Automatic Software Tuning of Parallel Programs for Energy-Aware Executions

Sébastien Varrette[1,2(✉)], Frédéric Pinel[1(✉)], Emmanuel Kieffer[1(✉)], Grégoire Danoy[1,2(✉)], and Pascal Bouvry[1,2(✉)]

[1] Department of Computer Science (DCS), University of Luxembourg, Luxembourg City, Luxembourg
{sebastien.varrette,frederic.pinel,emmanuel.kieffer gregoire.danoy,pascal.bouvry}@uni.lu
[2] Interdisciplinary Centre for Security Reliability and Trust (SnT), University of Luxembourg, Luxembourg City, Luxembourg

Abstract. For large scale systems, such as data centers, energy efficiency has proven to be key for reducing capital, operational expenses and environmental impact. Power drainage of a system is closely related to the type and characteristics of workload that the device is running. For this reason, this paper presents an automatic software tuning method for parallel program generation able to adapt and exploit the hardware features available on a target computing system such as an HPC facility or a cloud system in a better way than traditional compiler infrastructures. We propose a search based approach combining both exact methods and approximated heuristics evolving programs in order to find optimized configurations relying on an ever-increasing number of tunable knobs i.e., code transformation and execution options (such as the number of OpenMP threads and/or the CPU frequency settings). The main objective is to outperform the configurations generated by traditional compiling infrastructures for selected KPIs i.e., performance, energy and power usage (for both for the CPU and DRAM), as well as the runtime. First experimental results tied to the local optimization phase of the proposed framework are encouraging, demonstrating between 8% and 41% improvement for all considered metrics on a reference benchmarking application (i.e., Linpack). This brings novel perspectives for the global optimization step currently under investigation within the presented framework, with the ambition to pave the way toward automatic tuning of energy-aware applications beyond the performance of the current state-of-the-art compiler infrastructures.

Keywords: HPC · Performance evaluation · Energy efficiency · Compiler infrastructure · Automatic tuning · MOEA · Hyper-parameter optimization

1 Introduction

With the advent of the Cloud Computing (CC) paradigm, the last decade has seen massive investments in large-scale High Performance Computing (HPC) and

R. Wyrzykowski et al. (Eds.): PPAM 2019, LNCS 12044, pp. 144–155, 2020.
https://doi.org/10.1007/978-3-030-43222-5_13

storage systems aiming at hosting the surging demand for processing and data-analytic capabilities. The integration of these systems in our daily life has never been so tied, with native access enabled within our laptops, mobile phones or smart Artificial Intelligence (AI) voice assistants. Outside the continuous expansion of the supporting infrastructures performed in the private sector to sustain their economic development, HPC is established as a strategic priority in the public sector for most countries and governments. For large scale systems, energy efficiency has proven to be key for reducing all kinds of costs related to capital, operational expenses and environmental impact. A brief overview of the latest Top 500 list (Nov. 2019) provides a concrete indication of the current power consumption in such large-scale systems and projections for the Exaflop machines foreseen by 2021 with a *revised* power envelop of 40 MW. Reaching this target involves combined solutions mixing hardware, middleware and software improvements, when power drainage of a system is closely related to the type and characteristics of the workload. While many computing systems remain heterogeneous with the increased availability of accelerated systems in HPC centers and the renewed global interest for AI methods, the energy efficiency challenge is rendered more complex by the fact that pure performance and resource usage optimization are also Key Performance Indicators (KPIs). In this context, this paper aims at extending HPC middleware into a framework able to transparently address the runtime adaptation of execution optimizing priority KPIs i.e., performance, energy and power usage (for both the CPU and DRAM), as well as the runtime in an attempt to solve the following question: *Can we produce energy-aware HPC workloads through source code evolution on heterogeneous HPC resources?* To that end, we propose EvoCode, an automatic software tuning method for parallel program generation able to better exploit hardware features available on a target computing system such as an HPC facility or a cloud system.

This paper is organized as follows: Sect. 2 details the background of this work and reviews related works. Then, the EvoCode framework is presented in Sect. 3. Implementation details of the framework, as well as the first experimental results validating the approach, are expounded in Sect. 4. Based on a reference benchmarking application (i.e., Linpack, measuring a system's floating point computing power), the initial hyper-parameter optimization phase already demonstrate concrete KPIs improvements with **8%** performance and runtime gains, up to **19%** energy and power savings and even **41%** of energy and power usage decrease at the DRAM level. Finally, Sect. 5 concludes the article and provides future directions and perspectives.

