SOCRATES - A Seamless Online Compiler and System Runtime AutoTuning Framework for Energy-Aware Applications

Davide Gadioli*, Ricardo Nobre+, Pedro Pinto+, Emanuele Vitali*
Amir H. Ashouri×, Gianluca Palermo*, Joao Cardoso+, Cristina Silvano*
*Politecnico di Milano, Italy ×University of Toronto, Canada +University of Porto, Portugal

***Abstract*—Configuring program parallelism and selecting optimal compiler options according to the underlying platform architecture is a difficult task. Tipically, this task is either assigned to the programmer or done by a standard one-fits-all policy generated by the compiler or runtime system. A runtime selection of the best configuration requires the insertion of a lot of glue code for profiling and runtime selection. This represents a programming wall for application developers. This paper presents a structured approach, called SOCRATES, based on an aspect-oriented language (LARA) and a runtime autotuner (mARGOt) to mitigate this problem. LARA has been used to hide the glue code insertion, thus separating the pure functional application description from extra-functional requirements. mARGOT has been used for the automatic selection of the best configuration according to the runtime evolution of the application.** [1]

I. Introduction

Performance portability across different computing platforms is a challenging problem for application developers working on different computing fields from embedded to HPC systems. The problem is that application performance is strongly dependent on the underlying target platform, system runtime, and input data. Ideally, the solution can be expressed as a *morphable* code capable of adapting to the environment conditions. However, this approach faces several challenging problems not yet solved. Among them, we can mention that writing such a kind of code would require a flexible high-level language capable of expressing functional aspects, that can be easily manipulated and customized for later compilation and code generation phases.

In the past, customizing applications without a complete rewriting of the code, in terms of parallelism and compiler transformations, has been envisioned as a promising path [1], [7]. These approaches are typically based on the tuning of the application, compiler and system runtime knobs before the actual code deployment, thus finding a one-fits-all configuration for the target platform. However, selecting the most suitable configuration can be a hard task, if we consider that the application workload and resource partitioning change dynamically and the energy/power budget can evolve depending on external events. Only a few recent efforts (see, e.g., [6], [8], [13]) are applying strategies once the application has been deployed on the target system. The main problem of runtime solutions for application tuning is that they require a high-level of intrusiveness in the source code. Indeed, the original source code implementing the functional aspects should be enhanced with glue code needed to profile, monitor and configure the application according to extra-functional aspects.

This paper introduces a structured approach, called SOCRATES, for the runtime selection of the most suitable application configuration in terms of compiler flags and parallelism parameters of the OpenMP runtime. The main contribution of SOCRATES is to offer runtime autotuning features while avoiding any manual intervention by the application developer. SOCRATES uses an aspect-oriented language, LARA [3], to implement the separation of concerns between the functional and extra-functional parts of the application, while an application-level autotuner, mARGOt [6], is integrated for the optimal configuration selection. All changes to the application code required by SOCRATES are automatically performed by LARA. Furthermore, SOCRATES supports an energy efficient execution by introducing energy consumption as a key variable to be considered at runtime.

II. Proposed Methodology

The main goal of the SOCRATES framework is to provide to the application developer an energy-aware framework to enhance the application with a kernel-level compiler autotuning and adaptation layer in a seamless way. In particular, the starting point of the approach is a generic source code that describes the functional behavior of the application, i.e $o = f(i)$ where a generic function f computes the desired output o from the given input i. To reach the adaptivity goal, the framework performs two major actions on the original application. The first action consists of transforming the application into a *tunable* version, enhancing its structure to take as input a set of knobs $(k_1, k_2, \ldots, k_n)$ that affect its behavior, i.e. $o = f(i, k_1, k_2, \ldots, k_n)$. The idea is that a change in the configuration of the knobs results in a change of the extra-functional property (EFP) of the application f and its output o. Examples of EFPs of the function f might be execution time and power consumption, while EFPs of the output o might be solution accuracy and output file size. The second action consists of enhancing the tunable version of the application with the intelligence needed to configure dynamically its knobs according to application requirements and environment conditions. Thus, the application is enhanced with an adaptation layer that provides the ability to monitor its behavior and select the most suitable configuration. Even if the overall approach is suitable for different contexts, SOCRATES has been designed to address the following autotuning space:

[1]This work has been partially funded by the EU H2020-FET-HPC program under the project ANTAREX - AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems (grant number 671623).

