order of magnitude) the number of simulations needed for exploration. The proposed technique has been applied to two multimedia industrial applications (the `ffmpeg` transcoder and the parallel `pigz` compression algorithm) and to an `fft6` kernel. Results show an excellent overall exploration speedup ($> 80\%$) and very good accuracy of the discovered Pareto curve. Future work includes the combination of pseudo-random techniques to MDP to exploit the best characteristics from both worlds.

References

[1] G. Martin, "Overview of the MPSoC design challenge," in *Proc. 43rd ACM/IEEE Design Autom. Conf.*, 2006, pp. 274–279.
[2] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[3] E. Zitzler, L. Thiele, M. Laumanns, C. Fonseca, and V. da Fonseca, "Performance assessment of multiobjective optimizers: An analysis and review," *IEEE Trans. Evol. Comput.*, vol. 7, no. 2, pp. 117–132, Apr. 2003.
[4] T. Givargis, F. Vahid, and J. Henkel, "System-level exploration for Pareto-optimal configurations in parameterized systems-on-a-chip," in *Proc. Int. Conf. ICCAD IEEE/ACM*, 2001, pp. 25–30.
[5] F. Catthoor, K. Danckaert, K. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*. Berlin, Germany: Springer, 2002.
[6] M. Palesi and T. Givargis, "Multiobjective design space exploration using genetic algorithms," in *Proc. 10th Int. Symp. CODES*, 2002, pp. 67–72.
[7] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multigranular simulation," *SIGPLAN Not.*, vol. 37, no. 7, pp. 18–27, Jul. 2002.
[8] J. Yi and D. Lilja, "Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations," *IEEE Trans. Comput.*, vol. 55, no. 3, pp. 268–280, Mar. 2006.
[9] S. J. Russell and P. Norvig, *Artificial Intelligence: Modern Approach*, 1st ed. Englewood Cliffs, NJ: Prentice Hall, 1995.
[10] G. Beltrame, D. Bruschi, D. Sciuto, and C. Silvano, "Decision-theoretic exploration of multiprocessor platforms," in *Proc. ACM/IEEE CODES+ISSSS*, 2006, pp. 205–210.
[11] W. Fornaciari, D. Sciuto, C. Silvano, and V. Zaccaria, "A sensitivity-based design space exploration methodology for embedded systems," *Design Autom. Embedded Syst.*, vol. 7, no. 1, pp. 7–33, 2002.
[12] D. Sheldon, F. Vahid, and S. Lonardi, "Soft-core processor customization using the design of experiments paradigm," in *Proc. DATE*, 2007, pp. 1–6.
[13] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," in *Proc. IEEE/ACM Int. Conf. ICCAD*, 1997, pp. 522–529.
[14] M. Lukasiewycz, M. Glay, C. Haubelt, and J. Teich, "Efficient symbolic multiobjective design space exploration," in *Proc. ASP-DAC*, 2008, pp. 691–696.
[15] G. Ascia, V. Catania, A. G. D. Nuovo, M. Palesi, and D. Patti, "Efficient design space exploration for application specific systems-on-a-chip," *J. Syst. Archit.*, vol. 53, no. 10, pp. 733–750, 2007.
[16] G. Palermo, C. Silvano, and V. Zaccaria, "Multiobjective design space exploration of embedded systems," *J. Embedded Comput.*, vol. 1, no. 3, pp. 305–316, 2005.
[17] D. C. Montgomery, *Design and Analysis of Experiments*, 5th ed. New York: Wiley, Jun. 2000.
[18] G. Palermo, C. Silvano, and V. Zaccaria, "An efficient design space exploration methodology for multiprocessor SoC architectures based on response surface methods," in *Proc. Int. Conf. SAMOS*, 2008, pp. 150–157.
[19] L. W. Schruben and V. J. Cogliano, "An experimental procedure for simulation response surface model identification," *Commun. Assoc. Comput. Mach.*, vol. 30, no. 8, pp. 716–730, 1987.
[20] D. Atienza, J. M. Mendias, S. Mamagkakis, D. Soudris, and F. Catthoor, "Systematic dynamic memory management design methodology for reduced memory footprint," *Assoc. Comput. Mach. Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 2, pp. 465–489, 2006.
[21] A. Jaszkiewicz and P. K. Redakcyjnego, "Multiple objective metaheuristic algorithms for combinatorial optimization," Habilitation thesis, Inst. Comput. Sci., Poznan Univ. Technol., Poznan, Poland, 2001.
[22] A. Tumeo, C. Pilato, F. Ferrandi, D. Sciuto, and P. Lanzi, "Ant colony optimization for mapping and scheduling in heterogeneous multiprocessor systems," in *Proc. Int. Conf. SAMOS*, 2008, pp. 142–149.
[23] G. Palermo, C. Silvano, and V. Zaccaria, "Discrete particle swarm optimization for multiobjective design space exploration," in *Proc. 11th EUROMICRO Conf. DSD*, Sep. 2008, pp. 641–644.
[24] A. Sangiovanni-Vincentelli, L. Carloni, F. D. Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proc. 41st Design Autom. Conf.*, 2004, pp. 409–414.
[25] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *Proc. IEEE Int. SoC Conf.*, 2006, pp. 199–202.
[26] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, Mar. 1998.
[27] M. Laumanns, L. Thiele, and E. Zitzler, "An efficient, adaptive parameter variation scheme for metaheuristics based on the epsilon-constraint method," *Eur. J. Oper. Res.*, vol. 169, no. 3, pp. 932–942, Mar. 2006.
[28] N. Dutt and M. Mamidipaka, "eCACTI: An enhanced power estimation model for on-chip caches," Center Embedded Comput. Syst., Univ. California, Irvine, Tech. Rep. TR-04-28, 2004.
[29] T. Murata, H. Ishibuchi, and H. Tanaka, "Multiobjective genetic algorithm and its applications to flowshop scheduling," *Comput. & Ind. Eng.*, vol. 30, no. 4, pp. 957–968, Sep. 1996.
[30] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto, "ReSP: A nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration," in *Proc. Asia South Pacific Design Autom. Conf.*, 2008, pp. 673–678.
[31] *MOMH: Multiple Objective Meta Heuristics* [Online]. Available: http://home.gna.org/momh/
[32] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evol. Comput.*, vol. 8, no. 2, pp. 173–195, 2000.
[33] C. Erbas, *System-Level Modeling and Design Space Exploration for Multiprocessor Embedded System-on-Chip Architectures*. Plaats, Amsterdam: Amsterdam University Press, Oct. 2006.
[34] T. Okabe, Y. Jin, and B. Sendhoff, "A critical survey of performance indices for multiobjective optimisation," in *Proc. Congr. CEC*, vol. 2. 2003, pp. 878–885.

