Mapping Medley: Adaptive Parallelism Mapping with Varying Optimization Goals

Murali Krishna Emani

Lawrence Livermore National Laboratory, USA
emani1@llnl.gov

Abstract. In modern day computing, the performance of parallel programs is bound by the dynamic execution context that includes inherent program behavior, resource requirements, co-scheduled programs sharing the system resources, hardware failures and input data. Besides this dynamic context, the optimization goals are increasingly becoming multi-objective and dynamic such as minimizing execution time while maximizing energy efficiency. Efficiently mapping the parallel threads on to the hardware cores is crucial to achieve these goals. This paper proposes a novel approach to judiciously map parallel programs to hardware in dynamic contexts and goals. It uses a simple, yet novel technique by collecting a set of mapping policies to determine best number of threads that are optimal for specific contexts. It then binds threads to cores for increased affinity. Besides, this approach also determines the optimal DVFS levels for these cores to achieve higher energy efficiency. On extensive evaluation with state-of-art techniques, this scheme outperforms them in the range 1.08x up to 1.21x and 1.39x over OpenMP default.

1 Introduction

Modern day parallel computing landscape is rapidly evolving in all aspects: right from applications composed of diverse workloads, middle-ware up to the hardware. The diversity and dynamic nature of all elements in this vertical stack is becoming more obvious than ever before. Given a parallel application is unlikely to run on the same platform and the same environment for its lifetime, we need a way to future-proof application development cost. The mainstream applications have no longer have the privilege of having exclusive access to hardware resources; but have to share dynamically with co-executing applications. Similarly, there is no longer a single optimization goal for the parallel applications. Earlier either it used to be either of latency or throughput or energy efficiency. However this is no longer the case. Multi-objective optimization is growing as the ultimate desired goal for parallel applications and systems, such as high throughput with minimum energy expenditure. Consider the case of software

[0] This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-696003

applications on a mobile device or embedded system. In a scenario when it is connected to external power source, the goal may be more on maximizing application performance. But when the power source is disconnected, it no longer has access to external power supply and has to rely only on its battery. In this scenario, the goal of maximizing energy efficiency may become as important as the application performance.

Thus the program performance is bound by the dynamic *execution context* which we define as composed of factors such as inherent program behavior, resource requirements, co-scheduled programs sharing the system resources, hardware failures, ever-changing software versions, and input data. Over-subscription with more software threads than the hardware threads may lead to program slowdown due to delays in threads gaining access to hardware. Under-subscription may result in poor resource utilization. Hence a judicious parallelism mapping is crucial to improve program performance. Thread to core affinity also impacts program performance. Frequent migration of threads and the data in respective caches drastically degrades the performance. It is also thus important to minimize thread placements across cores. Tuning CPU core frequencies is one approach to control the power consumption in the system. The frequencies can be lowered in many ways to improve power efficiency. Hence reducing power and the execution time may lead to high energy efficiency. Most of the existing approaches rely on a single mapping policy which remains the same irrespective of the current system characteristics. There is no ability to determine if this mapping is indeed optimal if the execution context changes. Any monitoring mechanisms if present, are reactive in that they observe the program execution with a configuration for few cycles. Based on the observed behaviour, the program mapping is varied. It is highly unlikely that the mapping determined by these approaches will be optimal for evolving workloads and hardware. Such policies cannot be easily advanced as they need radical changes in the policy, which are expensive to be performed at runtime.

Our idea: In this paper we focus on determining the best thread numbers for every parallel section of a parallel program and binding them to hardware cores. This is a key decision on maximizing parallelism with available resources. To optimize for the energy efficiency, we also determine the optimal frequency level for each core utilizing Dynamic Voltage Frequency Scaling (DVFS) mechanism. In the program execution context, we primarily focus on contention due to co-executing workloads, hardware failures and changes in the external power supply. We take inspiration from early work [8] which shows that a mixture of specialized models often outperforms a single policy. It maintains a collection of mixture of models which can be added to and updated as time goes on, selecting the model that is best suited to the current context. It avoids over-complex heuristics and over-fitting training data by allowing different models to be selected based on their worth. The work closest to our approach is the Ensemble mapping [5]. It uses predictive modeling that considers different mapping policies called *experts* at runtime and selects the one that is determined to be the optimal one at every parallel loop. As the program execution context changes, different mapping

policies will be dynamically selected at runtime. We extend and improve over the ensemble technique to optimize for both execution time and power and also consider the case of varying external power supply. We also optimize all executing programs in the system unlike just the target program in the ensemble method.

