Application Autotuning to Support Runtime Adaptivity in Multicore Architectures

Davide Gadioli Gianluca Palermo Cristina Silvano
Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria
Email: {name.surname}@polimi.it

***Abstract*—In this work, we introduce an application autotuning framework to dynamically adapt applications in multicore architectures. In particular, the framework exploits design-time knowledge and multi-objective requirements expressed by the user, to drive the autotuning process at the runtime. It also exploits a monitoring infrastructure to get runtime feed-back and to adapt to external changing conditions. The intrusiveness of the autotuning framework in the application (in terms of refactoring and lines of code to be added) has been kept limited, also to minimize the integration cost. To assess the proposed framework, we carried out an experimental campaign to evaluate the overhead, the relevance of the described features and the efficiency of the framework.**

I. Introduction

Several classes of applications expose to the designer some *application-specific* parameters, also known as *dynamic knobs* [1], that influence the algorithm behavior, such as the resolution in a video processing application or the number of trials in a Monte-Carlo solver. A change in the values of those parameters can influence the performance of the application because they can be related to the amount of data to be processed or iterations to be done for completing the assigned task. In the past, the configuration of such parameters was not considered at all as a possible degree of freedom. The application performance were defined by execution time, or throughput, leading the developer to choose the fastest configuration that reaches an acceptable level of quality. However, modern applications are growing in complexity and they usually expose an increasing number of algorithm-specific parameters and relevant target metrics, e.g. energy consumption. Meanwhile, multicore systems are the de facto standard in modern platforms and the trend is to further promote heterogeneity. One effect of using such architectures is that they introduce additional *resource-related* parameters that can be exposed at application level (for instance the number of threads) not directly impacting the functionality of the algorithm, while modifying its extra-functional characteristics.

A manual selection of the optimal application configuration is a difficult and time-consuming task since the design space might be huge and the performance might be composed of some conflicting metrics. This makes the manual selection of a single configuration as output of the optimization process no longer possible, thus relying on the Pareto set as trade-off concept. In literature there are several studies on Design Space Exploration (DSE) techniques [2] used to efficiently and automatically compute the Pareto set, however the decision-making mechanism adopted to select which one of the trade-off solutions should be deployed is always constrained by problem-specific requirements. Autotuning is defined as the ability to automatically find the best configuration satisfying the application requirements.

So far, autotuning was mainly been considered as a design-time phase to be done off-line. However, design-time autotuning can be considered only a starting point, because applications might change their requirements at runtime depending on external conditions and the execution environment. Therefore, the pre-determined trade-off could be no longer valid, thus requiring to change the configuration on the Pareto set. If the platform is mobile and relies on a battery, the application might be interested in a more energy-aware configuration, but when it is plugged on the power supply, it will change its requirements towards a more performance-oriented one. Additional runtime changes requiring a re-tuning can be due to the data-dependent performance of applications (e.g. Video Processing) or other system workloads that can generate noise and performance penalties depending on the access to shared resources. This dynamicity in terms of requirements and actual execution environment needs to be faced by offering the possibility to the application to runtime adapt.

The paper presents a light-weight framework, named ARGO[2], to enhance applications with a runtime adaptation layer. ARGO is meant to be deployed for a wide range of applications minimizing the integration effort from the application's developer. The target application is a source code, written in C++, that should repeatedly execute a task and expose at least one parameter that influence its behavior. The mechanism used to leverage the application behavior is to change the values of its *dynamic knobs* according to the applications requirements. The proposed framework is grounded on the Monitor-Analyze-Plan-Execute (MAPE) feedback loop defined in [3]. On one hand, there is the Application-Specific RunTime Manager (AS-RTM) that is in charge of automatically tuning the application knobs according to the requirements and the design-time knowledge. On the other hand, the monitor infrastructure senses the execution context and enables the application to self-adapt. The main contributions of the ARGO framework are:

- To autonomously select the best configuration according to the application requirements without any coordination with external actors;
- To adapt at runtime the application configuration to face changes in the execution environment or in the application requirements, expressed by multi-objective constraints;
- To minimize the integration effort and avoiding code refactoring.

This work was supported in part by the EC under the grants HARPA FP7-612069 and CONTREX FP7-611146.

[2]The name ARGO, has been borrowed by Greek mythology. ARGO was the ship on which Jason and the Argonauts sailed to retrieve the Golden Fleece. As that boat was a means for achieving the Golden Fleece (their goal), this framework aims at letting applications to reach their goals too.

978-1-4673-7311-1/15/$31.00 ©2015 IEEE

The remainder of this work is organized as follows. Section II provides an overview of the related work. Section III describes in details the proposed framework, while Section IV explains the integration between ARGO and a generic application. Experimental results are presented in Section V. Finally, Section VI concludes the paper.

II. Related Works

In general, tunable software knobs in applications can be represented as alternative algorithms deployed during application execution. The net effect perceived by the user is a different output QoS for the same input. Additionally, each configuration might use a different amount of hardware resources (including CPU cycles, memory, bandwidth etc.) which leads to different performance and energy/power consumption [4]. In addition to software knobs as integrating parts of the algorithm, in literature there are several techniques, such as loop perforation [5] and task skipping [6], that are not linked to a specific application but can be applied to trade-off accuracy of results and performance. The proposed framework is completely agnostic with respect to the nature of the software knobs.

