Investigating the Potential of Energy-savings Using a Fine-grained Task Based Programming Model on Multi-cores

Alexandru Iordan[1], Artur Podobas[2], Lasse Natvig[1], and Mats Brorsson[2]

[1] Norwegian University of Science and Technology
Trondheim, Norway
Email: {iordan,lasse}@idi.ntnu.no
[2] KTH Royal Institute of Technology
Stockholm, Sweden
Email: {podobas, matsbror}@kth.se

Abstract. In this paper we study the relation between energy-efficiency and parallel executions when implemented with a fine-grained task-centric programming model. Using a simulation framework comprised of an architectural simulator and a power and area estimation tool, we have investigated the potential energy-savings when employing parallelism on multi-cores system. In our experiments with 2 - 8 multi-cores systems, we employed frequency and voltage scaling in order to keep the relative performance of the systems constant and measured the energy-efficiency using the *Energy-delay-product.* Also, we compared the energy consumption of the parallel execution against the serial one. Our results show that through judicious choice of load balancing parameters, significant improvements of around 200 % in energy consumption can be acheived.

Keywords: Task Based Programming, energy-efficiency, multi-cores

1 Introduction

At the start of the new millennium, with performance being limited by high power budgets and heat dissipations requirements, the superscalar paradigm reached the point of diminishing returns. Faced with the constraint of *the power wall* [13], hardware developers were in need of new ways to efficiently use the ever increasing transistor count predicted by Moore's law.

Multi-core architectures have been a natural solution for the power wall: several less complex and significantly less "power hungry" cores are integrated on a single processor. Processors like Sun's Niagara T1 and T2, Tilera's Tile64 or IBM's Cyclops-64 use less complex cores, with shallow pipelines and simpler branch prediction, and lower clock speeds than previous CPU generations. Scaling down frequency (f) and supply voltage (V_{dd}) has a large effect on the chip's dynamic power, as shown by the relation: $power_{dynamic} \sim V_{dd}^2 \cdot f$. Developing parallel applications, that can take advantage of such chips has the potential of reducing energy consumption while still providing high performance.

This paper is a initial study of the impact of load-balancing on energy-efficiency in parallel executions. We use a very simple test scenario in which we increase the number of cores while proportionally decreasing their working speed. The applications in our experiments are developed using a paradigm called *Task Based Programming* (TBP). TBP organizes an application as a set of computational units (called *tasks*) that are scheduled across different cores. Parallel applications developed with TBP are known to handle irregular dependencies on input sets well and can adapt to varying computational load [17]. The base concept of the TBP model is that the programmer should identify and annotate pieces of code (tasks) which can be executed concurrently with other tasks, while the complexity of the hardware is abstracted away from him/her.

Generally, parallelization of applications has a cost of added overhead that sometimes scales badly, and this can increase the amount of energy that is used. In our study we employed a rather simple core design which we did not modify as we scaled up the number of cores. Our goal was not to find a way to develop a multi-core that was highly energy-efficient, but to investigate the potential to save energy by parallel executions. We used voltage and frequency scaling to keep the relative performance of the systems constant and to maintain the chip's power requirements under realistic values.

The rest of this paper is organized as follows: in Section 2 we give a brief introduction to power metrics, TBP and Wool library. Section 3 outlines the experimental methodology used in our experiments and Section 4 discusses our results. Section 5 presents related work and Section 6 describes our plans for future work. Section 7 concludes the paper.

2 Background

2.1 Power metrics

A well known metric that can balance performance and power requirement is *Performance*^{N}*/Watt*. The N parameter is used to increase or decrease the importance of the *Performance* component of the metric. For $N = 1$, this metric is used in the Green 500 list to rank world's most energy-efficient supercomputers [2].

Industry standard benchmarks like EnergyBench (for embedded systems) and SPECpower (for servers and multi-processor computers) use customized metrics to report on energy-efficiency. For example, SPECpower ranks a system using ssj_ops/Watt metric (stands for server side Java operations performed per Watt). This is a derived form of the *Performance/Watt* metric and it represents the number of the executed SPEC operations divided by the average power of the system.