Our technique ***Mapping Medley*** uses a collection of exclusive mapping policies where each policy takes as input the execution context i.e. current co-executing workload, hardware and power supply and then determines the best threads numbers and optimum frequency levels for all cores. At runtime the question of which mapping policy to select is crucial for achieving the optimization goals. The standard method would be to run each policy for few runs or cycles and observe the program behaviour, identify and select the best mapping policy. Policy evaluation in such manner would be prohibitively expensive in terms of the overhead incurred at runtime. We avoid this overhead by instantly selecting the best expert based on the context. Our idea is to optimize program performance and energy by determining best thread numbers, pinning them to cores and determine optimal frequencies for the cores. Predictive modeling is the core strategy to our approach.

This paper makes the following contributions:

- First to optimize multi-objective goals in varying execution contexts.
- Propose techniques to optimize execution time and energy efficiency simultaneously.
- Outperforms existing state-of-art approaches on extensive evaluation.

2 Related work and Motivation

Related work: The works closest to ours are Ensemble mapping [5] and Feedback-driven technique [6]. The ensemble mapping approach employs 'Mixture of Experts' concept [8] from machine learning domain. Here multiple specialized mapping policies called *experts* are employed which are offline trained machine learning models. These individual experts determine thread numbers and future system state. An online expert selector chooses the best expert based on what expert predicted the most accurate system state. This technique aims only at thread number prediction but does not mention about their placement and run all cores at maximum frequencies. The feedback driven policy [6] uses control theory-based techniques to tune different knobs based on feedback from the system. It first changes the power control knob to get the power consumption below a capped value and then tunes a performance knob to extract maximum performance possible. It relies on an incremental approach; tune for one goal first and later tune for another goal. Though this approach may eventually find the optimal configurations, it may take a while to reach which is not desirable at runtime. Our work directly tackles both execution time and energy efficiency at once, thus ensuring quick arrival to an optimal configuration. The approach presented in [14] uses analytic model to determine best number of threads at runtime. It includes an observe-and-change policy where every parallel loop is run with random thread numbers for few cycles. Then based on the observed performance,

it builds an online regression model to determine the optimal thread number. DVFS techniques are employed in solutions proposed in [11] which change the processor frequencies, according to the code characteristics and runtime information. A machine learning mapping policy is proposed in [15]. The policy employs no way to adjust the policy based on the execution context changes and no method to measure its efficiency online. Another ensemble search approach proposed in [2] involves running multiple configurations at the same time on partitioned system space. Once a best configuration is found, it replaces the previous best configuration. Multiple policy evaluation at the same time limits the physical resource availability for the target program. This problem worsens when the hardware is dynamic with changing number of processors. Energy efficient parallelism is well studied in embedded systems community dealing with computing devices with limited power sources in [3]. Adagio [13] is a runtime system that makes DVFS practical for complex high performance computing applications. Implications of thread level parallelism on performance and power are discussed in [9].

Motivation example: In this section, we provide an example to motivate the goal. The experimental set up is a co-execution of parallel programs with varying thread numbers and a sudden change in the power supply at runtime. On a two 4-core Intel Xeon machine laptop with 16GB RAM running Ubuntu 3.7 kernel, we ran a target program *pagerank* from Green-Marl benchmark [7]. There is a co-executing workload program *cg* with 4 threads till time t=25 sec and later another workload *is* with 2 threads both from NAS benchmark suite [1]. The number of processors remains constant throughout. We then simulated a change in power supply to the system. Till 30sec, external power supply was connected to this system after which it was disconnected leaving the system to run on its battery power till the end of program execution. In this set up we evaluated the target program performance and plotted the number of threads determined by the OpenMP default scheme, analytic, feedback and ensemble techniques as described in Section 4. The obtained speedups over default are 1x, 1.21x, 1.32x and 1.34x. We also then tried running the target exhaustively with all thread numbers to identify the maximum possible performance (1.6x) and plotted the optimal threads numbers. The thread numbers are plotted in the second graph in Fig. 1. We also measured the energy consumption and plotted the energy efficiency normalized over the default scheme. We observe that the evaluated techniques fall short of the optimal thread numbers needed for best performance. They become more unstable after time t=30sec and for poor performance and energy efficiency.