Especially in the context of High Performance Computing, there are several autotuning frameworks targeted to specific tasks. ATLAS [7] for matrix multiplication routine, FTTW [8] for FFTs operations, OSKI [9] for sparse matrix kernels, SPIRAL [10] for digital signal processing, Patus [11] and Sepya [12] for stencil computations are some examples in this area. Because of their application-specific design, those types of approaches are not easily portable to other application fields. ARGO is more independent from the application domain, similarly to the Green framework [13] and PowerDial [1]. In particular, the work in [13] uses loop and function approximation to perform a trade-off between a single QoS metric loss and execution time improvement. The framework builds its model in the initialization phase and provides a mechanism to sporadically update its information at runtime. The problem of this approach is the time to converge and its constrained applicability to a limited set of kernels. In [1], the authors propose a more general framework, rooted in control theory, that enables an application to adapt for reaching its performance goal. It uses the concept of application heartbeats [14] to dynamically monitor the throughput and, in case the goal is not reached, to trade-off a single QoS metric and performance. In particular, PowerDial autotunes application-specific *dynamic knobs* to adapt and drives the configuration selection by means of design-time knowledge. ARGO and PowerDial address the same problem, however ARGO enables a developer to exploit tradeoffs among more than two metrics and provides a generalized monitoring infrastructure to sense the application performance and execution environment.

Additional autotuning frameworks can be found in the HPC domains: ATune-IL [15], Active Harmony [16], OpenTuner [17] and PetaBricks [18]. They mainly rely on loop related knobs in the code and predictable execution context, despite of an application independent implementation. In the last group, there are also those works addressing the system adaptability through resource management and allocation, i.e. Arte [19], SEEC [20], AQuoSA [21], Tessellation [22] or the work proposed by Hanning et al. [23]. Although those methodologies are orthogonal to the decisions taken by the proposed framework, ARGO is able to sense the execution environment and consequently to adapt the application parameters to the resources assigned by the resource manager [24]. To this group belong the Barbeque RTRM [25], whose AS-RTM subsystem was the starting point used to design the proposed framework. However, the integration process in Barbeque RTRM requires the application code to match a specific abstract execution model. On the other hand, ARGO is designed as an autonomous adaptive layer that abstracts the multi-objective application requirements, thus avoiding to interact with any external actors and without any assumption on the source code structure of the application.

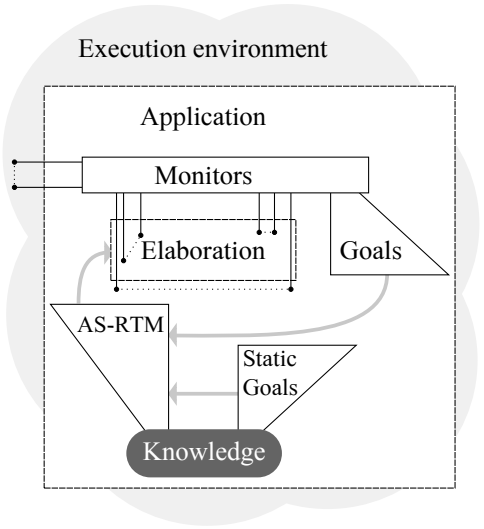


Figure 1. The framework structure. The AS-RTM selects the best configuration according to the runtime information provided by the monitors and the design-time knowledge.

Comparing to state-of-the-art approaches, ARGO has been designed to autotune any kind of software knobs exposed by the application, while adapting the configuration selection depending on runtime changes. Moreover, ARGO exploits its flexibility at a very limited integration cost in terms of additional lines of code and workflow refactoring.

III. Framework Description

The proposed framework is based on a modular approach. The monitor infrastructure uses only runtime information, while the AS-RTM mainly focuses on design-time knowledge. Figure 1 shows the framework structure. In particular, the monitor infrastructure is able to gather runtime information. The designer expresses the application requirements by using goals, when they are linked to runtime information, otherwise by using static goals. The AS-RTM tunes the application parameters according to (1) runtime and design-time knowledge and (2) the application requirements, by closing the loop. With respect to the MAPE loop [3], the monitor infrastructure represents the Monitor element, the goals and static goals defines the Analyze element, while the AS-RTM is the Plan element. However, it is the application that is in charge to executing the planned action, thus representing the Execute element.

A. Monitoring infrastructure

The monitor infrastructure has as central goal to keep updated the knowledge of the proposed framework at runtime. The monitors are application-level probes used to observe both

high-level (e.g. throughput and QoS) and low-level (e.g. hardware counters) metrics. The monitoring layer is flexible and follows a strict object-oriented approach to use the general idea of monitors as base class to be extended to trace each specific metric. The base Monitor class is composed of a circular buffer with a parametric size that stores the observed data by giving the possibility to extract statistical properties (mean, median, max, min, variance, etc.). To add a monitor that observes a specific custom metric which is not currently implemented, the Monitor class needs to be extended by implementing only the methods that actually gather the information. Being ARGO agnostic on the nature of the monitored metric, the Monitor class also implements all the features required to interact with the rest of the framework. ARGO provides a suite of predefined monitors with wide applicability both at high- and low-level. Some examples of monitors implemented in ARGO are:

Time Monitor. This monitor reads the time elapsed between a start point and a stop point. It uses the std::chrono interface and might be configured to use different time units (e.g. *nsec*, *usec*, *msec*, *sec*).