Another frequently used metric is the *Energy-delay-product* (EDP). When comparing scenarios that do not alter instruction count, this metric is equivalent to the reciprocal of *Performance*^{2}*/Watt*. Offering equal weight to energy consumption and performance, EDP ensures a balanced energy-efficiency comparison among the test systems. Since our focus is on studying the trade-off of energy-savings and performance on multi-core systems, we choose to use EDP for our experiments.

2.2 Task Based Programming

A *task*, which can be fine-grained or coarse-grained, is a section of the code that performs some operations over a set of parameters [16]. *Task Based Programming* (TBP) is a programming paradigm that allows the parallelization of applications that can be divided in multiple tasks. TBP comes in contrast with *data parallelism* where the same operations are executed by different nodes (or cores) with different data [15]. Using TBP model, the programmer should identify pieces of code (tasks) to be executed concurrently with other tasks.

TBP libraries, like Intel's TBB [3] and Cilk++ [1] increases the productivity of programmers since details like distribution of work to a set of cores and message passing between parallel processes are abstracted away from the programmer. Membarth et al. [20] performed a comparative study of several frameworks for parallel programming on multi-cores. Their results showed that TBP libraries like Intel's TBB and Cilk++ not only perform better than other frameworks like OpenMP or OpenCL, but also have a wide usability and provide a better productivity for the programmer.

2.3 The Wool library

Our approach involves a lightweight TBP library called Wool [12]. Wool is a work-stealing parallel library that was designed with the ability to scale well with small tasks (smaller than a hundred cycles). These characteristics make Wool very efficient in dealing with work imbalance and also assure that it has very low overheads. The comparative study in [21] shows that Wool outperforms some other parallelization libraries (like Cilk++ or OpenMP) in terms of cycle costs for parallelization and task management operations.

Wool uses special data structures called *task queues* to store, manage and schedule the tasks for each worker thread. This differs from other models such as the GCC version of OpenMP which have one global queue containing all the tasks. Private queues generally improve performance of the system, since the locking-contention usually is much smaller compared to global queues. However, distributed queues face the challenge of balancing the work among them. More details and examples about Wool can be found in [12] and [21].

In Wool, load-balancing is implemented through the *randomly task stealing* technique. When a worker thread empties its own task queue, the stealing mechanism sweeps through all available worker queues to find available tasks to steal. Because making a task *stealable* adds an overhead, the programmer can control the number of stealable tasks per queue. In this way, the programmer has control over the load-balancing of the application.

Another scenario when task stealing is employed is when a worker is trying to synchronize with one of its children, and finds it stolen. In order to prevent threads from being idle a technique called *leap-frogging* is used. More details about this technique can be found in [22].

3 Experimental setup

3.1 Architectural simulations

Using the M5 simulator [7], we performed full-system simulations of several multi-core platforms. In a full-system simulation, the target system is able to run its own operating system and in our experiments we used the 2.6.27 Linux kernel. This type of simulation also makes it possible to record the behavior of all key components of the system: core, cache hierarchy, memory controller, main memory.

The basis for our modeled CPU is the Alpha 21364 processor from DEC. The choosing of this model was motivated by the fact that Alpha ISA is the most stable one for full-system multi-core simulations in M5 [6]. A second reason is that the 21364 was also validated in McPAT [19] (more details in section 3.2). In all multi-core systems simulated we used the same processor architecture with all parameters kept constant, except the three main parameters: number of cores, supply voltage and core frequency. We assumed voltage and frequency scaling at chip level so that we keep the power requirement under realistic values. We used the formula *core_frequency = single_core_frequency / number_of_cores* to maintain constant the relative performance of the test systems.

The maximum number of cores we simulated was limited only by the values for core frequency and V_{dd} (the way we calculated our voltage scaling values is described in section 3.2). The minimum value for V_{dd} we could assign was $2.3 * V_{th}$ [18] and in our case this limit is $0.19 * 2.3 = 0.44$ V.

