Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Implementation of an Energy-Aware OmpSs Task Scheduling Policy

Jan Christian Meyer[a*], Thomas B. Martinsen[b] and Lasse Natvig[b]

[a]*High Performance Computing Section, IT Dept., NTNU, Trondheim, NO-7491, Norway*
[b]*Dept. of Computer and Information Science (IDI), NTNU, Trondheim, NO-7491, Norway*

Abstract

The OmpSs programming model supports task-based parallelism in a similar manner to OpenMP. This whitepaper explores the possibility of implementing an energy-aware scheduling policy in run-time component of the OmpSs programming model, to adapt task execution schedules for balancing energy efficiency with parallel performance. A high-level design description of a run-time scheduling plugin to achieve this is presented, as well as key results from studying its effectiveness with 4 performance metrics, using 17 application benchmarks. The results show that the approach can be leveraged to improve energy efficiency in scenarios where dynamic power accounts for a large component of total power consumption, to benefits that can be programmatically balanced with predicted performance loss.

1. Introduction

The goal of this whitepaper is to give an overview of the design of an energy-aware task scheduling plugin for the runtime component of the OmpSs programming model, and survey some key results obtained from testing it with a range of benchmarks for implementations of the OpenMP task construct. The paper comprises an overview of key results from the report “Energy Efficient Task Pool Scheduler in OmpSs”[1].

2. Background

The OmpSs programming model [2] provides an implementation of the *task* construct similar to that introduced in OpenMP 3 [3], enabling applications to expose parallelism through annotating code sections with their data dependencies, and relying on a run-time system to schedule the execution of the resulting task dependency graph at run time. It implements this division of labor as a compiler which generates code for its Nanos++ runtime system, which admits configuring different task scheduling policies by providing compliant plugins, *i.e.* user-provided shared library code that manipulates the mapping of tasks to threads in response to events such as the appearance of a new task, or the completion of the last task scheduled for a thread. Plugins may associate custom data structures with threads and teams of threads, which provides a means of storing and analyzing continuously sampled performance and energy consumption data at run time, admitting that scheduling decisions can be based on such parameters.

Weissel and Bellosa [4] propose an energy-aware scheduling policy for non-realtime operating systems, which utilizes performance counters to determine the appropriate clock frequency for a running process. Their approach models the most effective clock frequency as a function of instructions per cycle (IPC) and memory requests per cycle (MRPC), and approximates it using a pre-computed look-up table constructed from testing six synthetic benchmarks. We take a similar approach in this whitepaper, with the key differences that the lookup table covers a different, greater parameter space, and that it addresses thread scheduling within an application program, as opposed to a granularity of processes at the operating system level.

Spiliopoulos, Kaxiras and Keramidas [5] develop governor modules for the Linux OS kernel that regulate frequency based on performance counter values. The governors make predictions at 50ms intervals, aiming to minimize the Energy Delay Product for memory intensive applications, where reducing the processor frequency lowers the latency of memory access relative to computation speed.

[*] Corresponding author. *E-mail address*: Jan.Christian.Meyer@ntnu.no.

3. Method

This section describes the methods employed to design an intelligent agent that dynamically adjusts processor frequencies in response to a task workload generated at run-time. Section 3.1 briefly details the design of the agent itself as a scheduling plug-in for OmpSs, Section 3.2 describes how its precomputed lookup tables are constructed, and Section 3.3 describes experiments to validate its accuracy.

3.1 Design of the scheduling plug-in

Preliminary studies using the Barcelona OpenMP Tasks Suite (BOTS) [6] and custom synthetic benchmarks revealed that energy efficiency may vary not only with selected task scheduling policies, but also with thread configuration. In some cases, activating all available cores was beneficial, while in others, performance can stagnate or decrease with a growing number of cores, due to resource contention. The implementation of the Nanos++ run time system relies on POSIX threads, and the Linux POSIX thread library does not support thread suspend/resume operations, due to the possibility of suspending a thread while it holds a lock. This precludes us from taking an approach of dynamically reconfiguring threads to expose load variations to the operating system. Instead, we take a more explicitly programmed approach, through applying the *userspace* governor that allows user programs to explicitly set core frequencies, and extend the Nanos++ Distributed Breadth First scheduling module with a separate *intelligent agent* thread, to regulate the task distribution and adjust frequencies in response to performance counter measurements.