Throughput Monitor. This monitor computes the throughput as the amount of elaborated data over the observed time interval. The metric is $data/second$. The time interval is measured as a difference between a start point and a stop point as the time monitor, while reporting also the throughput.

Memory Monitor. This monitor observes the resident set size of the virtual memory that the process is using. To gather the data, it parses the "/proc/self/statm" metafile. The measure is expressed in kilobytes, thus the monitor stores integer values.

System CPU Usage Monitor. This monitor computes the average utilization of the processors at the system-level. The unit of measure is a percentage, and it is computed as the system busy time (both on user and system level), over the considered time interval. To collect these data, the monitor parses the "/proc/stat" metafile. The metafile values are updated with a granularity of $msec$, but to get a significant measure, the interval of time should be greater than $50msec$.

Process CPU Usage Monitor. This monitor is similar to the System CPU Usage Monitor, but it computes the average utilization of the processor by the application. It is defined as the time the application spent executing on the processors over the elapsed time. The std::chrono interface is used to compute the latter, while the *getrusage* function at OS level is used for the former. Even in this case, to get a significant measure the interval of time should be greater than $50msec$.

PAPI Monitor. It is used to observe low-level metrics by wrapping the widely adopted PAPI [26] framework. It enables an application to observe platform-related metrics, such as cache misses or instruction per cycles, in a transparent way. The maximum number and type of observed metrics depend on the platform.

A concept linked to the monitors is the *Goal*, which represents a single requirement of the application on a monitored metric, i.e. *"The average throughput must be greater than 3 frame per seconds"*. At runtime the application might check if the goal is achieved, change the goal value, get the absolute or relative error and the Normalized Actual Penalty (NAP). The latter measures the degree of satisfaction with respect to the observed value and is defined as follows:

$$NAP = \left| \frac{goal_{value} - observed_{value}}{goal_{value} + observed_{value}} \right|$$

The Goal requires to define a Monitor to be used to keep track of the metric, the value of the goal, the statistical properties of the monitored metric (such as the average) and the comparison function (e.g. *"greater (or equal) than"* or *"less (or equal) than"*). Optionally, it is possible to set the minimum number of elements that the Monitor must have observed to evaluate the goal. If the Monitor does not have enough elements, the Goal is considered achieved by default.

B. Application-Specific Run-Time Manager

The purpose of this module is to exploit design-time knowledge to select the best configuration for an application, taking into account the information coming from the monitors. ARGO represents such knowledge as a list of Operating Points (OPs). Each OP represents a single configuration for the application in terms of software knobs (list of parameters), enhanced with the performance profiled at design-time (metrics value). Since we address multi-objective requirements, the application performance is usually composed of several metrics. The list of OPs derives from a Design Space Exploration (DSE) phase and all the points are Pareto-optimal with respect to the target metrics. This is not a strict requirements, because the list of OPs represents all the possible configurations that are available (thus that can be selected), while the metrics are used to predict the effects of changing OP. Obviously the Pareto filtering reduces the number of OPs and thus the overhead for selecting an OP.

Complementary to the concept of Goal presented before, a Static Goal represents the same concept, but on a metric contained in the OP definition not monitored at runtime. In fact, sometimes is not possible to monitor a metric at runtime, because it is too expensive to be computed, not available or can be constant during the execution. The main difference is that a Goal compares the goal value against the observed one, while a Static Goal compares the goal value against the one in the current OP. However, the application can always check if the goal is achieved, change the goal value, get the absolute or relative error and the Normalized Actual Penalty (NAP) as done for a Goal. Since they represent the same concept, in the following we will refer to both goals (static and not) as ArgoGoals.

As mentioned before, the objective of the AS-RTM is the selection of the best OP in the list. The AS-RTM uses two different concepts to define the application requirements: the rank value and the list of constraints. While the rank value consists of a combination (i.e. geometric or linear) of objective metrics to be minimized or maximized, the list of constraints is composed of all the ArgoGoals that are required by the application, ordered by priority. For example, in a H.264 encoder, we suppose to express the application requirements as *"minimize the PSNR, while maintaining a throughput of 25* $frame/second$ *and using at most* 300.0% *of CPU usage"*. In this case, the PSNR is considered as the (only) metric to determine the rank value while the list of constraints is populated by the goals on the throughput and CPU usage. In other words, the AS-RTM uses the list of constraints to prune the invalid OPs, which are those that break at least one constraint, while it uses the rank value to order the remaining solutions and thus selecting the best OP. When there are no OPs that respect the list of constraints, the AS-RTM relaxes the constraints starting from the one with the lowest priority.