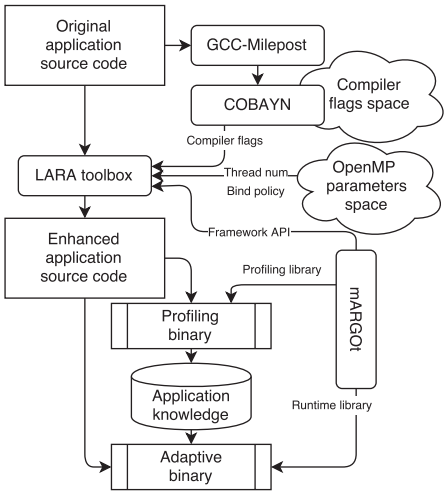

Figure 1. Tool flow of the SOCRATES approach from the original application source code to the generation of the application adaptive binary.

- *Compiler Options (CO)*: This knob represents a combination of compiler flags. We used four standard optimization levels from gcc: `Os`, `O1`, `O2`, `O3`, in addition to specific transformations such as: *-funsafe-math-optimizations*, *-fno-guess-branch-probability*, *-fno-ivopts*, *-fno-tree-loop-optimize*, *-fno-inline-functions*, *-funroll-all-loops* derived from [4];
- *Number of threads (TN)*: This knob sets the number of OpenMP threads between 1 and the number of logical cores;
- *Binding Policy (BP)*: This knob sets the OpenMP binding policy: `spread` or `close`. We set the environmental variable OMP_PLACES to `cores`.

Figure 1 shows the SOCRATES toolchain. The proposed methodology targets applications with one or more kernels representing different phases of the computation. To reduce the compiler space, every kernel of the original code is analyzed and code features are extracted by GCC-Milepost [5]. Then, the compiler autotuning framework COBAYN [2] is used to infer and extract the most promising compiler flags for every kernel. We generated several versions of the kernel, according to the autotuning space by using a LARA-controllable toolbox, while the code is enhanced with runtime autotuning capability by mARGOt. The enhanced code is then profiled for all the alternatives to create the *application knowledge* required by the final adaptive application binary.

Reducing the compiler space complexity. The first step of SOCRATES consists of pruning the compiler optimization space. An appropriate methodology is to select efficiently the most promising compiler options given a target application. To this end, we adopted the COBAYN framework to select the best optimization passes. COBAYN is an autotuning framework that identifies the most suitable compiler optimizations by using Bayesian Networks (BN). It uses application characterization to induce a prediction distribution by an iterative compilation methodology. This technique identifies a suitable set of compiler optimizations to be applied to the target kernel, thus reducing the cost of the compiler optimization phase.

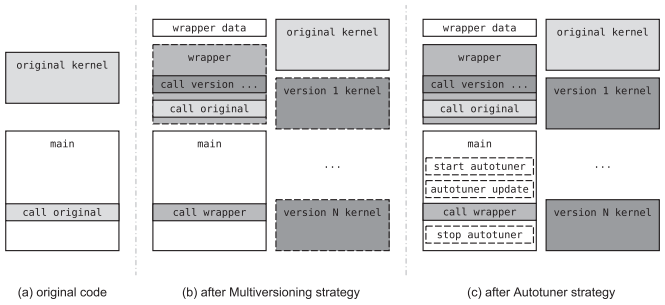

Figure 2. Example of the automatic application code transformation from the original code (a) to the final adaptive code (c).

We used GCC standard optimization levels and COBAYN predictions as reduced design space for the compiler flags. Application characterization is done by extracting static code features by GCC-Milepost, while COBAYN has been adapted to work at kernel function granularity. In SOCRATES, we used the compiler space adopted in the original COBAYN paper (128 flags combination) by reducing it to four alternatives.