Giovanni Beltrame (M'08) received the M.S. degree in electrical engineering and computer science from the University of Illinois, Chicago, in 2001, the Laurea degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2002, the M.S. degree in information technology from Cefriel, ICT Institute, Politecnico di Milano, in 2002, and the Ph.D. degree in computer engineering from the Politecnico di Milano, in 2006.

Until 2009, he was with the European Space Agency, Noordwijk, The Netherlands, as an Engineer. He is currently a Researcher with the École Polytechnique de Montréal, Montreal, QC, Canada. His current research interests include modeling and design of embedded systems, artificial intelligence, and robotics.

Luca Fossati received the Laurea degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2006, and the M.S. degree in electrical engineering and computer science from the University of Illinois, Chicago, in 2007. He received the Ph.D. degree in computer engineering from Politecnico di Milano, in 2010.

He is currently a Contractor with the European Space Agency, Noordwijk, The Netherlands. His current research interests include modeling and design of embedded systems, compilers, and genetic algorithms.

Donatella Sciuto (S'84–M'87–SM'08) received the Ph.D. degree in electrical and computer engineering from the University of Colorado, Boulder, and the MBA degree from Bocconi University, Milan, Italy.

She is currently a Full Professor with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milan, Italy. She has published more than 200 papers in international journals and conferences. Her current research interests include embedded systems design methodologies and architectures.

Dr. Sciuto has been a member of different program committees of EDA conferences and is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS, the *Journal Design Automation of Embedded Systems*, and the IEEE EMBEDDED SYSTEMS LETTERS. She was the Program Chair of DATE in 2006, Vice-General Chair of DATE in 2007, and General Chair in 2008. She was the Vice General Chair of ESWEEK in 2009 and will be the General Chair in 2010. She is the Vice-President of Finance of the IEEE Council of EDA. She is a member of IFIP 10.5 and EDAA.