2 Context and Motivations

Recent hardware developments support energy management at various levels and allow for the dynamic scaling of the power (or frequency) for both the CPU and Memory through integrated techniques such as Dynamic Voltage and Frequency Scaling (DVFS) able also to handle idle states. These embedded sensors

permit recent hardware to measure energy and performance metrics at a fine grain, aggregating instruction-level measurements to offer an accurate report of code region or process-level contributions. This can be done through *low-level* power measurement interfaces such as **Intel's Running Average Power Limit (RAPL)** interface. Introduced in 2011 as part of the SandyBridge micro-architecture, RAPL is an advanced power-capping infrastructure which allows the users (or the operating system) to specify maximum power limits on processor packages and DRAM. This allows a monitoring and control program to dynamically limit the maximum average power, such that the processor can run at the highest possible speed while automatically throttling back to stay within the expected power and cooling budget. To respect these power limits, the awareness of the current power usage is required. Direct measures being often unfeasible at the processor level, power estimates are performed within a model exploiting performance counters and temperature sensors, among others. These estimations are made available to the user via a Model Specific Register (MSR) or specific daemons which can be used when characterizing workloads. Thus RAPL energy results provide a convenient interface to collect feedback when optimizing code for a diverse range of modern computing systems. This allows for unprecedented easy access to energy information when designing and optimizing energy-aware code. Moreover, on the most recent hardware architectures and DRAMs, it was demonstrated that RAPL readings are providing accurate results with negligible performance overhead [4,7]. Finally, it is also worth to note that non-Intel processors such as the recent AMD architectures (Ryzen, Epyc) also expose RAPL interfaces which can be used via the AMD μProf performance analysis tool.

At the NVIDIA GPU accelerator level, A C-based API called **NVidia Management Library (NVML)** permits to monitor and manage various states of GPU cards. The runtime version of NVML is embedded with the NVIDIA display driver, and direct access to the queries and commands are exposed via `nvidia-smi`.

In all cases, these fine-grained interfaces (i.e., RAPL, NVML...) are used in general-purpose tools able to collect low level performance metrics. Table 1 reviews the main performance analysis tools embedding Hardware counter measurement able to report fine-grain power measurements, as well as global profiling suites that eventually build on top of these low-level hardware counter interfaces.

Optimization and Auto-Tuning of Parallel Programs. Optimizing parallel programs becomes increasingly difficult with the rising complexity of modern parallel architectures. On the one hand, parallel algorithmic improvement requires a deep understanding of performance bottleneck to tune the code application with the objective to run optimally on high-end machines. This assumes a complete workflow of performance engineering of effective scientific applications (based on standard MPI, OpenMP, an hybrid combination of both or accelerators frameworks), including instrumentation, measurement (i.e., profiling and tracing, timing and hardware counters), data storage, analysis, and visualization. Table 1 presents the main performance and profiling analysis tools sustaining this complete workflow.

Table 1. Main performance analysis tools embedding hardware counter measurement for fine-grained power monitoring.

Name	Version	Description
Low-level performance analysis tools		
Perf	4.10	Main interface in the Linux kernel and a corresponding user-space tool to measure hardware counters
PAPI	5.7.0	Performance Application Programming Interface
LikWid	5.0.1	CLI applications & API to measure hardware events
Generic performance and profiling analysis tools		
ARM Forge/Perf. Report	20.0	Profiling and Debugging for C, C++, and Fortran High Performance code
TAU	2.29	Tuning & Analysis Utilities to instrument code
Score-P	6.0	A Scalable Perf. Measurement Infra. for Parallel Codes
HPC-Toolkit	2018.09	Integrated suite of tools/performance analysis of optimized parallel programs

