Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Power instrumentation of task-based applications using model-specific registers on the Sandy Bridge architecture

Jan Christian Meyer[a,*] 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

This whitepaper describes the technical side of a research work into the energy-efficiency tradeoffs of task-based execution with vectorization, through the application of recently available model-specific registers for counting energy use. It describes the mechanisms used to extract energy figures with respect to architectural and operating system concerns, and illustrates their utility in the process of collecting appropriate benchmark figures to examine such tradeoffs. A subset of obtained results is presented as an example, highlighting both the potential and limitations of the outlined measurement method.

1. Introduction

Beginning with the Sandy Bridge microarchitecture, recent generations of Intel processors feature Model Specific Registers (MSRs) which enable the instrumentation of processor power consumption at run time. While the presence of these registers opens a wide range of research questions pertaining to the impact of programming techniques on power efficiency, their practical application presents a number of technical issues, which have implications for the development of sound methodologies in corresponding research work.

The goal of this whitepaper is to describe the challenges encountered in the process of producing a research paper which instruments power consumption using Sandy Bridge MSRs, document the techniques used to employ them, and summarize the implications the technique had for the scope of research.

2. Implementation issues

Through hardware generations supporting the IA-32 and Intel64 instruction set architectures, x86 processors have differentiated a set of *privilege levels*, that modify the view of the available instruction set based on the processor's current mode setting. This mechanism primarily separates the concerns of securing the operating system from the concerns of user applications. It is hierarchically organized in 4 levels, referred to as *ring* 0 through 3, in correspondence with an illustration in the software developer manuals' section on basic architecture [1]. Ring 0 is

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

intended for the O/S kernel, rings 1 and 2 for device drivers and system-level services, and ring 3 for user programs.

The general interface to obtain MSR values is the *RDMSR* instruction, which accepts a numbered index specific to the set of model specific extensions in a given processor. This instruction is restricted to ring 0 execution, at the discretion of the operating system kernel. Without explicit documentation, it remains easy to speculate that the design decision to restrict this instruction is one of security concerns; given the model-specific nature of the output values, general availability to user code might expose different information in response to identical binary code, which might at best be unfortunate for binary compatibility reasons, and at worst expose information which should otherwise be subject to an operating system's security model. For research purposes in High-Performance Computing (HPC) application space, this is an impediment to users of large-scale platforms, as the typical multi-user security and maintenance concerns of the environment neither permits applications to escalate their privileges, nor admit modifications of the operating system kernel without potentially invalidating vendor support agreements.

The Performance Application Programming Interface (PAPI) [2] is an undertaking to augment the Linux O/S kernel with an interface to expose a number of performance counters and similar information to users, while taking care of the aforementioned concerns within the operating system. Although there is an ongoing effort to integrate this in the general distribution of the Linux kernel, the time frame before its general availability on HPC systems can be expected prompted development of the method described in this whitepaper, and suggests that it may remain applicable for some time.

3. Method

Section 3.1 briefly details the design and construction of a library interface to leverage Sandy Bridge MSR values for energy instrumentation. Section 3.2 elaborates on the limitations a straightforward application of such a library imposes upon experimental design in a research setting.

3.1 Library construction

Although missing a direct programming interface to extract model specific registers, Linux exposes the MSR register values as a per-cpu device file, mapped as /dev/cpu/(*cpu-index*)/msr on our systems. Consistent with other files representing raw devices, reading rights for these files are assigned to administrative users only, but accessing values through the file system still relaxes the requirement of executing the code itself with ring 0 privileges.

For proof-of-concept testing, development was performed on an isolated system, using the *sudo* mechanism to provide reading privileges for user-space programs to open the msr device file. While this solution is not generally usable in a production environment, it enables prototyping for initial assessment of effectiveness and efficiency.

Sandy Bridge processors provide energy counters for the processor package, as well as divided in *powerplanes* 0 and 1 (pp0, pp1). Powerplane 1 represents an integrated graphics unit; this is not in use for the benchmarks reported here, and package energy counters follow pp0 closely, to within a small constant difference. The units of measurement are provided in the data sheets of the processor and hard-coded in the program logic of a library interface which decouples the instrumentation code from application logic.

The library interface presents a data structure containing a file descriptor, pairs of counters for the package energy, pp0, pp1 and time. Its state is manipulated using initialize, start, stop, estimate, and finalize functions.

Initialization amounts to opening the msr file descriptor for reading. Start and stop functions record the MSR counter values from fixed offsets in it. Note that file reading is implemented using *pread*, so as not to modify the position of the file handle structure. If advanced, the virtual file descriptor provides a continuous stream of data, but we wish to sample its state as near as possible to the time the function was called. Start and stop functions simply record raw counter values, as well as timestamps using *gettimeofday*. This keeps the overhead of the functions low, and permits the counter values to be converted into standard units offline, using the estimate function.