Runtime configuration selection. The objective of SOCRATES is to tune dynamically the target application according to the system evolution. We used the mARGOt autotuning framework to select dynamically the most suitable configuration based on the classical MAPE-K loop [9] for autonomous systems. mARGOt is a dynamic autotuner where the definition of application requirements might change at runtime (e.g. considering different energy requirements) and adaptivity layer changes the configuration accordingly. The intrusiveness of mARGOt in the application code is limited to an initialization call in the application and to *start*/*stop*/*update* calls around the regions of interest. mARGOt is partitioned into two modules: the monitoring infrastructure and the Application-Specific Run-Time Manager (AS-RTM). The monitoring gathers insight on the actual behavior of the target kernel and execution environment. The AS-RTM is in charge of selecting the most suitable configuration at runtime based on three types of information: *i)* application requirements; *ii)* design-time knowledge of the kernel (i.e. profiling information) and *iii)* feedback information collected from monitors. The application requirements are defined as a constrained multi-objective optimization problem, thus the application designer is able to define the most suitable configuration defining only extra-functional objectives and constraints.

Integration issues. LARA strategies are used to enhance automatically the original source code for making the application tunable and to integrate the mARGOt framework. In particular, we use code transformation and code insertion strategies specified in LARA aspects to interact with the application source code. MANET [11] is used as source-to-source compiler to weave the cross-cutting concerns described in the aspects in C applications.

There are two main strategies: *Multiversioning* and *Autotuner*. Figure 2 shows an example of how the application

code evolves during the entire process: from pure functional code to adaptive code, ready to be deployed. The first strategy, *Multiversioning*, generates different versions of the target kernel and a mechanism to choose which version to call at runtime. The autotuning space is composed of GCC compiler flags, binding policy and number of OpenMP threads. The first two parameters must be statically defined, while the number of OpenMP threads can be controlled dynamically. The first action of the *Multiversioning* strategy clones the kernel several times. Each function clone represents a different version of the kernel in terms of compiler options and binding strategy. No cloned versions have been generated to manage the number of threads variable because it does not require to be known at compile time. For each function clone, the strategy inserts GCC pragmas to set compilation flags (e.g., `#pragma GCC optimize ("O2,no-inline"))` and OpenMP pragmas (e.g., `#pragma omp for num_threads(NT) proc_bind(close))` to configure the parallelization of the kernels. The strategy also generates a wrapper, which selects the target version of the kernel, according to control variables. Afterwards, the strategy replaces each call of the kernel from application source files, with a call to the wrapper (see Figure 2b). The entire process is fully automated. The second strategy, *Autotuner*, is responsible for integrating mARGOt into the application. First, the connection between the generated kernel versions and the autotuner is done by exposing variables containing the current configuration. Then, the strategy inserts the required headers and the initialization function call at the `main` function. Finally, as shown in Figure 2c, the strategy surrounds the call to the wrapper with the mARGOt API code to monitor EFPs and to update the most suitable configuration.

III. Experimental Results

The platform used for the experiment is a NUMA machine with two Intel Xeon E5-2630 V3 CPUs for a total of 16 cores with hyperthreading enabled and 128 GB of DDR4 memory (@1866 MHz). The experimental campaign is based on 12 apps from the Polybench/C benchmark suite [12]. We used SOCRATES framework to automatically generate the additional code without any manual intervention on the target applications. In the experimental campaign, we considered the autotuning space presented in Section II. We used mARGOt to perform two tasks. The first one to profile the application to perform a Design Space Exploration (DSE) and build the knowledge required by the autotuner. The second task to tune the application at runtime according to application requirements given by the experiment. In this work, we used a full-factorial analysis over the design space, however our approach is agnostic with respect to the used DSE strategy.

