Studying The Impact Of Application-level Optimizations On The Power Consumption Of Multi-Core Architectures *

S M Faizur Rahman
Univ Of Texas At San Antonio
srahman@cs.utsa.edu

Jichi Guo
Univ Of Texas At San Antonio
jguo@cs.utsa.edu

Akshatha Bhat
Univ Of Texas At San Antonio
abhat@cs.utsa.edu

Carlos Garcia
Univ Of Texas At San Antonio
carlosg@cs.utsa.edu

Majedul Hoque Sujon
Univ Of Texas At San Antonio
msujon@cs.utsa.edu

Qing Yi
Univ Of Texas At San Antonio
qingyi@cs.utsa.edu

Chunhua Liao
Lawrence Livermore Nat. Lab
liach@llnl.gov

Daniel Quinlan
Lawrence Livermore Nat. Lab
dquinlan@llnl.gov

ABSTRACT

This paper studies the overall system power variations of two multi-core architectures, an 8-core Intel and a 32-core AMD workstation, while using these machines to execute a wide variety of sequential and multi-threaded benchmarks using varying compiler optimization settings and runtime configurations. Our extensive experimental study provides insights for answering two questions: 1) what degrees of impact can application level optimizations have on reducing the overall system power consumption of modern CMP architectures; and 2) what strategies can compilers and application developers adopt to achieve a balanced performance and power efficiency for applications from a variety of science and embedded systems domains.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*Processors* [Optimization, Compilers]; C.1.4 [**Computer System Organization**]: Processor Architectures—*Parallel Architectures*

Keywords

Power consumption, energy efficiency, application level optimization, compiler optimization

1. INTRODUCTION

Power consumption and dissipation is a primary concern in the design and deployment of modern architectures, from high-end supercomputers, to multi-core desktops, to energy efficient laptops, and to embedded chips in cell phones and MP3s. Since it critically determines the operating costs, cooling requirements, and failure rates of key architectural components, software should be made energy-aware and use the full power of computers only when necessary. Most Operating Systems can automatically scale down the voltage or frequency of microprocessors when they are idle. However, it is less clear how compilers and application developers can effectively optimize the energy efficiency of their applications while preserving a satisfactory level of performance.

This paper conducts an extensive experimental study to help compilers and developers make informed decisions when optimizing their applications for energy efficient computing. Based on an existing infrastructure for empirical tuning of application performance [18], we studied the variation of the overall system power consumption of multi-core architectures when using the machines to evaluate a wide variety of different applications using varying optimization and runtime configurations. The study has produced insightful answers to the following important questions.

- How much impact can application level optimizations make in terms of reducing the overall system power and energy consumption of modern CMP architectures?
- What strategies can compilers and application developers adopt to achieve a balanced performance and power efficiency?

The results are significant and can be used to guide future compilers and developers to effectively reduce the power consumption of their applications while achieving a balanced level of performance and energy efficiency. In particular, we found that for most benchmarks, while the variations in performance are much more dramatic than those in power consumption, there seems to be no direct correlation between the performance attained and the power consumption incurred by application-level optimizations. Similar levels of performance can often be attained while incurring dramatically different power consumptions. As a result, collectively optimizing both the performance and power efficiency of applications is not only possible, but also immensely profitable.

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 presents the benchmarks used in the evaluation and our experimental methodology. Section 4 presents the overall variations of power consumptions when running the benchmarks with different optimization and runtime configurations. Sections 5 through 7 study the impact of varying groups of optimizations. Finally, Section 8 presents our conclusions.

*This research was supported in part by the National Science Foundation under Grants 0833203, 0747357, 0855247, and by the Department of Energy under Grant DE-SC0001770

CF'12, May 15–17, 2012, Cagliari, Italy.
ACM 978-1-4503-1215-8/12/05.