As illustrated in Figure 1, the agent looks up frequencies in a pre-computed table, indexed by values of Instructions Per Cycle (IPC), Last Level Cache Misses Per Cycle (LLCMPC) as supplied by the PAPI interface [7], and the number of active cores. In a similar way to the approach of Weissel and Bellosa [4], this requires the table to be constructed from a representative selection of training benchmarks.

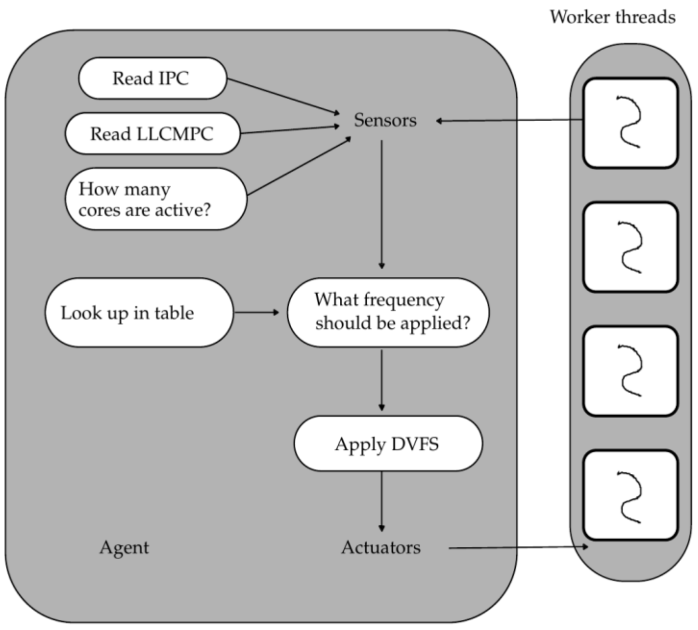

Fig 1. Overview of the Intelligent Agent

3.2 Generation of the look-up table

Out of the three dimensions of the look-up table, only the number of cores is inherently discrete, so the IPC and LLCMPC dimensions require partitioning of their domains into *bins*. The granularity of this partitioning affects the potential accuracy with which an optimal frequency can be chosen, but this creates a trade-off with the cost of the training procedure, as the number of required samplings grows as the product ($\mathrm{IPC}_{\mathrm{bins}}$ · $\mathrm{LLCMPC}_{\mathrm{bins}}$ · Cores · Frequencies), and purposefully producing every combination of IPC and LLCMPC would require a customized synthetic benchmark to control both independently.

The training procedure initializes every combination of IPC, LLMPC and Cores with the highest possible frequency. A set of computational kernels are then executed for all available frequency and thread configurations, recording energy consumption, IPC and LLCMPC. The most energy effective frequency is chosen according to an operations per Joule metric, and tabulated as optimal for the parameter configuration. In order to adjust table entries where the relevant parameter combination has not been measured, the table is updated with the constraint that no bin may list a frequency higher than any bin which represents a combination of higher LLCMPC, higher core count, and lower IPC value. This constraint reflects an assumption that increasing clock frequency introduces an energy cost with no benefit when increased parallelism adds memory traffic without improving performance. It serves to eliminate the artificially high frequencies that would otherwise appear in the look-up table, at parameter combinations not produced by the training set. Training was performed using a selection of 8 benchmarks listed in Table 1, which were chosen to reflect a variety of common application kernel behaviours (*dwarfs*), as identified by Asanovic *et al.* [8].