Since the runtime observations provide a view only on the used configuration, the AS-RTM exploits proportional error

```
1  #include ``argo.hpp''
2
3  int param;
4
5  int main ()
6  {
7    argo::init();
8
9    while( work_to_do() )
10   {
11     argo::block_foo
12     {
13       // do the computation
14       do_job(param);
15     }
16   }
17 }
```

(a) Application code integrated with ARGO

```
1  <blocks><block name="foo" /></blocks>
2  <monitors>
3    <monitor name="foo_time" type="TimeMonitor" block="foo"/>
4  </monitors>
5  <goals>
6    <goal name="exec_time" block="foo" monitor="foo_time">
7      <spec dFun="Average"  cFun="Less" observations="3" />
8      <fixed value="200"/>
9    </goal>
10 </goals>
11 <managers>
12   <manager block="foo" start_state="default">
13     <state name="default">
14       <rank type="Linear" objective="Maximize">
15         <field name="param" coefficient="1.0" />
16       </rank>
17       <constraint field="exec_time" target="exec_time" priority="10"/>
18     </state>
19     <expose><param field_name="param" var_name="param"/></expose>
20   </manager>
21 </managers>
```

(b) Required XML configuration file

Figure 2. This example shows how to integrate ARGO in an existing application code that exposes the elaboration as a loop, using the glue-code automatically generated by the framework tool from an XML configuration file.

propagation to extend the new information to all the other OPs. For example, if in the current configuration, the measured throughput is halved with respect to the expected one, the framework assumes that the throughput of all the OPs will be halved. Which is equivalent to double the actual goal value.

A simple implementation of this algorithm consists of updating the design-time knowledge of each constraint with runtime information, if the constraint is related to a Goal, and to loop over the list of OPs and apply the definition to find the best one. However, the computational overhead introduced is constant and linear with respect to the number of constraints and OPs and it is paid on every iteration of the elaboration. To reduce the overhead, the proposed approach bases the decision to be taken at the i-th iteration on the one taken at the previous iteration. In this way, the overhead introduced by ARGO is proportional to the magnitude of the changes from the previous iteration, which is the same of the simple implementation in the worst case scenario, but it is lower in the general scenario. In the remainder of the work we use the *term* valid Operating Point to refer to an OP that satisfies all the constraints in the list.

It is important to stress the fact that the rank value and the list of constraints are defined against the structure of an OP. Since it is the designer that defines the OP metrics and parameters, the framework can be applied to any elaboration that manifests a stationary performance with respect to a given configuration. Which might be composed not only by "command line" options, but also by values that influence the application, for example the index of a function version. This provides to the framework a considerable flexibility.

In the Introduction, we stated that the application requirements can completely change based on external stimuli or the application internal state. For this reason, ARGO gives the possibility to the developer to define an arbitrary number of states, each one characterized by a different rank function and list of constraints, to be selected at runtime. The framework structure is versatile enough to be deployed in a very wide range of applications.

IV. INTEGRATION

To facilitate the integration of ARGO in the application, the framework provides a tool that automatically generates the required code from XML configuration files. In particular, the developer needs to express for each elaboration block the list of OPs, the desired monitors and the application requirements. This information is contained in two different types of configuration files. The first one includes the list of OPs, while the second one contains the ARGO configuration. In this way, it is possible to exploit a separation of concerns: the source code of the application defines the elaboration logic, the ARGO configuration file describes how the framework must adapt, while the list of OPs defines the design-time knowledge of the application. The structure of the XML file that represents the list of OPs is very simple: it contains a list of elements, where each element represents an OP and it is composed of the list of metrics and the list of parameters. Every field of the OP is represented as a pair name-value. In the current implementation, the tool uses the MULTICUBE [27] syntax. The ARGO configuration file is mainly composed of four sections including the list of all the elaboration blocks of the application, the list of used monitors and possible related goals and finally the application requirements for each elaboration block.

To better clarify the required effort, Figure 2 provides an integration example considering a mini application. Figure 2a shows the original source code written in black, while the integration code required to adopt the proposed framework written in bold red. The application itself is very simple: on lines 9-16 the elaboration block, named "foo", performs the loop over the available jobs, while the function *do_job* (line 14) actually performs the computation. In this example, we suppose that the elaboration is influenced by the parameter *param*, expressing the amount of processed data and representing the software knobs of the application. We also defined for the application a target requirement that is to maximize the amount of data computed by the function considering a time constraint of $200ms$ on the function call latency.

The configuration file required to adapt the elaboration at runtime is shown in Figure 2b. Line 1 represents the list of the elaboration block of the application, in this case there is only the "foo" block. The second section of the file is described at lines 2-4 and it defines the used monitor. The third section represents the ArgoGoals and it is listed in lines 6-10. In this example, the designer defines a goal on the average of the observed execution times, in particular they must be lower than $200ms$. The Goal is evaluated only if the monitor observes at least three iterations, determined by the observation field. The remainder of the configuration file defines the application requirements. In particular it creates a Constraint on the field "exec_time" of the OP, using the Goal "exec_time" as target, while the constraint function maximizes the OP field "param". Moreover, Line 19 describes that the value "param" included in the best OP determined by ARGO should be exposed (thus copied) in the local variable "param" within the application code to determine the adaptation.

Starting from the enhanced source code and the configuration file, the integration tool automatically generates the required glue-code for the adaptation. The application developer can in any case use directly the API exposed by ARGO to bypass the automatic generation of the code.

V. Experimental Results

The experiments described in this section have been carried out to assess the benefits of the proposed framework. Since it operates at runtime, impacting on the application execution time, in Section V-B the overhead introduced by ARGO have been evaluated. Section V-C describes two use-cases to assess the ARGO ability to adapt, while Section V-D aims to evaluate the benefits of targeting multi-objective adaption with respect to a state-of-the-art approach.