The original 21364 has a clock frequency of 1.2 GHz and was produced in 180 nm technology. We assumed a 65 nm process technology and by linear scaling, similarly to Li and Martinez [18], we can determined the single-core frequency at 3.32 GHz. However, using this frequency for the single-core system and then scaling down V_{dd} results into voltage values very close to the $2.3 * V_{th}$ limit. For such low values of V_{dd} the chip's leakage current increases significantly and the static power can become dominant. To alleviate this we chose to assign a lower core frequency of 2 GHz to the single-core system. Since the speed of the cores does not affect the way they process the applications, just the execution time, the trends presented in our results are the same for both sets of experiments (the ones with a single-core running at 3.32 GHz and the ones with the single-core running at 2 GHz).

Table 1 lists the main characteristics of our experiments. Further details about M5's architectural parameters and characteristics listed in Table 1 and Table 2 can be found in [6] and [7].

The modeled system uses a cache hierarchy with split data and instruction private L1 caches. All cores share a 2 MB on-chip L2 cache through a common bus and implement a MOESI cache coherence protocol. Details about the cache hierarchy are given in Table 2.

Table 1. Main characteristics of modeled processor core

Parameter	**Value**
Process technology	65 nm
Nominal V_{dd}	1.1 / 0.89 / 0.82 / 0.78 / 0.76 / 0.74 / 0.73 / 0.72 V
V_{th}	0.19 V
Type of execution	Out-of-Order
Instruction set	Alpha
No. Cores	1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 cores
Clock frequency	2000 / 1000 / 666 / 500 / 400 / 333 / 285 / 250 MHz
Fetch/issue/commit width	4 / 4 / 4 insts./cycle
Inst. window size (Int / FP)	20 / 15 entries
Functional units	4 integer ALUs 1 integer multiply / divide 1 FP ALU 1 FP multiply/divide

Table 2. Cache parameters

Cache	**Size**	**Assoc. (bits)**	**Block size (bits)**	**Access latency**	**MSHRs Targets / MSHR**	**Banks**
L1 private iCache	32 KB	2	64	2	4 MSHRs /4 tgts	1
L1 private dCache	32 KB	4	64	2	4 MSHRs /4 tgts	1
L2 shared Cache	2 MB	4	64	10	4 MSHRs /4 tgts	4

3.2 Power estimations

Our simulation framework includes a power and area estimation tool called McPAT [19]. Developed in collaboration by HP-labs and the University of Notre Dame, McPAT models all major system components of a computer system (including in-order and out-of-order cores, network-on-chip, shared and private caches, memory controllers). Using information from the ITRS 2007 roadmap [4], McPAT supports design space exploration for single and multi-core architectures ranging from 90 nm to 22 nm production technology.

According to the *Process Integration, Devices, and Structures* chapter in ITRS 2007, there are 3 types of circuit logic: high performance (HP), low operating power (LOP) and low standby power (LSTP). HP devices include chips of high complexity and performance such as the microprocessors for desktop or server computers. LOP devices include relatively high-performance mobile circuits, like those in notebooks. LSTP devices are typically intended for low performance applications like cellular phones. We performed our estimations for the 65 nm technology and the HP device type.

McPAT is able to report both dynamic and static power. Dynamic power for each system component is defined as: $power_{dynamic} \sim AF \cdot C \cdot V_{dd}^2 \cdot f$, where AF is the activity factor, C is the total load capacitance, V_{dd} is the supply voltage and f is the clock frequency. AF is estimated using access statistics and component's characteristics provided by the architectural simulation of that component. The capacitance is computed with analytic models for each basic circuit block that makes up the system component. In addition to the dynamic power component McPAT also estimates the leakage power. As described in [19] the leakage current is estimated using MASTAR [5] and data from Intel.

The developers of McPAT validated the Alpha 21364 processor model we used in our simulations against published data. As the results in [19] show, McPAT is able to estimate the power requirements of the Alpha 21364 with an average error of 21 % (this value is highly influnced by the fact that validation was done for peak power and not average power).