Table 1. Computational kernels used for training and verification

Benchmark	Dwarf
Dense matrix multiplication	Dense linear algebra
Sparse matrix-vector multiplication	Sparse linear algebra
3D Stencil	Structured grids
N-body	N-body methods
FFT	Spectral methods
NQueens	Backtrack and branch-and-bound
Histogram	Map reduce / unstructured grids
Merge sort	Graph traversal

3.3 Experimental procedure

The agent was configured to sleep for a 250ms interval, before using recorded values to predict optimal frequency for the next interval. In order to measure the accuracy of the approach, the average of the predicted frequencies over the course of an experiment was compared to the optimal frequency found for each of a set of kernels, as determined from runs at all available frequencies. Experiments were performed with four different criteria for optimal frequency choice: maximal operations per Joule, Energy Delay Product, and both metrics constrained by an additional requirement that performance cannot drop by more than 10% relative to maximum frequency. In addition to the benchmarks used to train the agent, an extra set was added to validate its accuracy independent of the bias inherent to testing with the training set. These additional benchmarks are categorized in Table 2.

Table 2. Computational kernels used for verification only

Benchmark	Dwarf
Quick Sort	Graph traversal
Reduction	Map reduce / Dense linear algebra
Black Scholes	Dense linear algebra
Vector operation	Dense linear algebra
Fibonacci	Graph traversal
Strassen	Dense linear algebra
SparseLU	Sparse linear algebra
2D Convolution	Structured grids
Unstructured 3D stencil	Unstructured grids

The experimental platform used is a dual-processor configuration with two 8-core Intel Xeon E5-2670 CPUs, each with 20MB shared Level3 caches as last-level cache. The PAPI events collected were the number of instructions completed (PAPI_TOT_INS), Level3 cache misses (PAPI_L3_TCM) and total cycles (PAPI_TOT_CYC), which admit the derivation of IPC and LLCMPC figures. Energy instrumentation was accomplished by utilizing a library which reads the Sandy Bridge RAPL Model Specific Registers, as described in "*Power instrumentation of task-based applications using model-specific registers on the Sandy Bridge architecture*" [9].

4. Results

This section presents the results obtained for the benchmark suite using 16 threads, corresponding to fully populating all processing cores on the test system. This selection is made because the agent's adjustments of clock frequencies impact dynamic power use, and full system utilization creates the conditions where this component accounts for its greatest possible part of total power consumption. The tendencies described here are present, but less visible for lower degrees of parallelism also; for a complete review of all tested configurations, the reader is referred to [1]. Subsections 4.1 through 4.4 summarize results for each of the four metrics used as optimization criteria for the intelligent agent, relating energy consumption and performance penalty to a baseline of running the benchmarks at maximum clock frequency with a distributed breadth-first task scheduling policy. Subsection 4.5 presents a measure of the accuracy of its predictions, comparing to empirically determined, optimal choices of constant frequency for each benchmark.

4.1 Operations per Joule

Achieving an optimal operation count per energy unit favors lowering frequencies regardless of the impact on performance, optimizing for an absolute energy gain.