The figure demonstrates that there still is a large room for improvement in execution time in terms of better thread numbers and energy efficiency. We try to tackle this issue of how to quickly and efficiently obtain the best thread numbers for maximizing speedup and optimal core frequencies for higher energy efficiency. We discuss our idea in the next section.

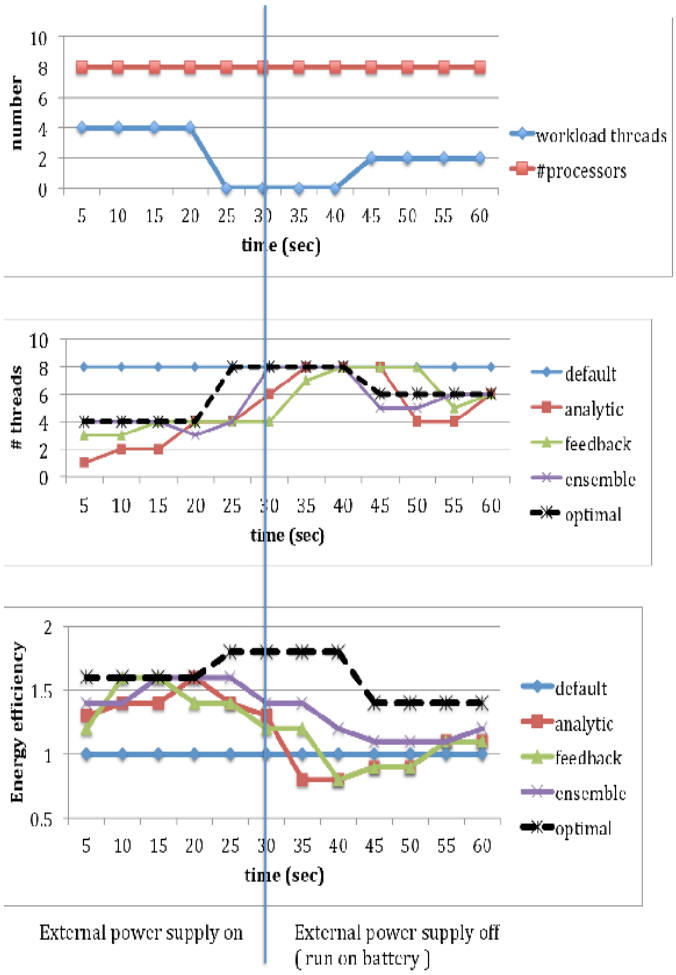


Fig. 1: Graph showing how #threads and energy efficiencies of different approaches vary with a change in the power supply. The top graph shows the number of processors and co-executing workload threads. The second graph shows the #threads determined by default, analytic, feedback, ensemble and optimal values. The bottom graph shows energy efficiency values normalized to the default policy. At time t=30 sec, external power supply is removed to run rest of the program execution on battery source. It can be observed that all policies become unstable and move away from the optimal, with a change in power supply and remain far from the optimal value.

3 Mapping Medley

3.1 Optimal thread number

The primary goal of this work is to achieve maximum speedup with minimum energy expenditure. This goal can be achieved by tuning multiple configurable parameters or *knobs*. In this work we limit the tunable parameters to (i) *thread number*, (ii) *thread placement* and (iii) *DVFS level of cores*. on which the threads are placed. We try to optimize (a) *Execution time:* We try to achieve best execution time by (a) determining the best number of threads for the target program that minimizes the execution time and (b) setting up threads-to-core affinity that minimizes data movement across cores. (b) *Energy efficiency:* Once the optimal number of threads are determined and pinned to respective cores, we then maximize energy efficiency. This can be achieved in multiple ways; here we utilize the most widely used technique: changing the frequency levels of the cores where the threads are mapped.