In order to correlate the values for frequency and voltage scaling we resorted to a report from Intel [10] documenting this relation (in steps of 200 MHz) for three Pentium processors. From that report we extracted an average voltage step of 0.052 V per 200 MHz. This represents 3.88 % of the 1.34 V nominal voltage used for that family of processors. Since we simulated CPUs in the 65 nm technology process which has a 1.1 V nominal voltage [4], we calculated the voltage scaling step as 3.88 % of this nominal voltage for a frequency step of 200 MHz. By subtracting each core frequency in Table 1 from the single-core value and dividing the result by 200, we calculated the number of frequency steps. We used this number of steps to proportionally reduce V_{dd}. The resulting voltage values for each multi-core configuration we simulated are reported in Table 1.

3.3 Benchmarks

In our experiments we used a subset of the Barcelona OpenMP Task benchmark Suite (BOTS) [11]. BOTS is a benchmark compilation assembled by the Barcelona Supercomputing Center (BSC) to assess the performance of task-based programming models. Some of the kernels come from other benchmark collections (like *FFT* or *strassen*) and the others were written by the team at BSC (like *sparselu*). For our experiments, we changed the default OpenMP parallelization to a Wool implementation. This change was motivated by our wish to use a lightweight library with a low parallelization overhead. To allow for reasonable simulation times (from 12 hours to 2 days), the workloads we used are generally smaller than real-life problem sizes. Table 3 lists the size of the workloads we used in our experiments. The benchmarks have been cross-compiled for the Alpha ISA using a cross-compiler (consisting of gcc-4.3.2 and glibc-2.6.1) [6]. All benchmarks have been compiled using the flags: -O3 -static -pthread. A brief description of the BOTS subset that we used is given in section 4.

Table 3. Input workloads used in experiments

	Alignment	FFT	Fib	nQueens	SparseLU	Strassen
Input	20 proteins	512x512 matrix of floats	40'th element	13x13 board	100x100 sparse matrix of 20x20 blocks	1024x1024 matrix

4 Results

The main two issues that need to be addressed in order to improve a system's energy-efficiency when running parallel applications are parallel overhead and load imbalance. In order to quantify the effect of the two causes on energy consumption, we performed experiments in which we altered Wool's ability to deal with load imbalance. By controlling the number of tasks that each worker queue is allowed to mark as stealable, we also controlled the task distribution across worker threads. Taking into consideration that an imbalanced task-tree will also force the worker threads to perform more "management operations" (search for task to steal, successful/unsuccessful steals), modifying the number of stealable tasks also has effect on the overhead. We covered a wide range of testing points, from a minimum of 1 stealable task to 10000. There were some benchmarks (*alignment* and *sparselu*) for which the load imbalance for low values of stealable tasks was so large that its execution required an unreasonable long simulation time. For these applications we reduced the test range. For reasons of space limitations for this paper, Table 4 only lists the lowest EDP values and the corresponding number of stealable tasks. For the same reason, we do not report or discuss performance-orientated metrics like speed-ups.

Alignment is a protein alignment benchmark that is based on the Myers and Miller algorithm. In a *master-slave* manner, all the tasks in its execution are spawned by the main worker thread (the thread that is used to start the program). The children tasks, which do not spawn any other tasks, need to be available for the other workers. That is why the number of stealable task has a big effect on the energy-efficiency of the system executing this benchmark. All the other threads need to steal work from the main thread and a low number of stealable task leads to race contention among them. Our experiments showed a progressive decrease in EDP as the number of stealable tasks increased up to a threshold of around 200 tasks. After this point the improvements come at a much slower pace. For the stealability parameters listed in Table 4, the workload is almost perfectly balanced, and the energy consumption for all multi-core executions is below the single-core one (see Fig.1)

Fibonacci is a recursive benchmark that calculates the Fibonacci series of a given value n. The workload in the tasks are fine-grained, with leaf-nodes having only a single *if* and *return* statement. Using a recursive, divide-and-conquer approach, this application creates a very extensive task-tree which means that there is enough work for every worker. This application has good improvements when parallelized, with the workload evenly balanced among the workers even for low values of stealable tasks.

