

Application Autotuning through Dynamic Knobs

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

Abstract—Several classes of applications expose a set of parameters that influence their extra-functional properties, such as the quality of the result or the size of the output. This leads the application designer to tune these parameters in order to find the configuration that produces the desired outcome. From the architectural point of view, the trend in modern systems is to expose a high level of parallelism, often involving heterogeneous resources. To exploit the full potential of the hardware, the application designer must take into account resource-related parameters in the tuning process. Since the requirements of the applications and the resources assigned to each application might change at runtime, we argue that finding the one-fit-all configuration is not a trivial operation. For this reason we use a framework that enhances an application with an adaptation layer in order to continuously tune the parameters of the application according to the evolving situation, in a best effort fashion.

I. INTRODUCTION

One of the main tasks of an application designer is to reach the required performance on the target system. Unfortunately, the performance of an application is seldom defined by one metric, such as the execution time or its throughput. The performance is instead composed of a collection of metrics that are usually in contrast between them; for instance the time spent on elaborating the input against the quality of the result or the power consumption. A common approach is to write an algorithm that exposes a set of parameters, also known as *dynamic knobs* [2], that influence the performance of the application, such as the number of trials in a Monte Carlo solver or the resolution of the output frame in a video encoder. The possible values of these parameters define the design space of the application and in literature are described several Design Space Exploration (DSE) techniques [4] to automatically and efficiently compute the Pareto set, which represents all the optimal trade-offs between the metrics of interest. Since the application requirements may change at runtime – for instance if the platform is at first powered by a battery, then plugged in a power supply – and the system might vary the resources allocated to the application as well, we argue that is not trivial to select an a-priori one-fit-all configuration. For this reason we rely on the ARGO¹ framework [1]: it is grounded on the Monitor-Analyze-Plan-Execute (MAPE) feedback loop [3] to automatically tune the application parameters according to the evolution of the system. The main idea of the framework is to exploit design time knowledge of the application, obtained through a DSE, to select the best configuration according to the actual application requirements and the observed performance, both of them composed of a collection of metrics of interest. ARGO is implemented as an external library to be linked to the target application. It takes autonomous decisions without interacting with any other element. For these reasons we are able to minimize the intrusiveness of the integration, expressed in terms of lines of code to be changed.

¹The 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); it aims at letting applications to reach their goals too.

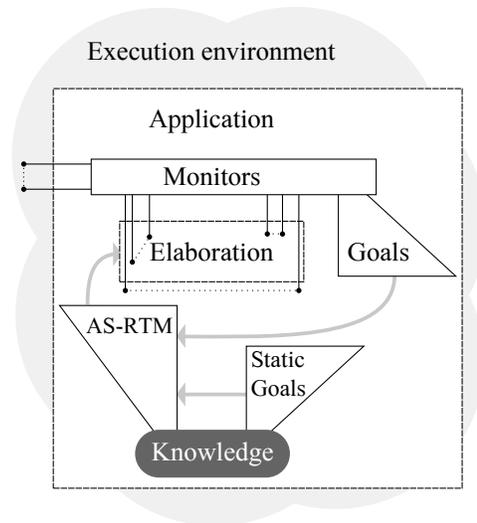


Figure 1. The framework structure. The AS-RTM selects the best configuration according to the runtime information provided by the monitors and the design-time knowledge.

II. FRAMEWORK STRUCTURE

The framework follows a modular approach, as showed in Figure 1. A monitor infrastructure is used to gather insight on the actual performance of the application (Monitor element). ARGO relies on a monitor suite to observe the most common metrics such as the resident set size of the virtual memory that the process is using, the process- and system-wide CPU usage or low-level metrics exposed by the widely adopted PAPI framework [6]. Moreover, to observe application-specific metrics, such as the quality-related ones, the object-oriented implementation enables the application designer to easily integrate a custom monitor, defining the methods that actually gather the new data. On the other side, ARGO embeds the design time knowledge in the list of Operating Points (OPs), where each OP represents a configuration and the performance reached by the application using that configuration. The framework is agnostic about the technique to be used to perform the DSE, in the current implementation it parses the MULTICUBE [5] syntax. The Application-Specific RunTime Manager is the main component of the framework that selects the best configuration (Plan element), within the list of OPs, according to a multi-objective constrained optimization that might involve observed metrics (using Goals) or design time computed metrics (using Static Goals). Since the dynamic knobs are heavily application-dependent, it is the application itself that is in charge of applying the configuration selected by ARGO (Execute element), closing the MAPE loop. In this way it is possible to deploy the framework in a wide range of applications, while minimizing the integration effort. In fact, we model the application as a sequence of different blocks of code that perform the elaboration iteratively. The idea is that at the beginning of each iteration, the application retrieves

```

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 }
    
```