Table I presents some metrics regarding the developed strategy and its application to each benchmark code. *Att* is the number of attributes checked in the LARA strategy about the source code of the application, including function signature information and OpenMP pragma information. *Act* is the number of actions performed on the code, including code insertions, cloning and pragma insertion. The *LOC* columns represent, in order, the number of logical lines of code of the original (*O-*) benchmark, the weaved (*W-*) benchmark and their difference (*D-*). The number of logical lines of source code in the complete LARA strategy is 265. This is used to calculate the *Bloat* metric [10], that roughly estimates how much code is weaved in the original application per line of code in the aspect files. These data present an overview of how complicated, time-consuming and error-prone it would be to execute these tasks manually. For instance, take the case of *2mm*, on the first row. The weaver automatically inspects multiple points in the program code, checking the value of 698 attributes and performs transformations (or insertions) on 378 of the inspected points. The resulting code has a number of logical lines of code that is an order of magnitude larger than the original one. From the *Bloat* value for *2mm*, we can see that, on average, we insert 7.29 lines of C code per line of LARA aspect code. The large differences from benchmark to benchmark are explained because their kernels may be very different in size and have different numbers of loops, which are closely related to the number of lines of code and actions performed, respectively.

Table I
Metrics collected from the application of LARA strategies.

Benchmark	Att	Act	O-LOC	W-LOC	D-LOC	Bloat
2mm	698	378	136	2068	1932	7.29
3mm	708	378	125	1801	1676	6.32
atax	684	250	81	1071	990	3.74
correlation	1347	410	138	2366	2228	8.41
doitgen	561	218	72	1018	946	3.57
gemver	631	218	94	1008	914	3.45
jacobi-2d	4429	154	145	2918	2773	10.46
mvt	339	154	64	571	507	1.91
nussinov	551	154	78	1356	1278	4.82
seidel-2d	445	154	47	565	518	1.95
syr2k	376	186	66	749	683	2.58
syrk	370	186	62	743	681	2.57
Average	928	237	92	1353	1261	4.10

The experiment shown in Figure 3 analyzes the trade-off space between power consumption and throughput of the target kernels by using a full-factorial DSE. In particular, it shows the distribution (as boxplot) between the throughput and the average power consumption. The values on the y-axis represent the distribution of the target metrics, for each evaluated application, considering only the Pareto-optimal configurations. Given the large power/performance swing, there is no one-fits-all configuration, thus confirming the importance of the proposed approach.

The experiment shown in Figure 4 assesses the benefits of the proposed approach when autotuning is done statically (compile-time) according to a given power budget. It shows the results in terms of execution time and selected configuration (y-axis), while changing the available power budget, for the target application (2mm). The plot shows the power-performance trade-off available in the Pareto curve and also highlight that there is not a clear trend on

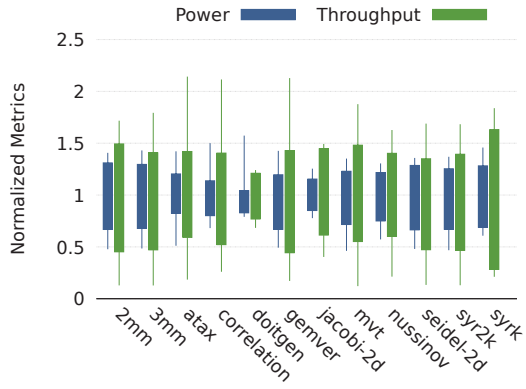

Figure 3. Power/Throughput distribution over the Pareto curve.

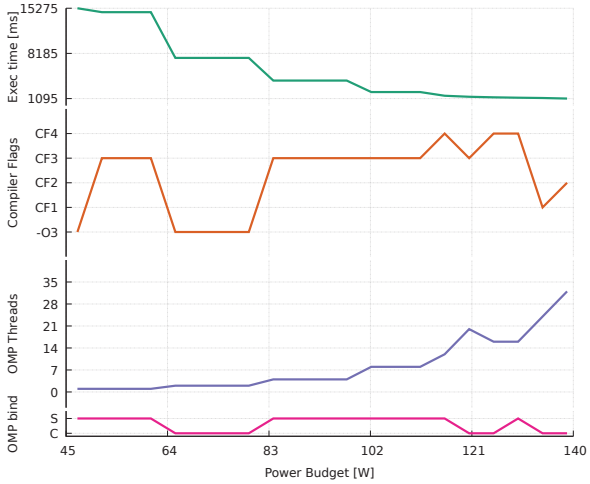

Figure 4. Static analysis of the proposed approach, that aims at minimizing execution time given a constraint on power budget (x-axis).

the selected software-knobs. For this experiment, the custom flag combinations suggested by COBAYN (*CF1-CF4*) are: CF1) *O3, no-guess-branch-probability, no-ivopts, no-tree-loop-optimize, no-inline*; CF2) *O2, no-inline,unroll-all-loops*; CF3) *O2, unsafe-math-optimizations, no-ivopts, no-tree-loop-optimize, unroll-all-loops*; CF4) *O2, no-inline*.