We built our approach over the mapping technique in [5].This ensemble technique is composed of multiple specialized mapping policies called experts. Each expert is an offline trained linear regression model trained in a specific setting. It has two predictors that predict (i) best thread number and (ii) expected system state. The online expert selector evaluates the most appropriate thread-predictor based on the current system state and determines thread numbers of that predictor to be ideal at that point of time. Each expert is tuned on program scalability and different hardware. The inputs to the thread predictor are a set of features that capture both code and system characteristics obtained from the compiler and the kernel respectively. The set of features are listed in Table 1.

We differ from [5] in the expert selection mechanism. They use a second machine learning model *environment predictor* that predicts what the system should be if the thread number was indeed optimal. This may cause additional overhead and may not accurately capture the system state specially when the power supply source varies. We use the four thread predictors or experts as in [5]. Here each model determines a thread number based on the current parallel section characteristics and execution context that include any co-executing workloads and hardware changes. Our approach now deploys a simple yet smart technique where it switches between the largest thread number with external power supply on and the least thread number when the external power supply is off. The reason is that large number of threads increase the power consumption though reduce the execution time. Note that if the power sources from a battery, the optimization goal now prioritizes energy efficiency to execution time.

3.2 Thread placement

Once the best number of threads are determined, we pin them to the cores to minimize frequent thread migration. It is widely acknowledged that the placement of parallel threads across cores can greatly affect the program performance. Migrating a thread from one core to another also involves either moving the data

it requires from caches of current core to the caches of the core to which it is migrated to. Else this thread has to remotely access its data from the caches of core it was previously running on. Both mechanisms are highly expensive in terms of the overhead and drastically degrade performance. Ideally threads finish their computation faster when the data they require is within caches of local cores. In this approach, once the thread number is determined, these threads are pinned to the hardware cores to enhance affinity. This reduces the chances of potential problems with thread migration as discussed above. If the thread number of a current parallel section is lesser or equal to thread number of previous parallel section, we do not change the affined cores. Only when the number of threads are larger, we include more affined cores. The threads are affined using *sched_setaffinity* system call.

3.3 DVFS level

Dynamic Voltage and Frequency Scaling (DVFS) is one of the methods to alter the processor frequencies to reduce power consumption of the cores. Each processor can be assigned with a set of frequencies that varies with the processor family and type. In our experiments the list of available frequencies is: (2.3, 1.8, 1.6, 1.4, 1.2, 1.0, 0.8) GHz. The frequency levels of each core can be tuned on-the-fly, however, in this work we change the frequency levels of only the cores to which threads are pinned to a single value. It may be noted that an optimal DVFS level for cores is determined at every parallel section.

3.4 Components

The two components core to our approach are (i) thread predictor and (2) frequency predictor. The thread predictor chooses best expert with its thread number and pins them to equal number of cores. The frequency predictor determines the best DVFS level and sets corresponding frequencies to these cores. The idea of our approach is shown in Fig. 2. We use *likwid* [10] to set CPU core frequencies and measure the power consumption obtained from from the MSR registers using likwid-powermeter.

Thread-predictor: Each expert policy takes code and system features as inputs to determine a thread number each. Our thread predictor chooses the best expert and its thread number based on the power supply status. It switches between the largest thread number with external power supply on and the least thread number when the external power supply is off. Let E^1, E^2, E^3, E^4 be the four thread mapping policies or experts. Let c be the set of code features, s the set of system features and p indicates the status of power supply, 0 for *full* (has external power supply) or 1 for *discharging* (runs on battery power). Let $t(E^i)$ be the thread numbers of i^{th} expert. Then the policy of this thread predictor 'g' is to determine the optimal thread number t (1-8) in this work as shown in Equation 1.

Feature	E^1	E^2	E^3	E^4
memory-accesses	1.05	-0.84	0.14	0.05
instructions	-1.52	1.12	0.95	0.03
branches	0.87	0.84	-0.87	-0.57
software-threads	-0.62	0.05	-0.48	0.004
processors	0.98	0.98	0.99	0.92
task queue size	0.003	0.02	-0.15	0.22
cpu load-1	0.002	0.03	0.473	0.01
cpu load-2	-0.013	0.227	-1.07	-0.62
cached memory	-0.07	0.002	0.007	0.03
pages free list rate	0.004	-0.08	0.01	-0.14
error	-1.21	-6.8	-3.03	-2.5