FFT calculates the Discrete Fourier Transform of a matrix in a recursive manner using the Cooley-Turkey algorithm. It showed good results when parallelized and all multi-core executions had an energy consumption under the single-core one (see Fig.1).

nQueens is a search-and-prune benchmark that generates all solution for the nQueens problem. It "builds" the solutions row by row and each valid position of a queen spawns a new task. Like Fibonacci, this is another application that benefits little from large numbers of stealable tasks. However nQueens does not generate a very large task-tree. The main worker thread starts the work by spawning a task for each valid position of a queen on the first row of the chessboard. The other workers steal these tasks and begin working with them. Since these tasks are coarse-grained, the workers need to steal fewer in order to keep busy. This application did not show improvements in energy consumption for all multi-core systems, as will be discussed later in the section.

SparseLU calculates the LU matrix factorization and the algorithm is fairly unbalanced. Just like Alignment, SparseLU spawns a relative small number of tasks and the challenge is to schedule them in a balanced manner among the worker threads. With the right number of stealable tasks (see Table 4), the workload imbalance is solved and the multi-core executions register a lower energy consumption than the single-core one.

Strassen is a parallel matrix multiplication algorithm. The algorithm subdivides the array into smaller arrays, and performs matrix multiplication on them. Like nQueens, Strassen showed partial improvements on energy-savings when parallelized. The parallel matrix multiplication executed faster on 2 and 3 cores compared to the 1 core system, but lost this advantage as the core count increased. Again, the reasons for this will be discussed later in the section.

Using the values for number of stealable tasks listed in Table 4, we performed a comparison of the parallel execution against the serial one. With these values, the performance of each benchmark on the multi-core systems is at a maximum (at least from the load-balancing point of view), so we only measured *Energy* for this study. Fig.1 presents this comparison. As you can see, for most configurations, the parallel executions show a higher energy-efficiency (marked by a lower energy) than the serial one. The biggest improvement recorded is for *Fibonacci* which shows a 239 % decrease in energy consumption when comparing the 3-cores execution to the single-core one. However, after a certain point the descendant trend of energy consumption stops for all benchmarks. There are three reasons for this behavior.

First, as the number of cores grows — so too are the scheduling and task management overheads. We recorded maximum increases of 52 % (for *Strassen*) in instruction count going from 2 cores to 8 cores.

Second is the less than linear algorithmic speedup of the parallel programs. In addition to this, the work stealing mechanism can induce stalls and serialization into execution. There are situations when a worker thread is forced to wait the completion of a task that was stolen from him. The *leap-frogging* technique can alleviate this problem only if the stolen task spawns children. A simple quantification of these behaviors can be made by examining the increase in execution time: 20 % increase when going from 2-cores to 8-cores for *Strassen*.

Third reason is the increase in static power as we scale down V_{dd}. Frequency and voltage scaling have a positive impact on dynamic power (when increasing the core count from 2 cores to 8, all executions showed an average of 43 % decrease of dynamic power), but they have the opposit effect on static power (leakage power to be more precise). Leakage power is influenced mainly by fabri-

Table 4. Best EDP values for multi-core test systems

		2 cores	3 cores	4 cores	5 cores	6 cores	7 cores	8 cores
Alignment	EDP	40.76	34.98	38.57	45.99	49.89	59.41	62.19
	No. of steals	900	900	1000	900	1000	900	900
Fibonacci	EDP	218.5	161.56	177.29	199.53	201.73	211.81	224.92
	No. of steals	3	4	4	4	4	3	4
FFT	EDP	217.36	181.28	172.94	205.42	222.66	245.46	267.35
	No. of steals	900	1500	3000	3500	7000	10000	10000
nQueens	EDP	1453.35	1498.70	1708.82	1978.66	2239.21	2581.41	2829.59
	No. of steals	4	5	5	6	9	9	7
SparseLU	EDP	79.39	53.83	58.06	64.09	71.58	80.46	88.50
	No. of steals	200	200	900	900	900	900	1000
Strassen	EDP	37.85	37.79	37.45	46.52	56.11	69.11	83.32
	No. of steals	9	100	90	90	900	900	900

cation parameters of the transistors (gate thickness, gate material etc.) but also by V_{dd} and V_{th} (the voltage at which the transistor is viewed as switched "on"). As we scale down V_{dd} and we get closer to the V_{th}, the leakage current increases and so is the static power. All multi-core configurations recorded an increase of 135 % of static power, when going from 2-cores to 8-cores.