Counter values are found at model-specific offsets, specified in the processor reference manuals [1]. The timing resolution of the running energy estimate is approximated to 1ms; as the energy register is a simple counter mechanism, it is expected to wrap around at ≈ 60 seconds at high power consumption. The estimator function can correct this within a single period, by detecting when stop values are smaller than start values, and adjusting their difference accordingly. At this granularity, the unit of the energy counter works in multiples of 15.3 microJoules, and the estimation function derives energy consumption using this factor.

3.2 Methodological implications

Application of this instrumentation method impacts the scope of our initial research in applying it, as measured effects must take place within a 60 second interval, and the reported energy use only captures the effect of the processor, not including memory traffic. To obtain representative results for energy, the instrumentation library is therefore linked into application benchmarks keeping strict control of the input size, and producing estimates for the memory footprints of computational kernels when operating with a steady rate of computation. In order to minimize the deviation due to memory traffic, working sets are kept within LLC size; the resulting performance curves witness that this does limit the impact of memory traffic, as the applications begin to show memory-bound behavior beyond these problem sizes [3].

Because we focus on task-based applications, benchmark selection was guided by codes which were already available in task-based versions, or can be modified to become so with modest effort. A 1D FFTW kernel [4], the BlackScholes formula from the NAS parallel benchmarks [5] and a tiled matrix-matrix multiplication routine are chosen. The BlackScholes and matrix-matrix-multiplication codes were supplied by UPC/BSC [6] and the FFTW benchmark was adapted to the OmpSs [8] implementation of OpenMP task constructs by Hallgeir Lien [7].

Measurements for the FFTW benchmark were obtained using a 1-dimensional single-precision out-of-place complex transform, resulting in a memory footprint of 16N bytes for an N-element computation. The BlackScholes implementation uses 6 input and 1 output array of single-precision floating point numbers, resulting in a 28N byte memory footprint for an N-element computation.

Measurements for the FFTW benchmark were obtained using a 1-dimensional single-precision out-of-place complex transform, resulting in a memory footprint of 16N bytes for an N-element computation. The BlackScholes implementation uses 6 input and 1 output array of single-precision floating point numbers, resulting in a 28N byte memory footprint for an N-element computation. Matrix multiplication was carried out by tiled decomposition, using the ATLAS dgemm kernel for individual tile products.

4. Results

This section presents partial results from our research work, with the intention to illustrate the application of the presented instrumentation technique. For the analysis of further results and their significance, the reader is referred to Lien *et al.* [3].

The techniques described in 3.1 enable us to insert start/stop energy instrumentation exactly around the computationally intensive kernel sections of the benchmark codes. Deriving power consumption from energy counter states at controlled time intervals, and estimating computational performance using more classical techniques results in the energy efficiency plots in Figs. 1, 2 and 3, reproduced from the research paper [3]. They are obtained from averages of 10 runs, with a relative sample standard deviation within 3%.

This enables us to compare the performance impact of different balances in task threadpool size versus vectorization with respect to energy efficiency, subject to the aforementioned restrictions of data set size and computation time. For the selected kernels, timing runs remained within the 60 second limit, and the problem sizes which correspond to the cutoff point for last-level cache size is marked with a blue line in Figs. 1-3. An unknown inaccuracy in power estimates is expected to partly undermine the validity of results to the right of this line, but they are included in the figures as they clearly expose the transition to memory-bound computation, and thus validate the calculation of the upper bound on problem size.

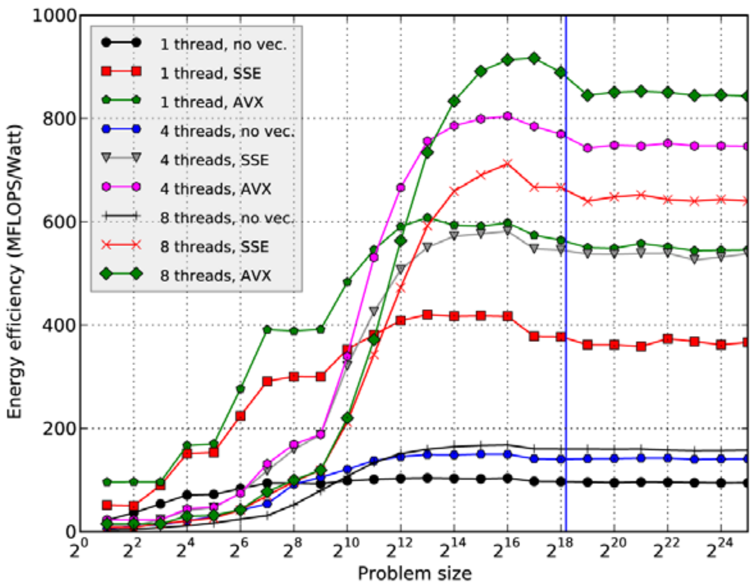

Fig 1. Energy efficiency of FFTW with various kernel threading/vectorization balances