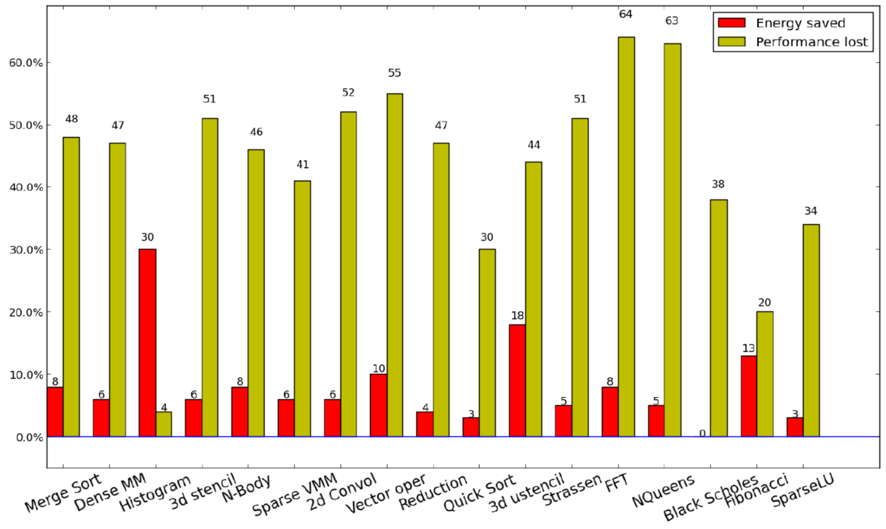

Fig 2. Benchmark performance and energy savings, with agent optimizing for Operations per Joule

4.2 Energy Delay Product

As the Energy Delay Product weighs energy consumption and performance equally, frequencies are not lowered unless the resulting energy savings are predicted to be greater than the performance degradation. Note that the sign of these magnitudes is chosen to visualize the trade-off between saved energy and *lost* performance, making performance improvements appear as negative values.

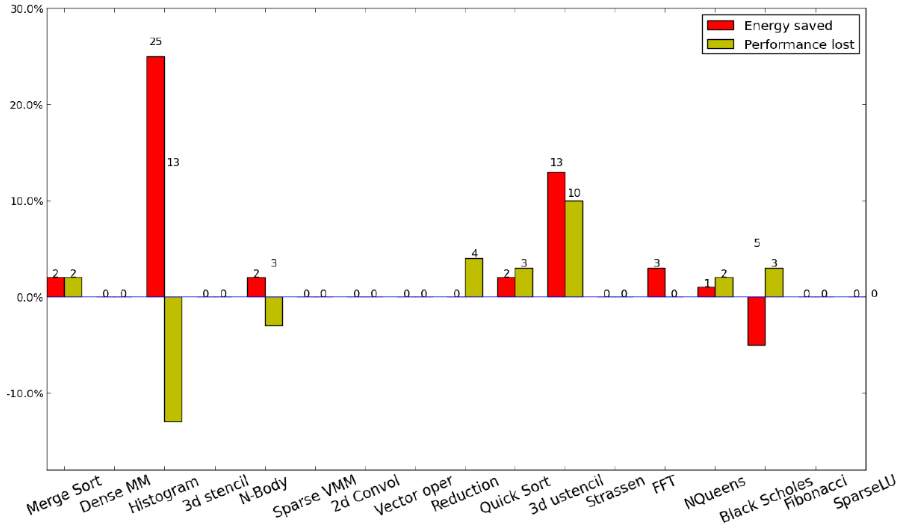

Fig 3. Benchmark performance and energy savings, with agent optimizing for Energy Delay Product

4.3 Operations per Joule with performance degradation bounded to 10%

Restricting the agent from lowering frequencies by applying a threshold value for allowed performance degradation limits the degree to which the agent lowers frequency, while still allowing it to respond in a great number of cases. This has the advantage providing a means to limit degradation with a more relaxed requirement than that of the Energy Delay Product, but comes with the disadvantage that inaccurate performance estimates can produce cases where the limit can be exceeded in practice.