id	name	suite	parallel.	description
1	jacobi7	kernel	pthread	7 point jacobi stencil kernel
2	jacobi27	kernel	pthread	27 point jacobi stencil kernel
3	gauss7	kernel	pthread	7 point gaussian stencil kernel
4	gemm	kernel	openmp	Matrix-matrix multiplication
5	gemv	kernel	openmp	Matrix-vector multiplication
6	ger	kernel	openmp	Vector-vector multiplication
7	BT	NPB	openmp	Block Tridiagonal PDE solver
8	CG	NPB	openmp	Conjugate Gradient
9	DC	NPB	openmp	Data Cube operator
10	EP	NPB	openmp	Embarrassiingly Parallel
11	FT	NPB	openmp	Fast Fourier Transform
12	IS	NPB	openmp	Integer Sort
13	LU	NPB	openmp	non-linear PDE solver
14	LU-hp	NPB	openmp	The hyperplane version of LU
15	MG	NPB	openmp	MultiGrid solver
16	SP	NPB	openmp	Scalar Pentadiagonal PDE solver
17	Black-scholes	PARSEC	openmp, pthread, *tbb*	Computational Financial Application
18	Bodytrack	PARSEC	openmp, pthread, *tbb*	Realtime Computer Vision Application
19	Stream-cluster	PARSEC	pthread, *tbb*	Machine Learning Application
20	Swaptions	PARSEC	pthread, *tbb*	Computational Financial Application
21	Dedup	PARSEC	pthread	Enterprise Storage Kernel
22	Freqmine	PARSEC	openmp	Data-mining Application
23	Fluid-animate	PARSEC	pthread, *tbb*	Animation Application
24	Canneal	PARSEC	pthread	Engineering Application
25	Raytrace	PARSEC	pthread	Rendering 3D Graphics
26	Vips	PARSEC	pthread	Multimedia Application
27	x264	PARSEC	pthread	Multimedia Application
28	Facesim	PARSEC	pthread	Animation Application
29	DMR	LoneStar	Galois	Mesh Refinement Application
30	DT	LoneStar	Galois	Mesh Generation Application
31	SP	LoneStar	Galois	Heuristic SAT-solver
32	adpcm	MiBench	Serial	Adaptive Differential Pulse Code Modulation
33	basicmath	MiBench	Serial	Mathematical calculations
34	bitcount	MiBench	Serial	Bit manipulation
35	CRC32	MiBench	Serial	Cyclic Redundancy Check
36	djkstr	MiBench	Serial	shortest path algorithm
37	FFT	MiBench	Serial	Digital signal processing
38	jpeg	MiBench	Serial	Image compression
39	patricia	MiBench	Serial	Traverse sparse leaf trees
40	qsort	MiBench	Serial	Sorting of large arrays
41	sha	MiBench	Serial	Secure hash algorithm
42	susan	MiBench	Serial	Image recognition package

Table 1: Benchmarks used for experiments

2. RELATED WORK

Power consumption and dissipation has long been an important concern in CPU design, especially for embedded systems [14, 4]. Optimizations for power management have traditionally focused on *dynamic voltage and frequency scaling* (DVFS) by operating systems [8, 5]. Compiler optimizations to enhance power efficiency of applications have mostly focused on varying instruction scheduling schemes [13, 16, 22, 26] and thread-allocation and scheduling strategies [2, 1, 23]. Power consumption has been estimated using architectural simulation [4, 16, 2], offline profiling [6], and real time monitoring of hardware counters [23, 18]. This paper investigates the impact of a wide variety of compiler optimization and runtime configurations on the overall system level power with DVFS turned-off to minimize OS interferences.

Valluri and John [24] studied the impact of different compiler optimization levels on processor power/energy on a Dec Alpha 21064 CPU. Kandemir et al [11] studied the power/energy impact of both low-level compiler optimizations (instruction scheduling and register assignments) and three loop reordering optimizations (loop interchange, tiling, and unrolling) on both the CPU and the memory system using a matrix multiplication computation. Seng and Tullsen [21] studied the power/energy effects of different compiler optimization levels and three specific optimizations (loop unrolling, vectorization, and function inlining) on an Intel Pentium 4 Processor. In contrast, our work uses a much more extensive collection of benchmarks to empirically study the impact of both compiler optimization and runtime configurations on the overall system power consumption and dissipation of modern CMP architectures. Our objective focuses on providing insights to guide future compiler and application-level optimizations, with supporting compiler-architecture co-design only as a secondary goal.

3. EXPERIMENTAL DESIGN

We have studied 42 benchmarks, summarized in Table 1, by evaluating these benchmarks under varying compiler optimization and runtime configurations on two multi-core architectures. The following first presents details of the benchmarks and then summarizes our tuning infrastructure and methodology of data collection.

3.1 Benchmarks

Table 1 summarizes the 42 benchmarks we used in our evaluation, where each benchmark is given a unique integer identifier. Some benchmarks, e.g., those in rows 1-16, include both multi-threaded and sequential implementations, while others include only sequential implementations (e.g., rows 32-42) or only multi-threaded implementations (e.g. lines 17-31). The following further categorizes these benchmarks into five groups.

3.1.1 Matrix And Stencil Kernels

We have selected six scientific kernels: three dense matrix computations (rows 4-6) and three stencil codes (rows 1-3), to study the impact of finely parameterized loop optimizations. Using POET [28], an interpreted program transformation language designed to support the fine-grained parameterization of compiler optimizations, a specialized optimization script is built for each kernel so that the configurations of these optimizations can be empirically tuned [19, 27]. Each matrix kernel is optimized with 6 optimizations: OpenMP loop parallezation, loop blocking, array copying, loop unroll-jam, scalar replacement, and innermost loop unrolling [27]. Each stencil kernel is parallelized with three strategies, single time step parallelization, pipelining, and wavefront parallelization, with the locality within each thread enhanced with loop blocking combined with time-skewing [19]. Each POET script can be reconfigured via command-line options so that the blocking and unrolling factor of each loop can be arbitrarily adjusted, and each optimization can be optionally turned off.