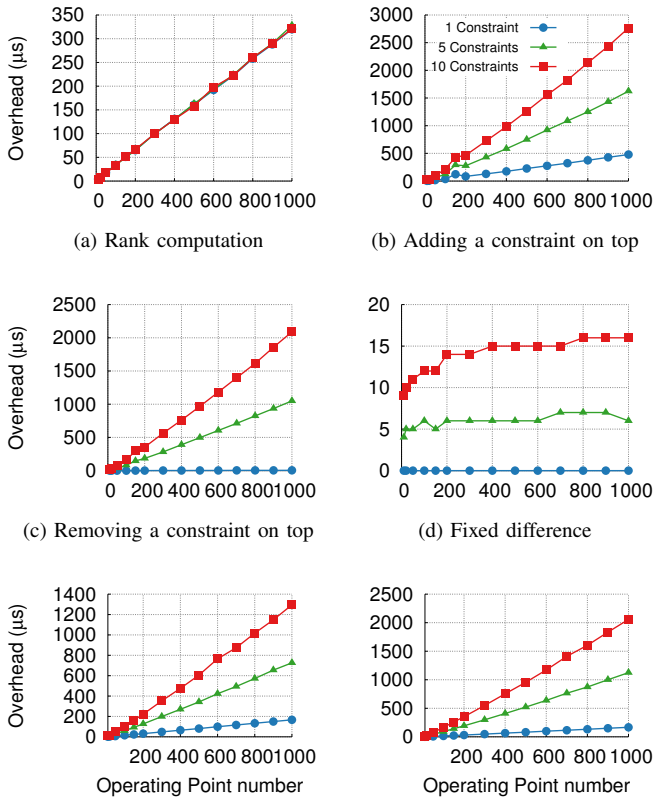
A. Experimental Setup

We consider a case study based on the Stereo-Matching application. As described in details in [28], the algorithm is implemented with OpenCL APIs [29] and it has been designed to export a set of 6 parameters (software knobs) impacting on both application-specific and platform metrics. Five out of six parameters have been derived from the algorithm, while one parameter is used to control the application parallelism.

The elaboration block under analysis wraps together all the OpenCL kernels composing the Stereo-Matching application representing the processing of each pair of images. The changes in the parallelism level are performed through the OpenCL device fission mechanism, to let ARGO selects a different device every time it needs to change the fraction of used resources. The number of frame elaborated in one second (fps), is the measure of the application throughput. A quality metric is computed as the disparity error of the output given the current OP with respect to the exact result. Since the exact result is unavailable at runtime, this metric is measurable only at design-time. The target platform used for the experiments is a workstation with an Intel Xeon QuadCore CPU E5-1607 at 3.0 GHz and 8 GB RAM, running a Linux distribution based on kernel 3.5. OpenCL 1.2 runtime provided by Intel OpenCL SDK 2013.

B. Overhead evaluation

To evaluate the overhead introduced by ARGO, we have set up a benchmark suite to evaluate the execution time of the most demanding operations of in the worst case scenario (WCS). Every operation is repeated several times to average the noise from the measures. Figure 3 shows the results of the benchmarks by varying the number of OPs and constraints, up to 10 and 1000 respectively. In particular, we have evaluated the following operations:



(a) Rank computation (b) Adding a constraint on top

(c) Removing a constraint on top (d) Fixed difference

(e) Constraints become unsatisfiable (f) Constraints become satisfiable

Figure 3. Evaluation of the ARGO overheads in the worst case scenario.

1) Rank Computation: This operation computes the rank of each OP by using the geometric rank function over three metrics. Figure 3a shows that the time spent to compute the rank value is dominant with respect to the time needed to update the constraints internal structure. The time seems to be proportional only with respect to the number of OPs. In this experiment, all the OPs are valid, forcing the framework to update the internal structure of each constraint.

2) Adding a constraint: When the user adds a constraint, a sorting operation is involved. Internally the implementation uses the std::sort function to organize the constraint internal structure. Figure 3b shows the results of the operation. In the WCS, all the OPs are valid and the application adds a constraint on top of them, which is satisfied by all the OPs. This forces the lower constraints to check all the OPs. For this reason, the overhead is proportional to the number of constraints, however the dominant factor is the number of OPs.

3) Removing a constraint: Removing a constraint forces the framework to update the internal structure of all the other constraints. In the WCS, the application removes the constraint with the highest priority, when all the OPs are valid for all the constraints of the list, except for the constraint on top, which cannot be satisfied by all the OPs. Figure 3c shows the overhead of this operation. It is interesting to notice that the

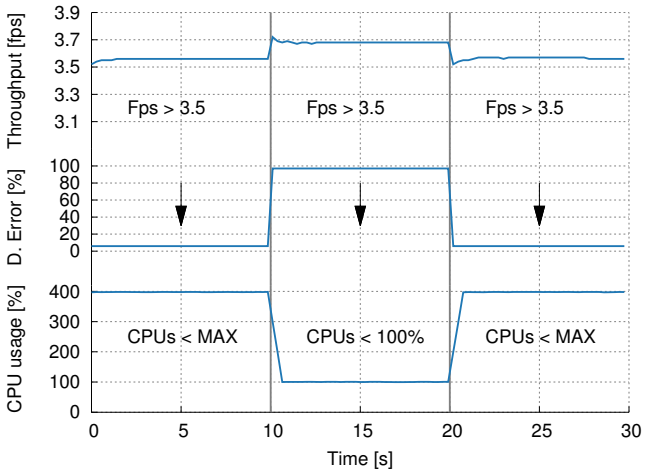


Figure 4. Execution trace of the application in the first use-case.