Figure 2. This example shows how to integrate ARGO in an existing application code that exposes the elaboration as a loop.

the configuration to use in the current iteration. The runtime overhead introduced by ARGO every time the application asks a new configuration depends on the number of constraints and the magnitude of changes from the previous iteration. Thus, with 10 constraints and 1000 Operating Points it ranges from few microseconds to 2 milliseconds in its worst case scenario.

III. FRAMEWORK INTEGRATION

To employ the concept of separation of concerns, our workflow is based on three kinds of files. The source code of the application describes the functional behavior, while we use two configuration files written in XML to express the adaptation layer: one file describes the design time knowledge and the third one describes the monitor infrastructure and the multi-objective optimization. ARGO uses a tool that automatically generates the glue-code required to integrate the framework in the target application. To better clarify the required effort, Figure 2 provides an integration example considering a toy application. It shows the original source code written in black, while the integration code required to adopt ARGO is 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. Since the code of toy application exposes directly the elaboration loop, the integration requires only to include the created header file, to initialize the framework and then to wrap the execution call with the generated macro, highlighted in red bold. In this way the framework can observe and tune the elaboration block. Since no assumptions are made on the structure of the application code, the tool generates a hierarchy of methods to interact with the application, that requires to write the glue-code using more fine grained functions. In the worst case, the application designer is able to directly use the framework API.

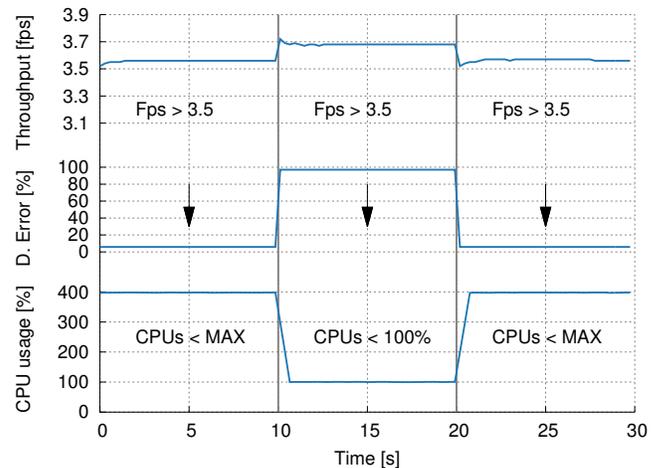


Figure 3. Execution trace of the application in the first use-case, showing the Throughput, the computational error (Disparity Error) and the CPU usage.

IV. USE CASE EXAMPLE

Let us consider the development of the Stereo Matching application, that computes a disparity map from two stereo-images of a scene. We are interested on monitoring three metrics: the application throughput, the computation error (Disparity Error) and the fraction of the CPU used. Suppose also that the application requirements have been set to maintain a throughput greater than 3.5fps , while minimizing the disparity error for the entire execution. To create the need of adaptation, within the period $10\text{s} - 20\text{s}$ the available resources have been reduced to only 1 core out of 4 to accommodate high priority tasks. To acquire runtime knowledge, the application monitors the throughput and the CPU usage of the application. The execution trace of the application is shown in Figure 3. 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 disparity error value equal to 97%.

REFERENCES

- [1] D. Gadioli, G. Palermo, and C. Silvano. Application autotuning to support runtime adaptivity in multicore architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*.
- [2] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *2011 ACM SIGPLAN Notices*.
- [3] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer, 2003*.
- [4] 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, 2009 IEEE Transactions on*.
- [5] 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*.
- [6] 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*.