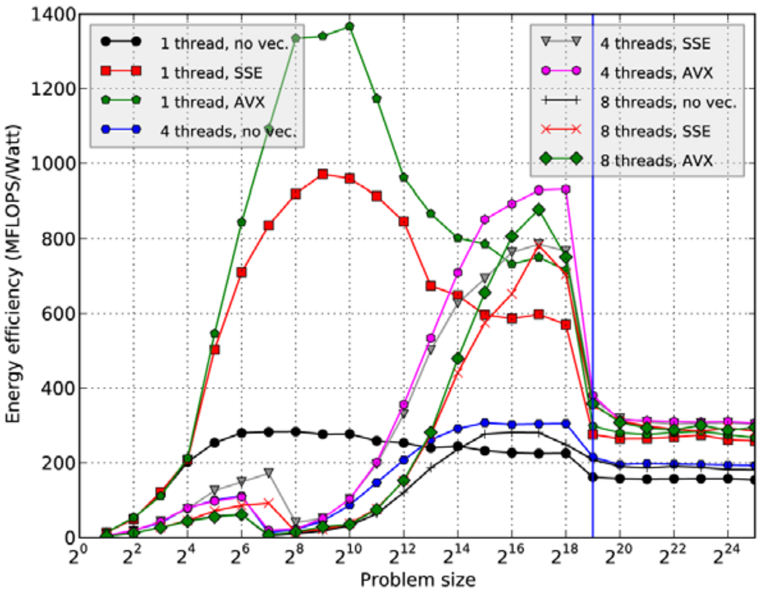

Fig 2. Energy efficiency of BlackScholes for with various kernel threading/vectorization balances

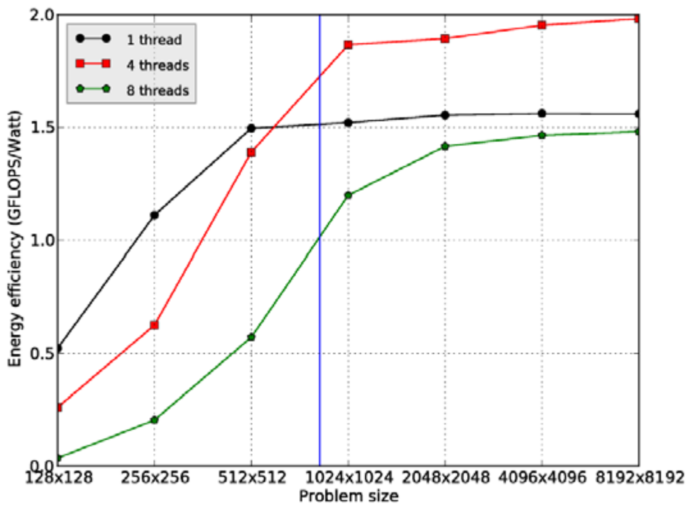

Fig 3. Energy efficiency with various thread counts for matrix multiplication

5. Conclusions and future work

We have described the technical design of a basic program library which extracts energy estimates from Sandy Bridge MSR values, and illustrated its application in context with published research work, highlighting the technical impact on research methodology. The limitations we have presented can be circumvented in several different ways, and there is ongoing and future work in extending the scope of application for the technique. An immediate direction for future work is to adapt the instrumentation code to provide continuous sampling data within the valid time frame, and apply it to identify the power intensive sections of run time without injecting the instrumentation into the application benchmark code. Another fruitful direction is to decouple the instrumentation code further from the application code, in an effort to make it manageable to execute securely without admitting user-space code to run with administrative privileges. This may enable migration of the proof-of-concept implementation onto larger scale infrastructure resources without violating their security policies.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. The authors would also like to thank Hallgeir Lien, Jørn Amundsen and Abdullah Al Hasib for their substantial contributions.

References

1. Intel® 64 and IA-32 Architectures Software Developer Manuals, version 044, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html on 17.12.2012
2. Performance Application Programming Interface, http://icl.cs.utk.edu/papi/, accessed on 17.12.2012

3. Hallgeir Lien, Lasse Natvig, Abdullah Al Hasib, and Jan Christian Meyer, “Case Studies of Multi-core Energy Efficiency in Task Based Programs”, in Proceedings of 2nd Int'l Conf. on ICT as Key Technology against Global Warming (ICT-GLOW 2012), A. Auweter et al. (Eds.): LNCS 7453, pp. 44–54.
4. M. Frigo and S. Johnson, “The Design and Implementation of FFTW3,” Proceedings of the IEEE, vol. 93, pp. 216 –231, feb. 2005.
5. Bailey, D.H. et.al., “The NAS parallel benchmarks summary and preliminary results”, Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, 1991, pp.158-165.
6. Code received from Barcelona Supercomputer Centre.
7. Hallgeir Lien, “Case Studies in Multi-core Energy Efficiency of Task Based Programs ”, Master Thesis, Dept. of Computer and Information Science, NTNU, 2012.
8. J. Perez, R. Badia, J. Labarta: A dependency-aware task-based programming environment for multi-core architectures, 2008 IEEE International Conference on Cluster Computing, pp. 142-151 (2008)