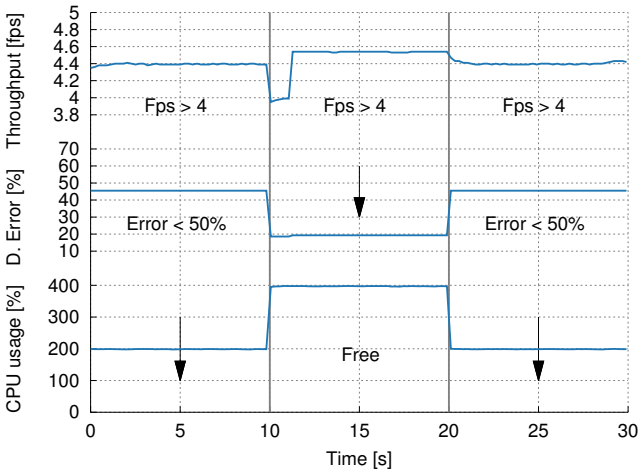


Figure 5. Execution trace of the application in the second use-case.

dominant factor of the introduced overhead depends on the number of constraints. This is due to the fact that when an OP becomes valid, each constraint must allocate memory to keep track of this event.

4) Fixed difference: In Section III-B, it is stated that the complexity of finding the best OP is proportional to the difference between the current situation and the previous one. This experiment (Figure 3d) aims at proving that the overhead caused by a change that involves a fixed number of OPs, is independent on the number of OPs.

5) Unsatisfiable requirements: This experiment aims at evaluating the overhead of selecting the best OP when in the current situation there are no valid OPs. In the WCS, all the OPs were valid for every constraint in the previous situation, however in the current one the constraint with the highest priority invalidates all the OPs. The results in Figure 3e show the overhead of the operation.

6) Satisfiable requirements: This experiment evaluates ARGO in the symmetric situation with respect to the previous case. In the WCS, all the OPs are invalidated only by the constraint with the highest priority, thus relaxing its goal value, the constraint validates all the OPs. The result in Figure 3f shows the same behavior as removing a constraint. In fact, ARGO performs the exact operation done when a constraint is removed, except for freeing the used memory from the removed constraint.

C. Dynamic Behavior

The experiment described in this section shows how the ARGO framework dynamically reacts to changes in the application requirements, by considering two use-cases on the stereo-matching application.

The first use-case aims at assessing the framework capability to adapt to system changes. In particular, the application requirements have been set on maintaining a throughput greater than $3.5fps$, while minimizing the disparity error for the entire execution. To create the need of adaptation, within the period 10s-20s the available resources have been reduced to only 1 core out of the 4 available, e.g. to accommodate high priority task. To acquire runtime knowledge, the application uses a Throughput Monitor and a Process CPU Usage Monitor, with an observation window of 5 elements. The execution trace of the application is shown in Figure 4. Up to $10s$, the OP selected by ARGO respected the throughput requirement presenting a normalized error of 6%. After $10s$, the change in the requirements made ARGO to select a different OP, since the previous one is no more valid due to the resources. In the selected OP, the application continues to respect the throughput requirements at the cost of a normalized error value equal to 97%. If no OPs are valid, the framework is unable to deliver such performance by only tuning the defined knobs. Thus the only available solution for ARGO was to sacrifice the application throughput and to notify the event. When all the resources returned to be available (after $20s$), the framework adapts its status by selecting the initial OP. It is important to notice that ARGO is agnostic on the meaning of the metrics: the possibility to increase the throughput by decreasing the quality is implicitly contained in the design-time knowledge.

The second use-case aims at showing how the framework reacts when the application has more than one state, with completely different requirements. In this scenario, the Stereo-matching application can be considered as part of a surveillance scenario. In particular, the application uses a low-energy state, if no potential threats are detected, otherwise the application is allowed to use all the resources to analyze the environment. To express these requirements, the designer defines two states in ARGO as well. The low-power state, is defined as a constraint on the minimum error of the elaboration (50%) and on the required throughput ($4fps$). The latter has higher priority, while the rank function minimizes the used CPU resources. When the application detects a threat, it must focuses on acquiring as much information as possible. For this reason, the second state uses only one constraint on the throughput ($4fps$), while the rank function minimizes the disparity error. In this state the resources are not constrained.

Figure 5 shows the execution trace of the application in this use-case. After $10s$ the application detects a potential threat and switches the ARGO State. In this case, it is possible to notice that the design-time knowledge slightly overestimates the actual performance of the OP. As consequence of this behavior of the application, using the selected OP on the current platform, the monitored application throughput is not respecting the requirements. The framework perceives the situation and reacts by selecting a faster OP. This configuration generates a slightly worst normalized error value, but it enables the application to reach its goal.