However, none of these tools embed automatic software tuning solutions. To that end, *Auto-tuning* [10] arose as an attempt to better exploit hardware features by automatically tuning applications to run optimally on a given high-end machine. An auto-tuner tries to find promising *configurations* for a given program executing on a given target platform to influence the non-functional behavior of this program such as runtime, energy consumption or resource usage. A configuration can be created by applying code changes to a program, also called **static** *tunable knobs* or *code transformations*. Alternatively, **runtime** tuning knobs such as the number of threads, the affinity or mapping of threads onto physical processors, or the frequency at which the cores are clocked can be adapted. The literature offers numerous studies dedicated to the optimization of the runtime knobs, much less on the code transformation exploration since this requires the use of advanced compiler infrastructures such as LLVM [9]. Furthermore, optimization is often limited to a single criteria such as runtime, while it is desirable to improve multiple objectives simultaneously which is more difficult as criteria may be conflicting. For instance, optimizing for performance may reduce energy efficiency and vice versa. In all cases, the ever-increasing number of tunable knobs (both static or runtime), coupled with the rise and complexity escalation of HPC applications, lead to an *intractable and prohibitively large search space* since the order of the code transformations applied matters. This explains why a wider adoption of auto-tuning systems to optimize real world applications is still far from reality and all modern compilers such as GCC or LLVM rely on static heuristics known to produce

good results on average but may even impede performances in some cases. The huge search space induced by the quest of optimal sequences of tunable knobs for each region composing a given source code represents a severe challenge for intelligent systems. Two approaches are traditionally considered: (1) Machine Learning (ML) [1], recognized to speed up the time required to tune applications but is too dependent on rare training data and thus fail to guarantee the finding (local and global) of optimal solutions. (2) Iterative search-based methods, relying on exact or approximated (i.e., Evolutionary Algorithm (EA) inspired) heuristics. Identified as computationally expensive [6], this approach mainly targets performance or execution time optimization. Moreover, their suitability for a specific application depends on the shape of its associated search space of possible configurations. Nevertheless, search-based approaches remain the most promising and effective ones despite their identified concerns. To optimize simultaneously multiple objectives i.e., performance, runtime, energy and power usage (for both the CPU and DRAM), while minimising the time consuming evaluations of the objective vector on the target computing system, we propose EvoCode, a search-based framework for automatic tuning of parallel programs which permits to evolve a given source code to produce optimized energy-aware versions. Combining both exact and approximated heuristics in a two-stage Multi-Objective Evolutionary Algorithm (MOEA) optimization phase relying on the LLVM Compiler Infrastructure, the proposed approach is detailed in the next section.

3 Toward Automatic Software Tuning of Parallel Programs for Energy-Aware Executions

An overview of the EvoCode framework is proposed in Fig. 1 and is now depicted. It aims at tuning an input program denoted as $\mathcal{P}_{ref}$ for an optimized execution on a target computing system such as an HPC facility or a cloud system. "*Optimized*" in this context means the generation of semantically equivalent programs $\mathcal{P}_1, \mathcal{P}_2, ...$ demonstrating improvement for selected KPIs i.e., performance, energy and power usage (both for the CPU and DRAM), as well as runtime. In practice, we assume that $\mathcal{P}_{ref}$ is composed of multiple regions $\{R_1, ..., R_r\}$, where each region delimits a single-entry single-exit section of the source code subjected to tuning. For instance, an OpenMP section, an outermost loop within a loop nest, or a function definition associated to a function call (i.e., at least the `main()` function). The identification and analysis of these regions in $\mathcal{P}_{ref}$ (eventually to isolate code portions that can be ignored) corresponds to the **Step A and B** in EvoCode. Note that some regions may exist as CUDA kernels for hybrid (CPU+GPU) runs. Then in **Step C**, EvoCode will operate a two-stage MOEA optimization phase aiming at the automatic evolution of P_{ref} as follows: (1) a *local* optimization is achieved aiming at the best configuration selection for each region $R_{i,j}$. Typically, a categorical hyper-parameter optimization for the foreseen tunable knobs is performed for the selected KPIs leading to different versions of these regions. (2) A *global* MOEA combines the regions to measure the effect on the entire program instead of considering the effect only

for individual region executions. In this way, we optimize the whole program execution instead of focusing on specific regions, to provide semantically equivalent programs $\mathcal{P}_1, \mathcal{P}_2, ...$ based on approximated Pareto-optimal solutions. In practice, new multi-objective surrogate-based approaches [8] hybridizing multi-objective meta-heuristics (e.g., NSGA-III [3]) and Machine Learning models based on Gaussian Processes are proposed to minimize the time consuming evaluations of the objective vector on the target computing system. More precisely, configurations are evaluated using surrogate versions of the objectives functions handled by an oracle. If the prediction error ε is smaller than a predefined threshold, the oracle will consider that evaluations are accurate (and thus do not need to be executed in the target system), else it will update the surrogate models with the true objectives values, obtained from the running evaluation of the selected configurations. After the Pareto set for the whole program is computed, a set of code configurations for the entire program can be selected from the Pareto set, either manually or automatically to allow for the **Step D** of EvoCode, to help for the decision making phase. For the initial developments of EvoCode, preferences rankings provided from the decision-maker (i.e., to avoid providing specific weight values between the objective functions) will be used as proposed in [2], where the pruning method restricts the considered solutions within the