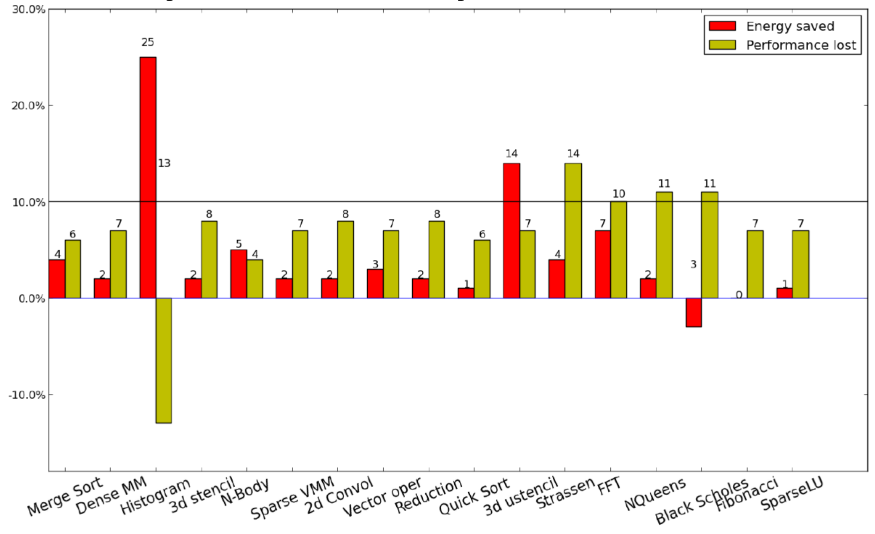

Fig 4. Benchmark performance and energy savings, with agent optimizing for Operations per Joule, 10% threshold

4.4 Energy Delay Product with performance degradation bounded to 10%

Restricting the admissible performance degradation for the Energy Delay Product restricts the number of admissible cases in a similar way to the description in Subsection 4.3. As observed in the results of Subsection 4.2, use of the Energy Delay Product already serves as a tight restriction of the frequency range employed by the agent. Accordingly, results are expected to resemble those shown in Subsection 4.2, and Fig. 5 validates this assumption.

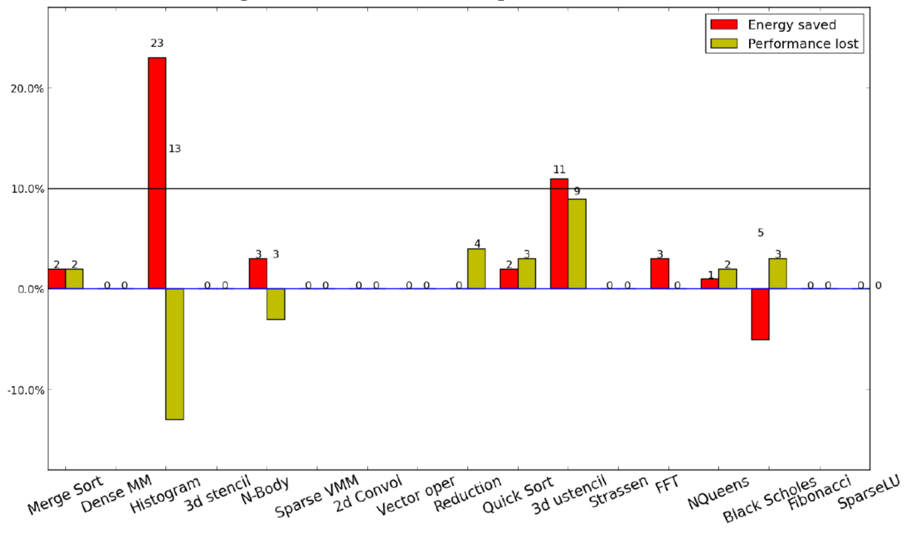

Fig5. Benchmark performance and energy savings, with agent optimizing for Energy Delay Product, 10% threshold

4.5 Accuracy of predictions

As a complete exploration of all possible agent predictions would require a result set of exhaustively testing all frequency variations with every 250ms interval of run time, it is infeasible to judge predictions relative to an empirically determined absolute optimum. As an approximation, we present the descriptive measure of the operating frequencies chosen averaged over runs, compared to the constant frequency that yields the best result. Table 3 presents the relative deviation between these numbers, averaged over all benchmarks for each optimization metric.