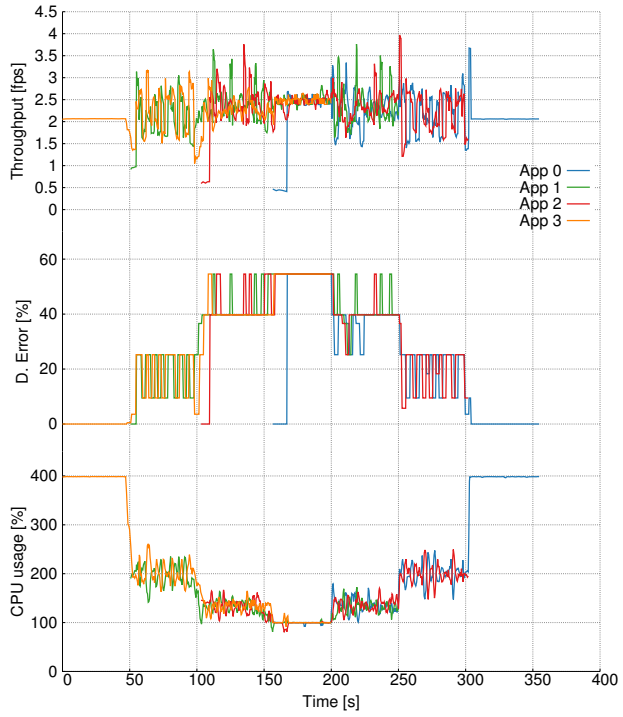


Figure 6. Execution trace for PowerDial approach

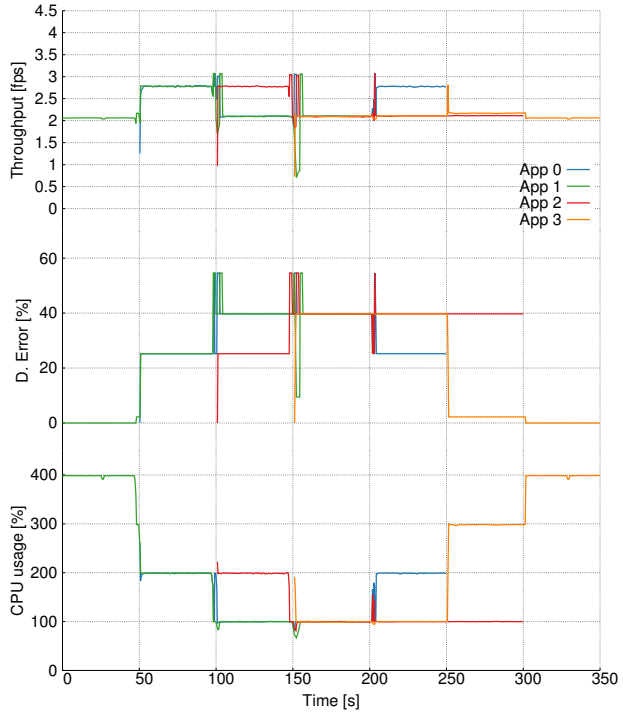


Figure 7. Execution trace for ARGO approach

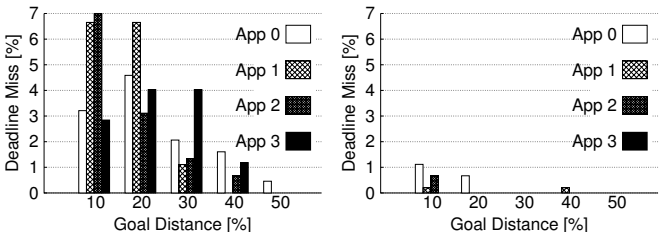
D. Comparison Results

This section shows the benefits of addressing multi-objective requirements by comparing ARGO to the PowerDial [1] approach. In particular, the PowerDial framework uses a control system to obtain the required speed-up to adapt, while the design-time knowledge is used to translate the speed-up value in an application configuration. Following the PowerDial approach, the baseline requirements minimize the normalized error, while providing a minimum throughput value, $2fps$. The framework is enabled to tune also resource-related parameters in terms of the number of used CPUs, besides the application-specific ones. By using ARGO, we have the possibility to add a constraint, that forces each application to use only the *available CPU fraction* of the system, thus using multi-objective requirements. The *available CPU fraction* can be defined (as described in [24] where ARGO has been used as resource manger) as the fraction of the processor that the application can use, defined as follows:

$$CPU_{available} = \Gamma - \gamma + \pi_{measured}$$

where Γ represents the maximum system CPU quota of the platform, γ is the monitored CPU fraction used system-wide, while $\pi_{measured}$ is the CPU quota actually assigned to the application. The experiment launches four application instances, in a sequential scenario. In particular, all the applications are executed for $200s$, but they start to execute with a delay of $50s$ to each other. Figure 7 shows the execution trace of the applications using the ARGO approach. Focusing on the throughput, it is possible to notice that it is more stable and predictable with respect to the PowerDial approach (Figure 6). This behavior is due to the fact that the design-time knowledge no longer models the application performance. Usually the DSE is performed in isolation, thus for the Stereo-matching application, the design-time knowledge is forged with the information that by increasing the number of threads, the application throughput increases as well. This is likely to hold also for several computing-intensive applications. However, if more than one application is executed in parallel, resource contention could happen, since all the applications take autonomous decisions and changing resource-related parameters do not impact the elaboration quality.