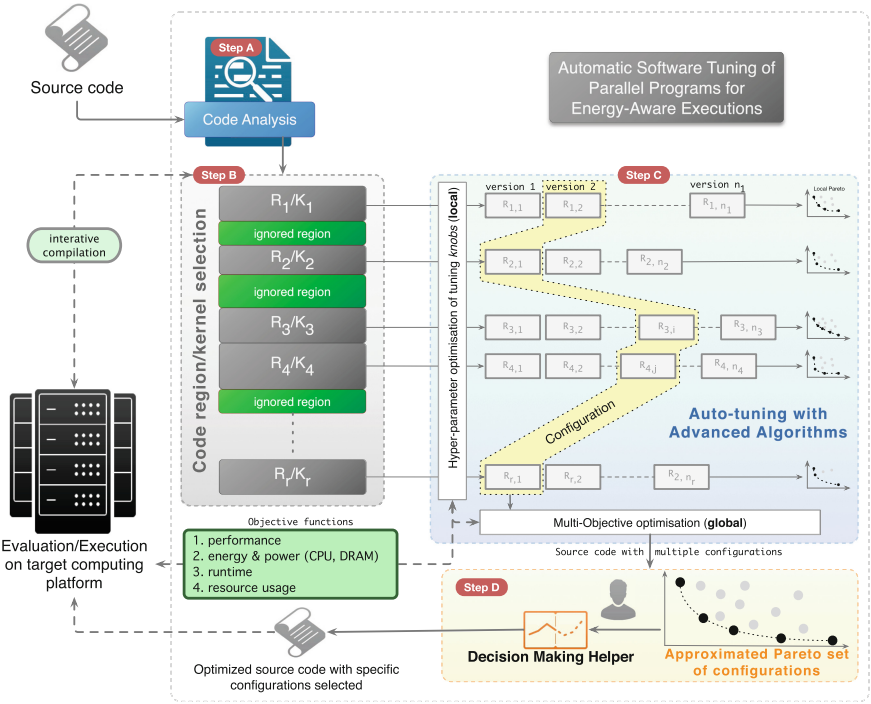

Fig. 1. Overview of the EvoCode framework.

Pareto set using threshold boundaries (e.g., afford for 10% performance penalties while reducing by at least 25% of the energy). Other heuristics relying on clustering methods i.e., grouping solutions with similar characteristics, will be then considered to improve the decision making process.

4 Validation and First Experimental Results

This section presents the first experimental results obtained by the EvoCode framework implemented as a dedicated Python module. The technical details of the environment supporting this implementation are provided in the Table 2. The experiments detailed in the sequel were conducted on the HPC facility of the University of Luxembourg [11], more precisely on the "*regular*" computing nodes of the `iris` cluster, each featuring 2 Intel Skylake Xeon Gold 6132 processors (totalling 28 cores, 2.6 GHz per node). For this reason, it was important to favor libraries able to scale and exploit effectively these parallel resources. For instance, the choice of DEAP was motivated by the fact that this framework works in perfect harmony with parallelisation mechanism such as multiprocessing and SCOOP. Then the application of the static tunable knobs (the only ones considered at this early stage of developments) was done through the LLVM compiler infrastructure. In practice, EvoCode exploits the flexibility offered by this suite to represent each program and source code from its LLVM bytecode or Internal Representation (IR) obtained using the appropriate front-end i.e., Clang for programs written in the C language family (mainly C, C++, OpenCL and CUDA). In particular, EvoCode takes as input the reference IR representation $\mathcal{I}^{ref}$ of the program P_{ref} to optimize, and the target computing system expected to run the derived programs (for instance the Skylake nodes of the `iris` cluster in this section). The static tunable knobs are sequences of codes transformations i.e., LLVM transformation *passes*. 54 such passes exist

Table 2. Libraries and components details part of EvoCode implementation.