Table 1: List of features and weights used in thread predictor obtained from [5]

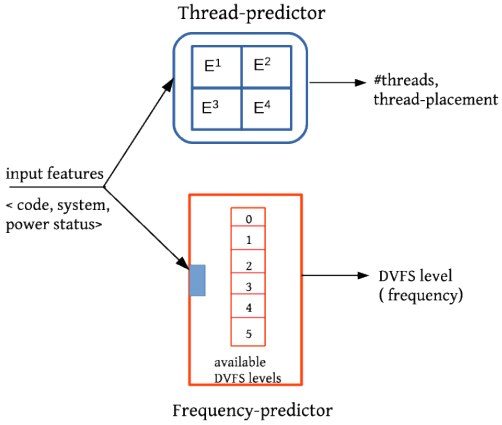


Fig. 2: Mapping Medley approach. The input feature vector is passed to two components (i) thread-predictor that determines the best number of threads and pins them to equal number of cores (ii) frequency-predictor that determines the optimal DVFS level and sets its corresponding frequency level to cores.

$$g(c,s,p) = \begin{cases} t|max(t(E^1),..,t(E^4)), & \text{if } p=0 \\ t|min(t(E^1),..,t(E^4)), & \text{if } p=1 \end{cases} \quad (1)$$

Thread placement: The number of threads are then pinned to cores using 1:1 thread-to-core mapping. For example, if this model outputs 5 threads, then these threads are affined to 5 cores. The affinity holds till the parallel section execution is finished. Let P_{max} be the maximum number of available processors and c_j be the j^{th} core. The set of cores to pin these threads is $(c_i, ...c_t)$ for any $i, t < P_{max}$ and $i < t$ chosen from the set of free cores.

Frequency-predictor: Let l denote a DVFS level. The frequency predictor model uses the policy 'f' that takes the combined code, system and power supply status as input feature vector and outputs the optimal DVFS level (0-6 in this work) as shown in Equation 2. The corresponding frequency is then set to the cores using: *likwid-setFrequencies -c set-of-cores-to-pin -f predicted-frequency.*

$$f(c, s, p) \rightarrow l \tag{2}$$

Machine learning: Both thread predictor and frequency predictor are offline trained machine learning models. We use linear regression model for the thread-predictor and a support vector machine (SVM) for the frequency predictor. The weights for four experts used by our thread predictor are listed in Table 1. These weights when multiplied with values of the extracted features, yield a thread number. Our thread predictor determines thread numbers that differ from [5] as they take the power supply information as one of the input features. SVM is a supervised classifier that assigns a class (DVFS level from 0-6) for every given input. We evaluated the frequency predictor using a regression model, but surprisingly it yielded poorer prediction accuracy of 78% compared to 89% of SVM. Hence we chose SVM to build the frequency predictor model.

Training data and Features: The training data is generated using the replicated set up as in [5]. On two different hardware platforms, training programs are run with co-executing workloads, while collecting all possible features. Two classes of program scalability on two platforms provides the training data for the four experts. Thread numbers are varied for each training run to determine the configuration that yields least execution time. In another set of training runs, the frequency levels are varied to determine the best DVFS level that has the least energy consumption. These supervised models are cross-validated to avoid overfitting the data. They are trained on a set of training programs and evaluated on new unseen test programs. The set of features are obtained after collecting all possible features and then eliminating those which do not provide any meaningful hints using entropy estimation.

Portability: It is always ideal to enable portability of the generated models to avoid extensive retraining on every platform of interest. The thread predictor captures basic system information as processors and memory in its feature set. The frequency predictor gets the number of processors from thread predictor and available set of frequencies from the kernel. It then determines the best frequency level. Moreover, on a new platform with different hardware, the corresponding set of available frequencies are known to choose the best one.

4 Experimental Setup

4.1 Platform

The hardware and software platform setup used in the evaluation experiments is listed in Table 2. Note, we have not evaluated this work on hardware that

supports the Intel Turbo Boost Technology in this work and would consider that as a planned future work. All the programs start execution at the same time and continue till the other finishes. Each experiment was repeated 10 times and the geometric mean value of execution time is reported. The energy efficiency is computed by the product of measured power and the execution time. The power consumed is obtained from likwid-powermeter [10] that reads power consumption values from the model specific registers (MSR). Note that the measured speedups and energy efficiencies are averaged for all co-executing programs.