Table 3. Computational kernels used for verification only

Metric	Avg. error, training benchmarks	Avg. error, unknown benchmarks
Operations per Joule	5.20%	6.29%
Energy Delay Product	1.34%	1.53%
Operations per Joule, 10% threshold	2.29%	2.84%
Energy Delay Product, 10% threshold	1.31%	1.54%

5. Discussion

The primary result visible from the results presented in Section 4 is that the intelligent agent approach can successfully improve energy efficiency based on monitoring the behavior of a dynamic task pool at run time. Its effectiveness is related to the degree of parallelism, and the tolerance for performance degradation to the benefit of saving energy. As shown in Table 3, the approach of training the agent with a representative set of benchmarks does introduce an expected advantage in overall accuracy for the chosen programs, but relative deviations form measurements of statically assigned optimal choices remain in the single-digit percentile range.

Restricting the tolerated performance degradation through optimizing for Energy Delay Product provides a balanced tradeoff between performance and energy consumption, but severely restricts the range of frequencies available to the intelligent agent on this particular architecture. Restricting it by means of imposing a threshold on the absolute energy consumption relative to peak performance increases this flexibility to admit more program cases, but this provides only approximate control, because the bound is imposed based on inaccurate estimates of future performance. As the effectiveness of both techniques relies on dynamic power being a major component of total consumption, their effectiveness relative to each other should be expected to show altered characteristics when employed on platforms with significantly different balance of static and dynamic power consumption. As the utilized test platform has relatively high idle power consumption, this is a promising observation for deploying the approach on hardware constructed with greater emphasis on energy efficiency.

A noteworthy result from Figs. 3-5 is that the *Histogram* benchmark displays improvements in both energy and performance using dynamic adjustments. Further investigation of this phenomenon showed that this is due to its variable task intensity throughout a run. As the application runs in alternating parallel and sequential phases, the agent was able to detect intervals when cores remained idle, and temporarily reduce their operating frequencies, to an overall gain without significant disadvantages. A more detailed description of this effect can be found in [1].

6. Conclusions and future work

We have described the design of an on-line task scheduling plugin for OmpSs, capable of adapting energy use to dynamic application behavior. Our tests demonstrated its applicability to a range of task-based benchmark programs, with attainable energy savings on a high-end platform, suggesting that it makes a viable approach also for more energy-constrained designs. A natural extension of this work would be to verify this expectation by testing the approach on a greater range of platforms. Extending the design of the intelligent agent to explicitly recognize hardware with heterogeneous computation resources would also make an interesting direction for further development.

Acknowledgements

This work was financially supported by the PRACE-2IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493.

References

1. Thomas B. Martinsen, “Energy Efficient Task Pool Scheduler in OmpSs”, Master Thesis, Dept. of Computer and Information Science, NTNU, 2013.
2. Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas, ”OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures”, Parallel Processing Letters, Vol. 21, Issue 2, 2011.
3. OpenMP Architecture Review Board, “OpenMP Application Program Interface”, http://www.openmp.org/mp-documents/OpenMP3.1.pdf, retrieved on 10.07.2013
4. Andreas Weissel and Frank Bellosa, “Process cruise control: event-driven clock scaling for dynamic power management”, Proceedings of the 2002 international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES’02), pp. 238-246, ACM, 2002.
5. Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas, “Green governors: A framework for Continuously Adaptive DVFS”, Proceedings of the 2011 International Green Computing Conference and Workshops (IGCC), IEEE Computer Society Washington, 2011.
6. Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé, “Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP”, Proceedings of the 2009 International Conference on Parallel Processing (ICPP), IEEE Computer Society Washington, 2009.
7. PAPI project, “Performance Application Programming Interface”, http://icl.cs.utk.edu/papi, retrieved on 10.07.2013.
8. Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowitcz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick, “A view of the parallel computing landscape”, Communications of the ACM, Vol. 52, No. 10, pp. 56-67, ACM, 2009.
9. Jan C. Meyer and Lasse Natvig, “Power instrumentation of task-based applications using model-specific registers on the Sandy Bridge architecture”, PRACE White Paper, 2012.