Component	Version	Description
Python	3.7.4	n/a
NumPy	1.17.4	Fundamental package for scientific computing in Python
DEAP	1.3.0	Distributed Evolutionary Algorithms in Python
Optuna	0.19.0	Define-by-Run Hyperparameter Optimization Framework
Pandas	0.25.3	Python Data Analysis Library
plotly	4.4.1	Data Analytic Visualization Framework
LLVM/ Clang	8.0.0	LLVM Compiler Infrastructure and its front-end for the C language family (C, C++, OpenCL, CUDA...)
LikWid	5.0.0	Performance monitoring suite for RAPL hardware counter

Table 3. LikWid-based objective values reported on EvoCode individuals evaluation.

Metric name	Counter	Event	Description
`perf` [MFlops]	n/a	n/a	Program result (Ex: Linpack)
`runtime` [s]	n/a	n/a	`time`: Runtime (RDTSC)
`energy` [J]	`PWR0`	`PWR_PKG_ENERGY`	RAPL Energy contribution
`power` [W]	`PWR0`	`PWR_PKG_ENERGY/time`	RAPL Power contribution
`dram_energy` [J]	`PWR3`	`PWR_DRAM_ENERGY`	DRAM Energy contribution
`dram_power` [W]	`PWR3`	`PWR_DRAM_ENERGY/time`	DRAM Power contribution

on the considered version, and examples of such transformations include `dce` (Dead Code Elimination), `dse` (Dead Store Elimination), `loop-reduce` (Loop Strength Reduction), `loop-unroll` (Unroll loops) or `sroa` (Scalar Replacement of Aggregates). It follows that an individual $\mathcal{I}_i$ in EvoCode corresponds to the ordered sequence of applied transformations and the resulting LLVM bytecode obtained using the LLVM optimizer `opt` to apply the transforms on the reference IR code $\mathcal{I}^{ref}$ or sub-part of it i.e., the identified code regions. The **generation** of the individuals, either in the local or the global phase, consists then in aggregating the regions, compiling the LLVM bytecode into an assembly language specific to the target computing architecture using the LLVM static compiler `llc`, before producing the final binary from the linking phase using the LLVM front-end i.e., `clang`. The program $\mathcal{P}_i$ is normally *semantically* equivalent to P_{ref} since built from, and validated against, the reference $\mathcal{I}^{ref}$. Checking this equivalence is left outside the scope of EvoCode which only validates the viability of the generated individuals from the fact that (1) the generation is successful, (2) the produced binary executes successfully on the target platform and (3) the outputs of the execution on a pre-defined set of random inputs (common to all individuals and initiated in the Step A) are equal to the ones produced upon invocation of the reference program. Then the time consuming **evaluation** of an individual consists in running and monitoring the hardware counters attached to the generated binary execution on the target platform. The energy metrics are collected from the `ENERGY` performance group of LikWid which supports the `PWR_{PKG,PP0,PP1,DRAM,PLATFORM}_ENERGY` energy counter from the RAPL interface on the Intel Skylake and Broadwell micro-architecture present on the considered computing platform. In particular, the reported fitness values are composed by a vector of the metrics presented in the Table 3, more precisely on the mean values obtained from at least 20 runs. The validation proposed in this section was performed against a set of reference benchmarking applications i.e., the C version of **Linpack** [5], STREAM (the industry standard for measuring node-level sustained memory bandwidth) or FFT. For the sake of simplicity and space savings, only the results tied to the reference Linpack benchmarks are now presented. The focus of this study was *not* to maximize the benchmark results, but to set a common input parameter set enabling fast evaluations for *all* individuals. The Linpack source code (in its C version) is structured in

13 functions, used as regions $R_1, \ldots, R_{13}$ optimized by the **local** optimization phase of EvoCode. An hyperparameter optimization is performed for each of these regions, and the complete program is for the moment rebuilt from the best configurations obtained for each region when it is planned for EvoCode to perform the **global** MOEA-based optimization phase to rebuilt the program. Thus the results presented in this paper focus on the sole local optimization phase. The Fig. 2 presents the optimization history of all trials in the EvoCode Hyper-parameter study for the `perf,runtime,energy,power,dram_energy` and `dram_power` metrics upon reconstruction of the full program from the individual region evolution. The Table 4 summarizes the best results obtained from the EvoCode auto-tuning, demonstrating improvement obtained for all criteria i.e., performance, runtime, energy, power, DRAM energy and DRAM power metrics. The improvement obtained at the DRAM level are quite astonishing (demonstrating up to 41% of energy and power savings), but the associated contribution in the energy and power dissipation is relatively small. For more classical metrics, the auto-tuning performed by EvoCode still exhibits 8% performance and runtime improvement, and up to 19% energy and power savings. This demonstrates quite significant gains, especially when considering that these results have been