5 Related work

There is a large body of work that focus on using parallel execution to improve performance and not energy consumption. What has got little interest thus far, at least to our knowledge, is quantifying the effect of TBP parallelization on energy efficiency on multi-core systems.

Li and Martinez [18] make a detailed power-performance exploration of parallel applications running on chip-multiprocessors. Using an analytical model, they perform extensive design space explorations to find the best multi-core configuration and also run simulations to verify the validity of the analytical results. They conclude that through judicious choice of parallelism's granularity and voltage/frequency scaling values, parallel computing can improve performance while maintaining or even reducing the power budget.

Contreras and Martonosi [9] make a study of Intel's *Threading Building Blocks* (TBB) and try to characterize some of the overheads associated with it. They emphasize the fact that this framework helps the programmer by abstracting away the complexity of the hardware. They also propose an improvement to TBB's task stealing mechanism in order to limit the parallelization overhead. Even though the focus of the authors is on performance, their implementation can also have a beneficial effect on the energy-efficiency of the system.

Sangyeun and Melhem [8] use an analytic framework to study the interplay between parallelism of an application, its performance and energy consumption. Their result demonstrate the advantage (quantified in energy or EDP) that can

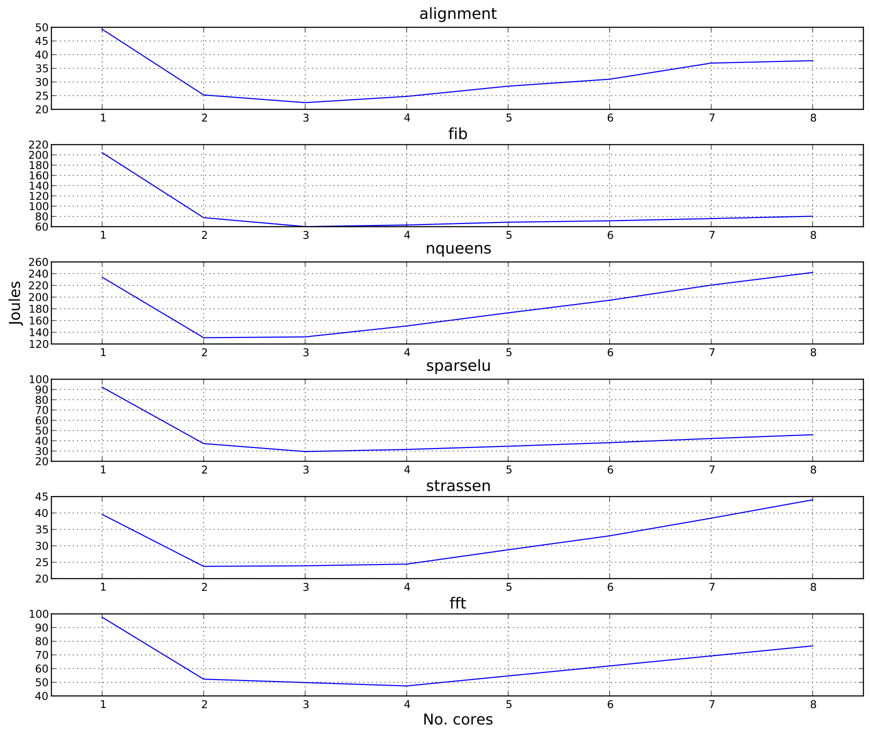

Fig. 1. Energy consumption comparison

be gained from employing dynamic voltage and frequency scaling to execute the serial and parallel part of an application at different levels of frequency and voltage. Also they study the scenario when individual processors can be turned off when not in use. Even if it assumes an simplified environment, this work provides valuable theoretical insights into energy-aware resource management.

There is a number of papers that is very relevant for the future development of our study. However, since they are not directly related to the current stage of the research, they are referenced in the next section.