3.1.2 NAS Parallel Benchmarks (NPB) [10]

This is a set of programs derived from computational fluid dynamics applications (rows 7-16 of Table 1), and we have selected their OpenMP [7] implementations in NPB version 3.2 [10]. We have compiled each of these benchmarks without any source level modification, using vendor compilers with varying optimization levels, and have evaluated them with different thread allocation and scheduling policies.

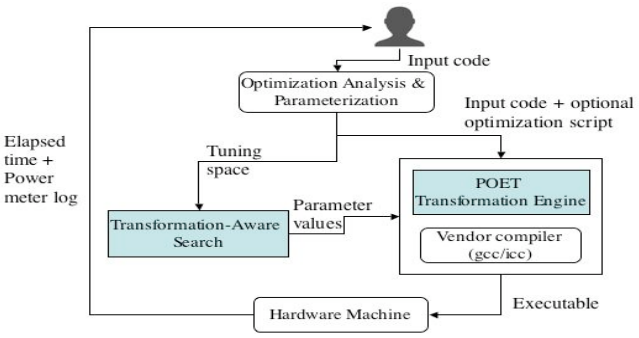


Figure 1: Empirical Tuning Workflow

3.1.3 `PARSEC` *benchmarks [3]*

The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite focusing on next generation shared-memory programs for chip-multiporcessors. It includes applications in recognition, mining and synthesis (RMS), and the applications mimic large-scale multi-threaded commercial programs using POSIX threads [15], OpenMP [7], and Intel Threading Building Blocks (TBB) [20]. We used all available parallel implementations and compiled them with no modification using vendor compilers with different optimization flags.

3.1.4 `LoneStar` *benchmarks [12]*

The LoneStar suite includes C++ applications which use pointer-based irregular data structures (e.g. graphs, trees) and amorphous data-parallelism, where an initial set of tasks which process composite data structures dynamically create additional tasks during execution. We selected three of these benchmarks (rows 29-31 of Table 1) which have been parallelized using the Galois system [17], which provide runtime support for speculative execution of tasks. We used the POET transformation engine, shown in Figure 1, to parameterize their invocations to the Galois library with varying runtime task scheduling policies (e.g., global and distributed task queues and stacks with different chunk sizes).

3.1.5 `MiBench` *benchmarks [9]*

This is a set of commercially representative embedded programs of six categories: Automotive and Industrial Control, Netowrk, Security, Consumer Devices, Office Automation, and Telecommuncations. We selected 11 applications from this suite (rows 32-42 of Table 1). All benchmarks are serial, and we have compiled them using vendor compilers with varying optimization flags.

3.2 Tuning Infrastructure

Figure 1 shows the workflow of our empirical tuning infrastructure, which we use to collect runtime statistics of the 42 benchmarks shown in Table 1. For each benchmark, we have determined its tuning space based on the underlying runtime model (i.e., sequential vs. multi-threaded) and the varying runtime configurations that are applicable (e.g., using different numbers of threads and different task scheduling policies). We compiled the `NAS, PARSEC,` and `MiBench` benchmarks only using vendor compilers with varying optimization flags. For the matrix and stencil kernels and the `LoneStar` benchmarks, however, we have used POET [28], an interpreted program transformation language, to support the fine-grained parameterization and tuning of optimizations. As shown in Figure 1, an optional POET optimization script was developed, either manually [19] or automatically via an optimizing compiler [27], for each of these benchmarks, and an optional POET transformation engine was invoked to apply additional source-level optimizations before invoking the vendor compilers.

3.3 Empirical Data Collection

We performed our experiments on two multi-core machines: an 8-core Dell Precision T7500n workstation with two 2.27GHz Intel Xeon quad-core processors (each with 4*256KB L2 and 4MB shared L3 cache) and 4GB memory; and a 32-core HPX workstation with four 2.0GHz eight-core AMD Opteron processors (each with 8*512KB L2 and 12MB shared L3 cache) and 16*4GB memory. For each machine, we connected a *Watts Up PRO 99333* power meter [25] with its power supply to log down the overall system power consumption per second of the machine while evaluating each benchmark in Table 1 with varying compiler optimizations and runtime configurations.