4.2 Applications

We use a variety of parallel programs from benchmark suites of different computational behaviours each with largest input data set. These include all OpenMP-based C programs from NAS [1], Parsec [12] and *pagerank* from Green-Marl project [7] benchmark suites. To ensure fair comparison, we replicate a similar experimental set up and workload applications from [5].

Hardware	Laptop with two 4-core Intel Core i3-2350M, @ 2.30GHz 16GB RAM, 3MB shared LLC
OS	64-bit Ubuntu, 3.7.10 kernel
Compiler	*gcc* 4.6 -O3 optimization

Table 2: Experimental setup

Workload type	**Programs**
light	(i) `ep, bodytrack` (ii) `is, ft`
heavy	(i) `pagerank, blackscholes, bt, sp` (ii) `lu, freqmine, bt, freqmine`

Table 3: Programs that constitute two types of workloads.

4.3 Competitive Policies

We evaluated our approach against the following state-of-art mapping policies. The experimental setup along with same set of workload programs are replicated to ensure fair comparison with the evaluated policies.

Default: OpenMP default policy [4] assigns a thread number equal to the current number of available processors.

Analytic: In [14] an analytical model determines the degree of parallelism at runtime based on observed speedups at fixed time-intervals and estimated using regression techniques.

Feedback: The feedback driven policy [6] uses techniques to adjust programs performance and power by tuning different knobs based on feedback based on

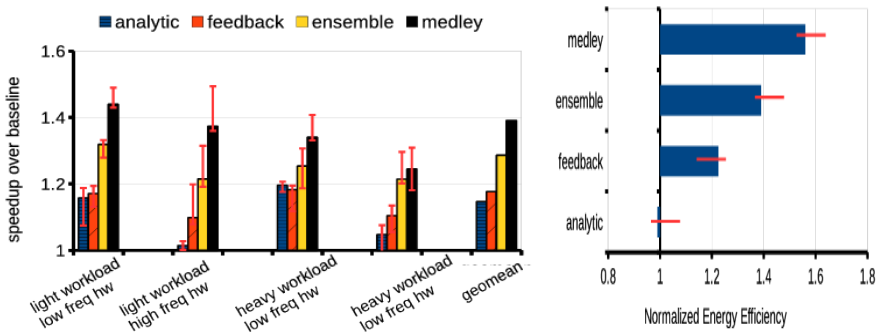
control theory principles. It first changes the power control knob to get the power consumption below a capped values and then changes to performance knob to extract maximum performance possible.
Ensemble: The Ensemble technique [5] uses a mixture of offline trained machine learning models that predict the best number of threads in dynamic program environments. The experts are highly specialized based on the executing environment and the online expert selector switches between experts choosing the optimal one as and when required.

4.4 Experimental Scenarios

The dynamic execution context is composed of co-executing workloads and hardware in terms of number of processors and power supply.
(i) Workloads: The external workload consists of multiple parallel programs selected from the above benchmarks. We vary the number of workload programs chosen from above programs classified as '*light*' and '*heavy*'. For each workload type, we consider different sets of programs as shown in Table 3. All results are averaged over these different benchmark sets. The same external workload is reproduced for all evaluated policies in all cases. This ensures a fair comparison across different mapping policies.
(ii) Hardware: To reflect any change in hardware, we vary the number of available processors during program execution. Changes in the number of processors can be due to several factors including hardware failures, assigning more/less cores for other high/low priority jobs, turning them off for saving power. The number of available processors is varied in two different frequencies: *low* and *high* where it is changed at every 40 seconds and 10 seconds in low frequency and high frequency settings respectively. We modify the processor count by switching '*online*' values (1 = enable, 0 = disable) for each CPU in */proc* file-system. Disabling a CPU is logically shutting it down on-the-fly. Hence no threads are scheduled on the disabled cores.
(iii) Power supply: In all experiments, we reflect a change in the power by starting each run with power supply on and later disconnecting it during program execution. The status of power source can be observed from observing the battery status value obtained from the kernel */sys/class/power_supply/BAT0/status*. If the status values reads *full*, it implies that the system has external power supply, else if it reads *discharging*, it means that power supply is no longer available and the system has to utilize its battery power.