6 Future work

We want to extend our experiments towards a larger number of cores and at the same time change the processor model. We are currently investigating an ARM model which is a much recent and scalable multi-core architecture. Also, a 4-core ARM Cortex-A9 evaluation board is commercially available and that makes it possible to validate our model and also to do experiments with both simulation and real execution.

We also plan to do a more detailed characterization of the specific mechanisms employed by Wool. Future work will focus on a detailed quantification in terms of EDP of Wool's load-balancing technique and possibly a comparison with other techniques (basic waiting, parking, etc. [12]).

Another approach to study the relation between task based parallelization and energy consumption is to use performance counters to track the behavior of real multi-cores. Weissel and Bellosa [23] and Goel et.al. [14] have been exploring power modeling with the use of performance counters. It is a long term goal for us to use similar approaches for achieving increased understanding of the energy issue, our models and their accuracy.

7 Conclusions

Although our study assumes a simplified environment and a simple test scenario we think it provides valid insights into how energy-efficiency and parallel execution relate. By integrating an architecture simulator (M5) with a power and area estimation tool (McPAT), we have put in place a framework for performance/power experiments. Using this framework, we studied the energy-efficiency of multi-core platforms running several BOTS benchmarks parallelized with the Wool library. Our experiments show improvements of the *EDP* metric when the parallel workload is balanced correctly for each benchmark and configuration.

Our experiments also show the potential for energy-efficiency improvements of parallel executions on multi-cores compared to the serial version of the same application on a single-core system. However, these improvements do not come for free. Task synchronization and management overhead, sub-linear speedups and increase in leakage power become more and more significant as the number of cores grows.

In all, we think the results we have found so far are promising and motivates for further research into energy-efficiency through parallelization.

References

1. Cilk++: A quick, easy and reliable way to improve threaded performance. `http://software.intel.com/en-us/articles/intel-cilk-plus/`.
2. The Green 500 list. `http://www.green500.org/`.
3. Intel Threading Building Blocks. `http://software.intel.com/sites/products/documentation/hpc/tbb/getting_started.pdf`.
4. International Technology Roadmap for Semiconductors 2007 Edition. `http://www.itrs.net/links/2007itrs/ExecSum2007.pdf`.
5. International Technology Roadmap for Semiconductors 2007 Edition, The Model for Assessment of CMOS Technologies and Roadmaps (MASTAR). `http://www.itrs.net/models.html`.
6. The M5 Simulator System webpage. `http://www.m5sim.org/wiki/index.php/Main_Page`.

7. N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4), 2006.
8. S. Cho and R. G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems*, 21(3):342 –353, 2010.
9. G. Contreras and M. Martonosi. Characterizing and improving the performance of Intel Threading Building Blocks. In *IEEE International Symposium on Workload Characterization*, pages 57–66, 2008.
10. Intel datasheet. Intel Pemtium M processor on 90 nm process with 2-MB L2 cache. `http://download.intel.com/support/processors/mobile/pm/sb/30218908.pdf`.
11. A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.
12. Karl-Filip Faxén. Wool - A work stealing library. *SIGARCH Computer Architecture News*, 36(5):93–100, 2008.
13. S.H. Fuller and L.I. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31 –38, jan. 2011.
14. B. Goel, S.A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *International Green Computing Conference*, pages 135 –146, aug. 2010.
15. W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.
16. C. Kessler and J. Keller. Models for parallel computing: Review and perspectives. In *Proceedings of PARS*, pages 13–29, 2007.
17. M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(1):1–47, 2004.
18. J. Li and J.F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 2:397–422, December 2005.
19. S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and many-core architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.
20. R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Frameworks for multi-core architectures: a comprehensive evaluation using 2D/3D image registration. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, pages 62–73, 2011.
21. A. Podobas, M. Brorsson, and K.F. Faxén. A comparison of some recent task-based parallel programming models. In *Third Workshop on Programmability Issues for Multi-Core Computers*, 2009.
22. D. B. Wagner and B. G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 208–217, New York, NY, USA, 1993. ACM.
23. A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 238–246, New York, NY, USA, 2002. ACM.