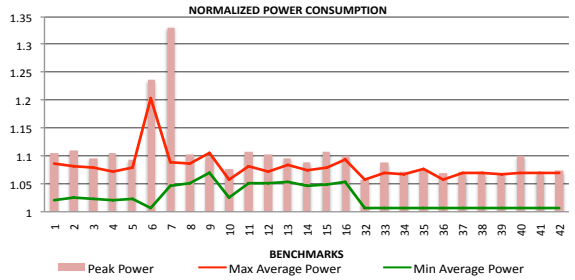
To ensure the accuracy of measurement, our scripts ran each benchmark continuously for at least 5-6 seconds so that the power meter can accurately log down power consumption of the duration. While using the machines to compile and run each benchmark, each machine is left completely idle otherwise. After each evaluation, we let the machine cool down until the power meter reading goes back to the lowest threshold before starting the next evaluation, to ensure statistics of different runs are completely independent of each other. Besides the power meter readings, while evaluating each benchmark, we collected both the elapsed time and the number of floating point and integer operations performed during each evaluation.

All benchmarks were compiled using gcc 4.4.4 and icc 11.1 on the 8-core Intel machine, and were compiled using gcc 4.6.1 on the 32-core AMD machine. For each runtime configuration, we repeated the evaluation three times and report the average of the measurements. The variations among repetitive evaluations are very minor as the machines were kept free of external interferences during the evaluations.

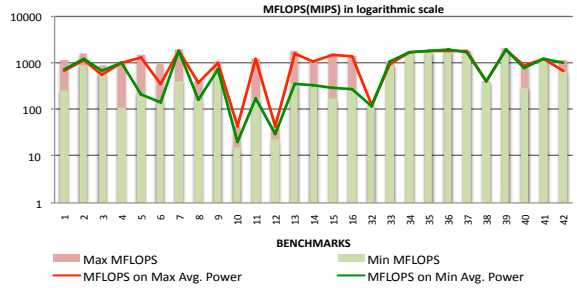
3.4 Terminologies

We use the following terminologies throughout the rest of the paper in discussion of our experimental results.

- *Peak Power*: the highest power consumption rate, in terms of *Watts*, attained at any moment across all evaluations of each benchmark;
- *Max (or Min) Average Power*: here the power consumption rates attained throughout the evaluation of each configuration are averaged for the duration, and the highest (or lowest) average rate among all configurations of the same benchmark is reported;
- *Max (or Min) MFLOPS*, the highest (or lowest) MFLOPS (Millions of floating point operations per second) rate attained among all configurations of each benchmark; For benchmarks 29-31, MIPS (Millions of integer operations per second) rates are used instead;
- *MFLOPS on max (or min) average power*, MFLOPS of the corresponding configuration that attained the *Max* (or *Min*) average power for each benchmark.

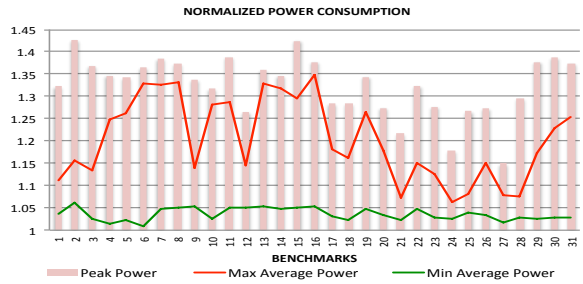


(a) Power normalized against machine idle power(154W)

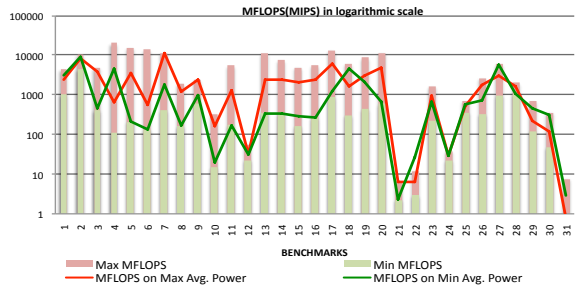


(b) *MFLOPS* of benchmarks

Figure 2: Sequential benchmark on 8-core Intel



(a) Power normalized against machine idle power(154W)



(b) *MFLOPS* of benchmarks (benchmarks 29-31 use *MIPS*)

Figure 3: Parallel benchmark on 8-core Intel

4. POWER CONSUMPTION VARIATION

Figures 2-4 show the overall variations of performance and power efficiencies when evaluating different benchmarks with varying compilation and runtime configurations.

From Figure 2(a), for sequential benchmarks, the power consumption peaks at 133% of the baseline on the 8-core Intel (we omitted evaluating sequential codes on the AMD machine as their runtime configurations are similar to those on the 8-core Intel), and the *average power consumption* are between 101%-120% of the machine idle power baseline. When running multi-threaded implementations of the benchmarks, the power consumption peaks at 143% of the machine idle power consumption baseline on the 8-core Intel machine in Figure 3(a) and at 170% of the baseline on the 32-core AMD in Figure 4(a). The *average power consumption* range between 101%-135% of the baseline on the 8-core Intel and between 101-165% on the 32-core AMD.