5 Evaluation

Here we show the summary of performance results of speedup and energy efficiency of all policies, averaged across all the experimental settings on the evaluated scenarios. In all cases, the baseline is OpenMP 3.0 default policy and the average values (geomean) are geometric means to avoid outliers. The measured speedups are for both target program and any co-executing workloads and the energy efficiency is across all cores.



(a) Speedup (b) Energy efficiency

Fig. 3: (a) Performance of program speedup all evaluated approaches averaged across all programs and experimental settings. Our approach outperforms all by greatly improving speedup by 1.39x over baseline. (b) Performance of energy efficiency of all evaluated approaches. The medley approach outperforms all by significantly by achieving 1.44x efficiency.

Speedup: Fig. 3(a) shows the summary of evaluation results for speedups across all benchmark programs and all experimental scenarios. The x-axis shows each scenario for workload type and frequency of hardware changes and the geomean. The analytic, feedback, ensemble improve program speedup by 1.15x, 1.18x, 1.28x over the baseline. Our medley mapping outperforms all these competitive techniques by recording 1.39x improvement over baseline. It has better speedup of 21.28% over analytic, 18.19% over feedback and 8.14% over ensemble techniques. The primary reason for these speedups is the optimal determination of thread numbers along with minimizing frequent thread placements across cores reducing latencies in data accesses.

OpenMP default policy assigns same number of threads as the number of available processors. Due to increased resource contention caused by the dynamic execution context and reduction in power source, it is unable to modify thread numbers accordingly. The analytic technique sustains workload changes but does not adjust to varying number of processors. It also suffers from frequent thread movements across cores. The feedback policy relies on the signal it receives from the system and changes configurations based on the observed system changes. It is a reactive policy and tunes the available knobs for execution time and power in exclusion. The ensemble approach is quick to react to execution context changes and selects thread numbers owing to the presence of expert mapping policies. It however does not pin down threads to cores which may lead to frequent changes in threads to cores placements. Our medley approach utilizes the same number of threads as the ensemble, but also pins threads to hardware cores to reduce any chance of thread migration to minimal. Therefore it further improves the program execution time.

Energy efficiency: Overall energy efficiency results. The values reported are normalized to the OpenMP default baseline. The value more than 1 implies that the policy achieved better efficient mapping over the baseline, else a poor mapping in terms of energy consumption.

Fig. 3(b) shows the energy efficiency values averaged across all evaluated scenarios. It can be observed that our medley approach always achieves better efficiency by lowering the power consumption and the execution time. It improves over 1.44x over the baseline outperforming the compared approaches. The ensemble technique also improves energy efficiency in a range of 1.12x to 1.55x. The feedback mechanism performs poorer due to the frequent change in the number of processors where it rapidly changes its configuration leading to fluctuations in thread numbers and DVFS levels. The analytic approach improves energy in only two scenarios with light workloads, however, with heavy workloads it significantly drops down below the default baseline. This is due to the increased execution time due to the enormous time taken to reach the optimal thread number.

6 Analysis

6.1 Thread number variation with change in power

In this section we analyze the thread number counts averaged across all parallel sections for all evaluated benchmarks, determined by all evaluated approaches in two phases: before and after the change in the power supply. This is to understand how the thread numbers are affected by the changes in optimization goals and external parameters. It can be observed from Fig. 4 that with external power supply on, all policies determine larger thread numbers most of the time. But when the power supply is removed, the system relies on battery power. Now all policies try to determine smaller thread numbers to minimize the energy expenditure.

6.2 DVFS level variation with change in power

Fig. 5 shows how frequently a DVFS level is determined by our approach before (top) and after (below) the power supply change. The values are normalized to 100%. It can be observed that with external power supply on, our policy determines larger frequency levels for all cores to improve the execution time of all running programs. After the power supply is removed, the goal to minimize energy expenditure is prioritized and lower frequency levels are determined.