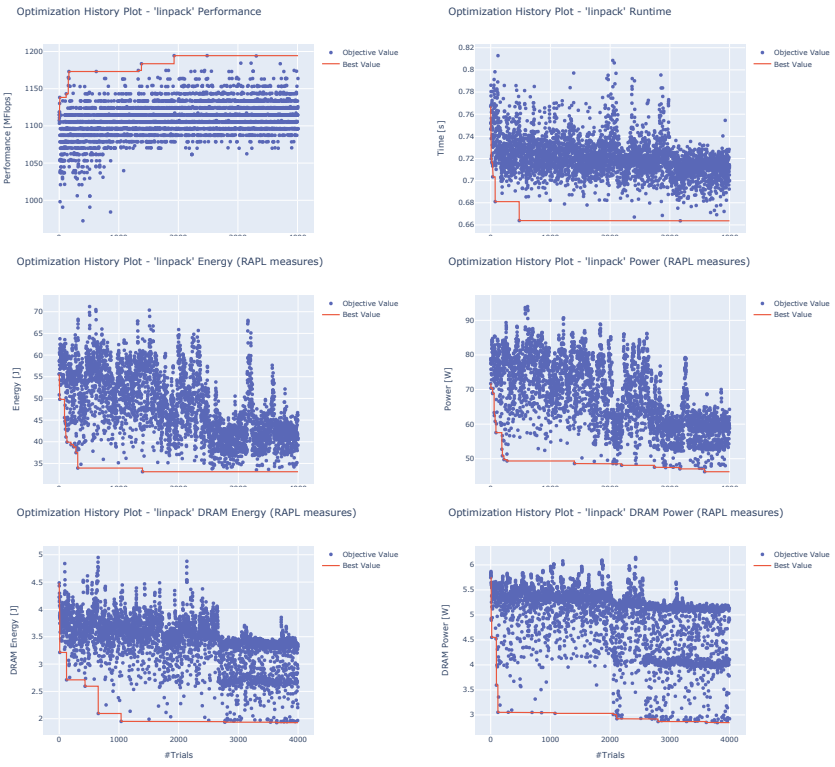

Fig. 2. Linpack benchmark Automatic tuning in EvoCode: local hyper-parameter optimization for performance, runtime, energy, power, DRAM energy & power metrics).

obtained against the program compiled with the highest level of optimization flags known by the compiler infrastructure (i.e., `-O3`).
The Fig. 3 reports the slice parameter relationships obtained for the sole energy optimization over the *reconstructed* program (thus considered in this case as a single region). Other similar figures were generated for the other fitness metrics, i.e., performance, runtime, power, DRAM energy and DRAM power, yet could not be presented for obvious space reasons. The objective of these analyses is to identify during the local optimization phase of EvoCode and for each optimized region the most sensitive code transformations to prune at an early stage of the global optimization unpromising configurations. Of course, it is crucial to correctly size the window for this local search strategy to avoid a premature convergence toward a local optima that may result in a non-diversity of the population. This type of evaluation is at the heart of the NSGA-III [3] heuristic currently under investigation within EvoCode.

Table 4. Best results obtained by EvoCode on the Linpack benchmark.

Metric		P_{ref} (`-O3` optimized)	Best EvoCode		
`perf`	(MFlops)	1109.39	**1194.43**	85,04	**+8%**
`runtime`	(s)	0.70	**0,65**	−0,05	**−8%**
`energy`	(J)	40.20	**33.08**	−7,11	**−18%**
`power`	(W)	57.27	**46.24**	−11,03	**−19%**
`dram_energy`	(J)	3.28	**1.93**	−1,35	**−41%**
`dram_power`	(W)	4.68	**2.85**	−1,83	**−39%**