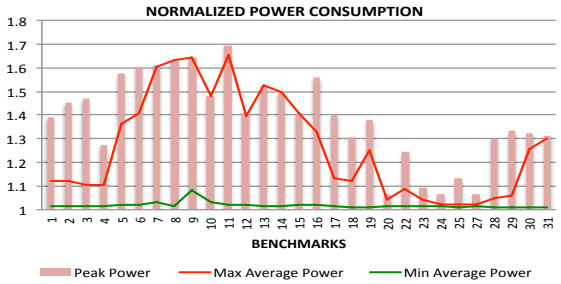
From Figures 2-4(b), the variations in performance are much more dramatic, ranging from 1% (benchmark 41 in Figure 2(b)) to orders of magnitude (e.g., benchmarks 19 and 20 in Figures 3-4(b)). So optimizing for performance naturally lead to better overall energy savings in most cases, as energy consumed = average power consumption * execution time. However, for some benchmarks, e.g., 32-39 and 41 in Figure 2, where compiler optimizations makes little difference in performance, the optimizations can instead focus on reducing the power consumption of the benchmarks, where up to 10% of energy saving may be resulted.

In Figures 2-4(b), on top of max and min MFLOPS (MIPS) rates attained for each benchmark, we plotted the corresponding MFLOPS (MIPS) rates when incurring the max and min average power consumptions. On the 8-core Intel, two parallel benchmarks (7, 9) in Figure 3(b) and eight sequential benchmarks (8-11, 12-16) in Figure 2(b) attained max MFLOPS rates while incurring the max average power consumptions. On the other hand, two parallel benchmarks (18, 27) in Figure 3(b) and eight sequential benchmarks (4, 7, 33-36, 39, 41) in Figure 2(b) attained max MFLOPS rates while incurring the least average power consumption. Further, for many benchmarks, incurring the min average power has attained comparable performance as that attained while incurring max power. In Figure 4(b), on the 32-core AMD, although incurring max power consumption has resulted in significantly better performance by employing a larger number of threads, the performance difference is insignificant for a number of benchmarks (e.g., 20, 24, 27, 29), and min average power has resulted in the best performance for 2 of the benchmarks (30, 31).

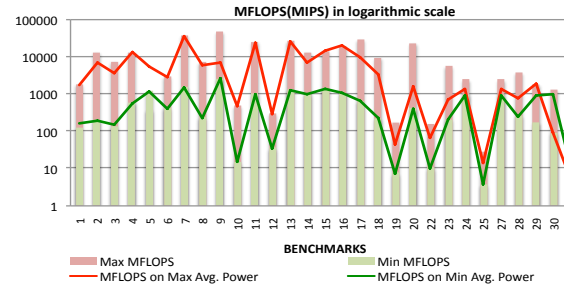
Therefore, when the inherit amount of concurrency within an application is limited, there seems to be little direct correlation between the performance levels attained and the power consumptions incurred by application-level optimizations. A pattern can be observed that when performance optimizations are ineffective (i.e., when performance enhancement is difficult), the optimization can instead focus on reducing the power consumption to achieve significant energy savings. Further, similar levels of performance can often be attained while incurring dramatically different power consumption, so collectively optimizing both the performance and power efficiencies of applications is not only possible, but also profitable, for a majority of the applications.

5. COMPILER OPTIMIZATION LEVELS

To investigate the overall trend of how applications are impacted by the aggressiveness of compiler optimizations, Figures 5-6 compare the MFLOPS and average power consumption attained when compiling the sequential and parallel implementations of each benchmark using the Intel *icc* compiler on the 8-core Intel machine with -O0, -O1, -O2, and -O3 optimization flags, which instruct the compiler to apply no optimizations, to optimize for speed (-O1 disables some optimizations that increase code size), and to apply additional loop transformations and data prefetching for improved memory-usage efficiency respectively. We study op-

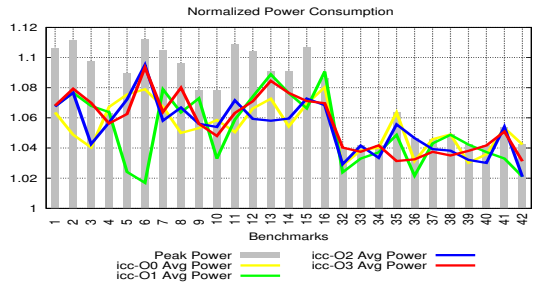


(a) Power normalized against machine idle power(360W)