The proportional error propagation used by ARGO to adapt the design-time knowledge according to the runtime observation can reduce the improvement, however, it does not change the basic fact that by increasing the number of threads, it also increases the application throughput. This leads each application to use more resources to reach the throughput goal and the effects of the resource contention are shown in the execution trace. In particular, the application throughput oscillates due to the effect of the Linux scheduler. Moreover, the framework attempts to compensate the oscillation, but actually it boosts the magnitude of the oscillation, until the resources are saturated. To cope with this problem, the ARGO approach limits the usage of the resources according to the *available CPU quota*. However, without a global resource manager that assign resources to the application, this procedure cannot be fair. Since the Linux scheduler is fair, on average the PowerDial approach is able to achieve the required throughput, however several deadline misses occur, due to the application oscillations. Figure 8 shows the percentage of deadline misses on the throughput, grouped by the distance between the measured application performance and the goal value, in percentage. It is possible to notice that the ARGO approach generates deadline misses only during the transitory due to the start or termination of each application. On the contrary, the PowerDial approach continuously generates deadline misses all over the time period where more than one application is active.



(a) PowerDial approach (b) ARGO approach

Figure 8. Distribution of deadline misses with respect to the goal.

VI. Conclusions

In this work, we presented the ARGO framework, to provide an application adaptation layer to dynamically react to changes in the application requirements and to face with variability in the execution environment. The framework flexibility enables the designer to express multi-objective requirements, by abstracting domain-specific knowledge. The modular implementation enables the framework to observe at runtime any metric relevant for the application to better adapt, while the separation of concerns minimizes the integration effort in the application source code. We have evaluated the framework overheads in the worst case scenario and its adaptation capabilities to face with runtime changes. Moreover we have compared ARGO with the PowerDial approach to evaluate the benefits of targeting multi-objective requirements. Experimental results show how the proposed framework is flexible, thus enabling the designer to successfully deploy it in a wide range of scenarios. Moreover, the overhead evaluation proves that ARGO can be considered a lightweight approach.

References

[1] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *ACM SIGPLAN Notices*, vol. 47, 2012.

[2] G. Palermo, C. Silvano, and V. Zaccaria, "Respir: a response surface-based pareto iterative refinement for application-specific design space exploration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2009.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, 2003.

[4] Y. Fei, L. Zhong, and N. K. Jha, "An energy-aware framework for dynamic software management in mobile computing systems," *ACM Trans. Embed. Comput. Syst.*, pp. 27:1–27:31, May 2008.

[5] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010.

[6] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006.

[7] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998.

[8] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, 2005.

[9] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*. IOP Publishing, 2005.

[10] M. Püschel, J. M. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing alogorithms," *International Journal of High Performance Computing Applications*, 2004.

[11] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011.

[12] S. A. Kamil, "Productive high performance parallel programming with auto-tuned domain-specific embedded languages," DTIC Document, Tech. Rep., 2013.

[13] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *ACM Sigplan Notices*. ACM, 2010.

[14] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats for software performance and health," in *ACM Sigplan Notices*. ACM, 2010.

[15] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Engineering parallel applications with tunable architectures," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010.

[16] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011.

[17] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014.

[18] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011.

[19] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano, "ARTE: An Application-specific Run-Time managEment framework for multi-cores based on queuing models," *Parallel Computing*, Sep. 2013.

[20] H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. Miller, S. Neuman, M. Sinangil, Y. Sinangil, A. Agarwal, A. Chandrakasan, and S. Devadas, "Self-aware computing in the Angstrom processor," 2012.

[21] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "Aquosaadaptive quality of service architecture," *Software: Practice and Experience*, 2009.

[22] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor *et al.*, "Tessellation: refactoring the os around explicit resource containers with continuous adaptation," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013.

[23] F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau, "Resource-aware programming and simulation of mpsoc architectures through extension of x10," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2011.

[24] E. Paone, D. Gadioli, G. Palermo, V. Zaccaria, and C. Silvano, "Evaluating orthogonality between application auto-tuning and run-time resource management for adaptive opencl applications," in *25th IEEE International Conference on Application-specific Systems, Architectures and Processors–ASAP*, 2014.

[25] P. Bellasi, G. Massari, and W. Fornaciari, "A rtrm proposal for multi/many-core platforms and reconfigurable applications," in *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC'12)*, 2012.

[26] V. M. Weaver, D. Terpstra, H. McCraw, M. Johnson, K. Kasichayanula, J. Ralph, J. Nelson, P. Mucci, T. Mohan, and S. Moore, "Papi 5: Measuring power, energy, and the cloud," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013.

[27] C. Silvano, W. Fornaciari, G. Palermo, V. Zaccaria, F. Castro, M. Martinez, S. Bocchio, R. Zafalon, P. Avasare, G. Vanmeerbeeck *et al.*, "Multicube: Multi-objective design space exploration of multi-core architectures," in *VLSI 2010 Annual Symposium*. Springer, 2011, pp. 47–63.

[28] E. Paone, G. Palermo, V. Zaccaria, C. Silvano, D. Melpignano, G. Haugou, and T. Lepley, "An exploration methodology for a customizable opencl stereo-matching application targeted to an industrial multi-cluster architecture," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2012.

[29] Khronos Group, "OpenCL Specification," 2012.