The last experiment shows the runtime effectiveness of SOCRATES. Figure 5 reports an execution trace of the target application (*2mm*), when the requirement changes from an energy-efficient policy optimizing Throughput per Watt^2 (Thr/W^2) – in the 0s-100s interval – to a performance-oriented policy optimizing the Throughput – 100s-200s interval – and back to optimizing Thr/W^2 – 200s-300s interval. When changing from an energy-aware to a performance-oriented policy (and viceversa), we can notice how the parameter sets change dynamically to meet the requirements.

IV. Conclusions

The paper introduced the SOCRATES framework to select application parallelism and compiler options at runtime.

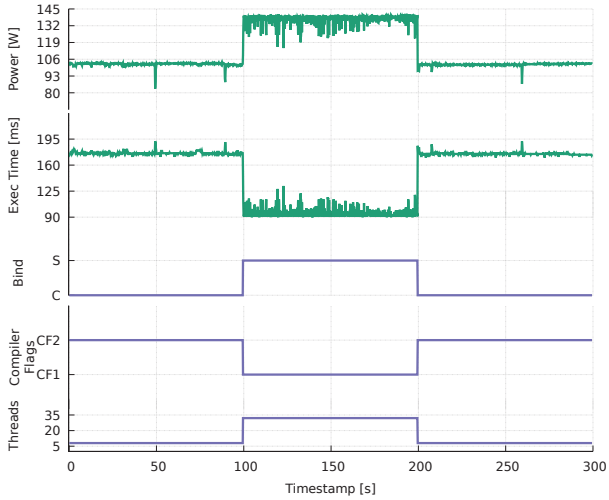

Figure 5. Execution trace of the 2mm application by varying application requirements at runtime.

SOCRATES is based on the mARGOt autotuner and the LARA aspect-oriented language and its main contribution consists of reaching this goal avoiding any manual intervention by the application developer. SOCRATES has been applied to the OpenMP Polybench suite by varying application requirements. Reported results show how SOCRATES can reach significant benefits in terms of exploiting runtime energy-performance trade-offs in a dynamic environment.

References

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Programming Language Design and Implementation*, 2009.

[2] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim.*, 13(2):21:1–21:25, 2016.

[3] J. M. P. Cardoso, J. G. F. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, and F. Gonçalves. Performance-driven Instrumentation and Mapping Strategies Using the LARA Aspect-oriented Programming Approach. *Softw. Pract. Exper.*, 2016.

[4] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu. Deconstructing iterative optimization. *ACM Trans. Archit. Code Optim.*, 9(3):21:1–21:30, Oct. 2012.

[5] G. Fursin et al. Milepost-gcc: Machine learning enabled self-tuning compiler. *Intern, Journal of Parallel Programming*, 2011.

[6] D. Gadioli, G. Palermo, and C. Silvano. Application autotuning to support runtime adaptivity in multicore architectures. In *Embedded Computer Systems: Architectures, Modeling and Simulation*, 2015.

[7] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *IEEE IPDPS*, 2009.

[8] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, 2011.

[9] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1), 2003.

[10] C. V. Lopes and G. Kiczales. *D: A language framework for distributed programming*. PhD thesis, Northeastern University, 1997.

[11] P. Pinto, R. Abreu, and J. M. P. Cardoso. Fault Detection in C Programs using Monitoring of Range Values: Preliminary Results. *ArXiv*, 2015.

[12] L.-N. Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench*.

[13] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali. Proactive control of approximate programs. *ACM SIGOPS Operating Systems Review*, 2016.