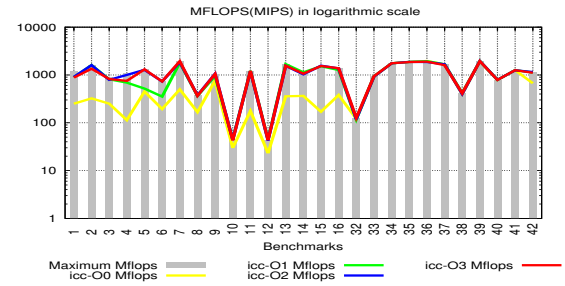


(b) *MFLOPS* of benchmarks (benchmarks 29-31 use *MIPS*)

Figure 4: Parallel benchmark on 32-core AMD



(a) Power normalized against machine idle power(154W)



(b) *MFLOPS* of benchmarks

Figure 5: Impact of compiler optimization levels on Sequential benchmark in 8-core Intel

timization levels of *icc* because of its known effectiveness in optimizing for Intel processors. The 32-core AMD machine uses only *gcc*, so its statistics are omitted here. To isolate the impact of vendor compiler optimizations, for the matrix and stencil kernels, we have disabled all the POET optimizations except OpenMP parallelization. For all benchmarks, we have used 8 threads with the default runtime thread scheduling policy to evaluate their parallel implementations.

From Figures 5-6(b), compiling using icc with O1- O3 flags makes little difference in the performance levels achieved, although all of them are able to significantly improve the performance levels attained by -O0 by a large margin for about half of the benchmarks. For the other half (benchmarks 1-3, 21-31,32-41), all the optimization levels attained similar performance as icc failed to decipher the complex data flow within these codes. Overall, the different optimization levels are fairly consistent in terms of affecting the performance levels of different benchmarks.

However, when looking at their impact on the average power consumption in Figures 5-6(a), the trend is rather mystic. In particular, in Figure 5(a), the average power consumption of differently optimized sequential codes vary by up to 5% (benchmark 35), but there does not seem to be any correlation between the relative performance and power efficiencies of the differently optimized code. In Figure 6(a), for the multithreaded implementations, the power variation is slightly bigger, by up to 11% (benchmark 6). While neither -O2 nor -O3 has a clear advantage over each other, they triggered similar or lower power consumption than -O0 for a majority of cases. Their better power efficiency is likely a benefit of the reduction of synchronization overhead by the OpenMP optimizations which are disabled by -O0.

Although the performance levels attained by icc -O2 and -O3 flags are almost identical, the optimized codes in many cases differ by a nontrivial margin in their average power consumptions, both for sequential and multithreaded implementations. This again indicates that reordering optimizations such as loop transformations and data prefetching can often be used to reduce power consumption of applications without sacrificing performance.

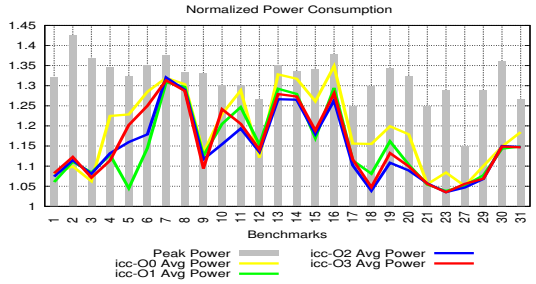
6. IMPACT OF MULTI-THREADING

Multi-threaded applications typically consume more power than sequential codes as they make more CPU cores busy. The performance and power consumption of their evaluations are directly impacted by the varying numbers of threads being used as well the the allocation and scheduling policies for the concurrent threads.

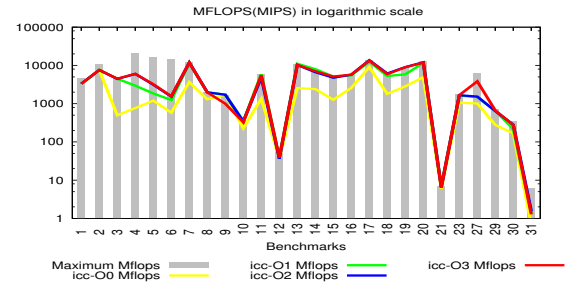
6.1 Varying Numbers Of Threads

Figures 7-8 compare the performance and power efficiencies of evaluating the multithreaded benchmarks using varying numbers of threads on the 8-core Intel and the 32-core AMD machines. Within each figure, the average power consumption and MFLOPS rates when evaluating each benchmark using different numbers of threads are plotted on top of the highest power and MFLOPS rates across all different configurations.