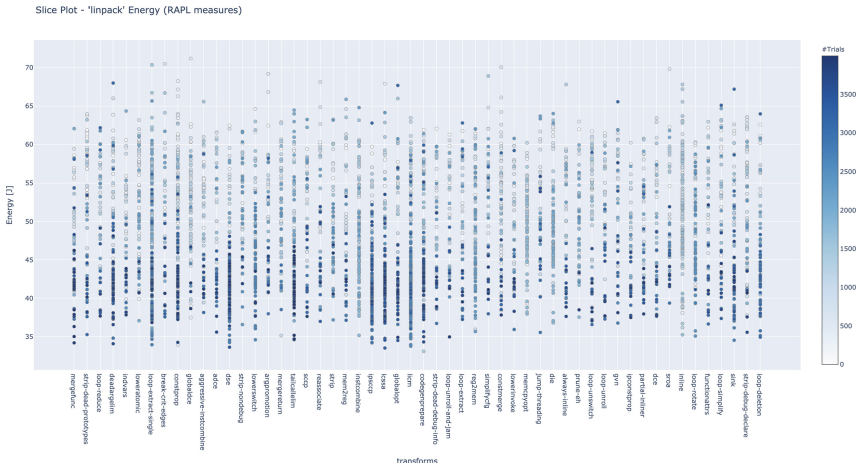

Fig. 3. Slice parameter relationships for the energy optimization tied to a single region (the full Linpack program).

5 Conclusion and Perspectives

This position paper presented the EvoCode framework aiming at the automatic software tuning of parallel programs for energy-aware executions. A reference source code is evolved using a two-stage MOEA heuristic (local and global) exploiting a compiler infrastructure (LLVM in this case) to apply **static** *tunable knobs* or *code transformations* to generate individuals[1]. The objective remains to address simultaneously multiple KPIs optimization i.e., performance, energy and power usage (for both for the CPU and DRAM) and the runtime, bringing a set of optimized binaries derived (and a priori semantically equivalent) from the reference program used as input of the EvoCode framework together with the target computing system. Our framework will also integrate a decision making process through post-Pareto-front analysis to suggest the best trade-off between the obtained solutions. EvoCode has been implemented and validated over a set of reference benchmarking applications being auto-tuned. The preliminary experimental results presented in this article (restricted to the most well-known benchmark i.e., Linpack) are quite promising. While illustrating and validating the local optimization strategy performed within EvoCode, they already demonstrate improvement for all considered metrics (ranging from 8% to 41%) when compared to the most optimized configuration set by the compiler on the reference program. The validation of the global MOEA phase within EvoCode through NSGA-III is under investigation and is bringing further improvements which will be presented in an extension of this work.

Acknowledgments. The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [11] – see hpc.uni.lu.

References

1. Agakov, F., et al.: Using machine learning to focus iterative optimization. In: International Symposium on Code Generation & Optimization (CGO 2006), pp. 295–305 (2006)
2. Carrillo, V.M., Taboada, H.: A post-pareto approach for multi-objective decision making using a non-uniform weight generator method, vol. 12, pp. 116–121 (2012)
3. Deb, K., Jain, H.: An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach. IEEE Trans. Evol. Comput. **18**(4), 577–601 (2014)
4. Desrochers, S., Paradis, C., Weaver, V.: A validation of DRAM RAPL power measurements. In: Proceedings of the 2nd International Symposium on Memory Systems (MEMSYS 2016), pp. 455–470 (2016)
5. Dongarra, J.J., Moler, C.B., Bunch, J.R., Stewart, G.W.: LINPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia (1979)

[1] Dynamic tunable knobs are also expected to be covered at a relative short term, but this will assume to rely on other compiling frameworks more suitable for such cases such as Insieme - see http://www.insieme-compiler.org.

6. Durillo, J.J., Fahringer, T.: From single- to multi-objective auto-tuning of programs: advantages and implications. Sci. Program. **22**, 285–297 (2014). https://doi.org/10.1155/2014/818579
7. Khan, K.N., Hirki, M., Niemi, T., Nurminen, J.K., Ou, Z.: RAPL in action: experiences in using RAPL for power measurements. TOMPECS **3**(2), 1–26 (2018)
8. Kieffer, E., Danoy, G., Bouvry, P., Nagih, A.: Bayesian optimization approach of general bi-level problems. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2017), pp. 1614–1621 (2017)
9. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California, March 2004
10. Naono, K., Teranishi, K., Cavazos, J., Suda, R.: Software Automatic Tuning (From Concepts to State-of-the-Art Results). Springer, New York (2010). https://doi.org/10.1007/978-1-4419-6935-4
11. Varrette, S., Bouvry, P., Cartiaux, H., Georgatos, F.: Management of an academic HPC cluster: the UL experience. In: International Conference on High Performance Computing & Simulation (HPCS 2014), Bologna, Italy, pp. 959–967. IEEE, July 2014