7 Conclusion and Future work

We presented a novel parallelism mapping technique that optimizes execution time and energy efficiency in dynamic execution contexts. The work is more relevant in modern day hardware devices where multiple workloads co-execute

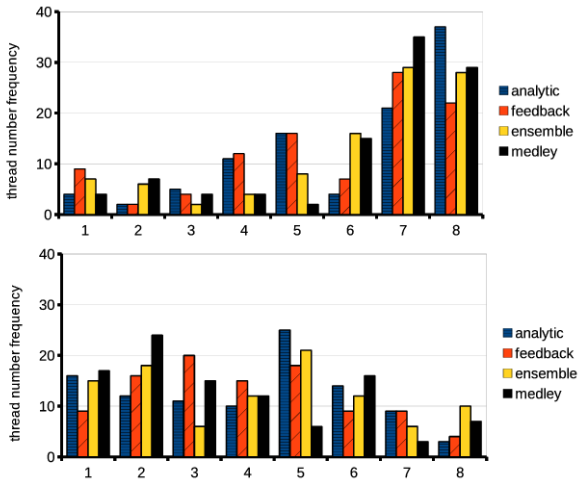


Fig. 4: Distribution of thread numbers by all evaluated policies before (top) and after (below) the power supply change. With external power supply on, all policies determine large thread numbers most of the time and vice-versa.

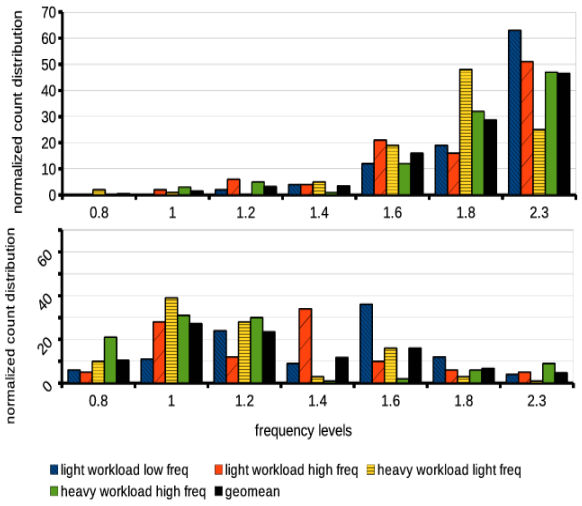


Fig. 5: Distribution of DVFS levels by our approach before (top) and after (below) the power supply change. With external power supply on, our policy determines larger frequency levels. After the power supply is removed, lower frequency levels are determined.

with changes in hardware and the source of power supply. Our technique determines best number of threads at runtime and pins them to underlying hardware cores and sets optimal core frequencies. As part of future work, we would like to evaluate on embedded platforms and mobile devices running parallel workloads. We also would explore changing DVFS levels per-core instead of all cores.

References

1. "NAS 2.3," http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html.
2. J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe, "Siblingrivalry: Online autotuning through local competitions," in *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '12, 2012.
3. M. F. Cloutier, C. Paradis, and V. M. Weaver, "Design and analysis of a 32-bit embedded high-performance cluster optimized for energy and performance," in *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, ser. Co-HPC '14, 2014.
4. L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.
5. M. K. Emani and M. F. P. O'Boyle, "Celebrating diversity: a mixture of experts approach for runtime mapping in dynamic environments," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.*
6. A. Filieri, H. Hoffmann, and M. Maggio, "Automated multi-objective control for self-adaptive software design," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015.
7. "Green-Marl," https://github.com/stanford-ppl/Green-Marl.
8. R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural Comput.*, vol. 3, no. 1, pp. 79–87, Mar. 1991.
9. J. Li and J. F. Martinez, "Power-performance implications of thread-level parallelism on chip multiprocessors," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on.* IEEE, 2005, pp. 124–134.
10. "likwid," https://github.com/RRZE-HPC/likwid.
11. A. Merkel and F. Bellosa, "Memory-aware Scheduling for Energy Efficiency on Multicore Processors," in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, ser. HotPower'08.
12. "Parsec 2.1," http://parsec.cs.princeton.edu/.
13. B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: Making dvs practical for complex hpc applications," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 460–469. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542340
14. S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014.
15. Z. Wang, M. F. P. O'Boyle, and M. K. Emani, "Smart, adaptive mapping of parallelism in the presence of external workload," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13, 2013.