From Figures 7-8(a), on both machines, the average power consumption are relatively sorted in increasing order as the number of threads used to evaluate each benchmark increases. In Figure 7 (a), the ordering is violated only in two cases for benchmarks 5 and 16, where using two threads had incurred higher average power consumption than using four threads. In Figure 8(a), four similar cases (benchmark 15, 17, 25 and 28) exist on the 32-core AMD, where using 32 threads has incurred less power consumption than the other configurations. The power consumption variations range from 1% - 25% on the 8-core Intel in Figure 7(a) and from 1% - 40% on the 32-core AMD in Figure 8(a). From Figures 7-8(b), using more threads has resulted in better overall performance in most cases on both machines in a consistent fashion.

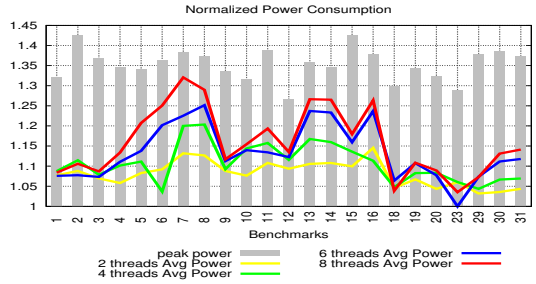


(a) Power normalized against machine idle power(154W)

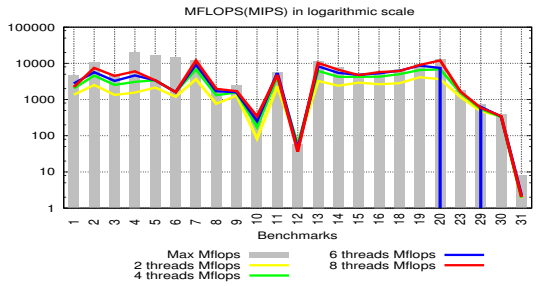


(b) *MFLOPS* of benchmarks (benchmarks 29-31 use *MIPS*)

Figure 6: Impact of compiler optimization levels on Parallel benchmark in 8-core Intel



(a) Power normalized against machine idle power(154W)



(b) *MFLOPS* of benchmarks

*There is no data for benchmark 23 using 6 threads because the benchmark requires the total number of threads to be a power of 2.

Figure 7: Impact of varying numbers of threads on 8-core Intel

We believe the small number of anomalies in Figures 7-8(a) are due to the default thread scheduling policies on the machines failing to group related threads on the same sockets. As discussed in the following, the overhead of the extra data movement across the sockets could be prohibitive when using inefficient thread scheduling schemes.

6.2 Varying Scheduling Policies

The efficiencies of multithreaded applications are often affected by how their concurrent tasks are distributed to different threads and how the threads are scheduled on different CPU cores. Figures 9-10 show their impact on the performance and average power consumption of 16 selected benchmarks on both the 8-core Intel and the 32-core AMD machines. These selected benchmarks are parallelized either through OpenMP, where we used the OMP_SCHEDULE environment variable to specify both a scheduling policy (STATIC, DYNAMIC or GUIDED) and a chunk size for the GUIDED policy, or through amorphous data parallelism, where we used POET scripts to vary the dynamic task scheduling policies using ChunkedFIFO, ChunkedLIFO, dChunkedFIFO or dChunkedLIFO and associated chunk sizes for the three LoneStar benchmarks (29-31). The other parallel benchmarks are not selected as it is more difficult to modify their scheduling policies. For each selected benchmark, *Max (or Min) average power* stands for the highest (or lowest) average power consumption rate across different scheduling policies and associated chunk sizes, and *MFLOPS on max (or min) average power* stand for their MFLOPS rates when attaining the corresponding power consumption.

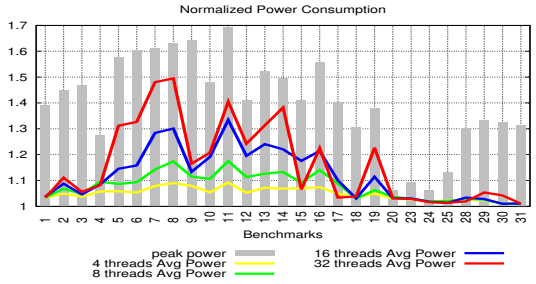
From Figure 9(b), on the 8-core Intel, the varying schedule policies made little difference in the overall performance of the benchmarks, as the Max and Min *MFLOPS* are within 1-2% of each other for all benchmarks. Their impact on the average power consumption in Figure 9(a), however, is more significant, where the variation is up to 8% of the machine idle power for two OpenMP NAS benchmarks and is close to 20% for the three LoneStar benchmarks. Since the 8-core Intel machine has only two sockets, and 8 threads are used to evaluate each benchmark, the dynamic scheduling of tasks may not significantly impact the performance. However, the movement of data resulted from different scheduling policies can significantly impact the power/energy consumption of applications, even on a machine with only two sockets.

From Figure 10(a), a similar degree of variation on the average power consumption can be observed on the 32-core AMD machine, where 5%-28% of variation is observed for the NAS and LoneStar benchmarks. However, as the number of CPU cores increase, the performance of applications can also be dramatically impacted by the scheduling policies, by up to factors of 7 (benchmark 9) in Figure 10(b).

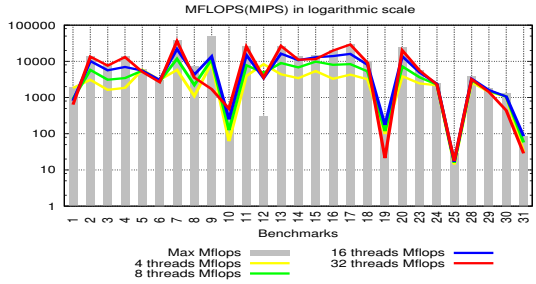
An interesting trend is that when incurring the highest average power consumption, the corresponding MFLOPS rates are either the best among all other scheduling policies, or the worst. A natural speculation from this trend is that when the extra data movements among the CPU cores are necessary and productive, they lead to the best performance. But when they are unnecessary, they clog the communication channels of other useful data movements and lead to the worst performance. Note that in many cases the best performance is attained while incurring the lowest power consumption. Therefore adapting scheduling policies to minimize unnecessary data movements can dramatically improve both the performance and power efficiencies of multithreaded applications on many-core architectures and should be exploited whenever possible.

6.3 Thread Affinity

Data locality is an important factor that often critically determines the overall performance of multi-threaded appli-

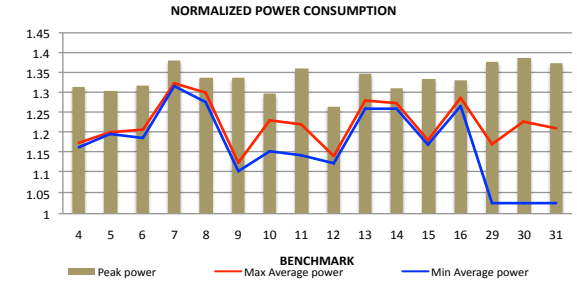


(a) Power normalized against machine idle power(360W)

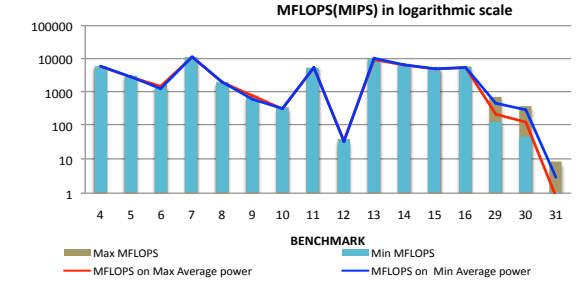


(b) *MFLOPS* of benchmarks (benchmarks 29-31 use *MIPS*)

Figure 8: Impact of varying numbers of threads on 32-core AMD



(a) Power normalized against machine idle power(154W)



(b) *MFLOPS* of benchmarks (benchmarks 29-31 use *MIPS*)

Figure 9: Impact of scheduling policies on 8-core Intel

cations. Our 32-core AMD and 8-core intel machines have 4 and 2 sockets respectively. To investigate the impact of placing related threads on different sockets, we pinned down each thread to a specific core using environment variables (e.g., GOMP_CPU_AFFINITY for the gcc compiler) for OpenMP applications and using library calls for applications parallelized with Pthread.

Figures 11-12 show the impact of evaluating the selected benchmarks with varying thread affinity configurations on the 8-core intel and 32-core AMD machines, where *Max (or Min) average power* stand for the highest (or lowest) average power consumption across different affinity configurations, *MFLOPS on max (or min) average power* plot their MFLOPS rates when incurring the corresponding power consumption, *Max (or Min) MFLOPS* stand for the highest and lowest MFLOPS rates achieved across different affinity configurations, and *Average power on max (or min) MFLOPS* plot their corresponding average power consumption.

From Figures 11-12(a), different thread affinity configurations have resulted in significant variations in the average power consumption of these benchmarks, ranging from 2%-15% on the 8-core intel and up to 25% on the 32-core AMD. The variation in performance (MFLOPS rates) in Figures 11-12(b) is more sporadic. On the 8-core Intel, variation of up to 120% in performance can be observed for benchmark 6 in Figure 11(b). In Figure12(b), on the 32-core Intel, a variation by factors of 6 is observed for benchmark 7 but the variation is small otherwise.

Therefore, thread affinities seem to significantly affect the power consumption of applications in a majority of cases but makes a significant difference in performance only occasionally. Therefore it should definitely be exploited for collective optimization of performance and power efficiency. There does not seem to be any direct correlation between the performance and the power efficiency of the different configurations, which indicates empirical tuning may be required.

7. SEQUENTIAL OPTIMIZATIONS

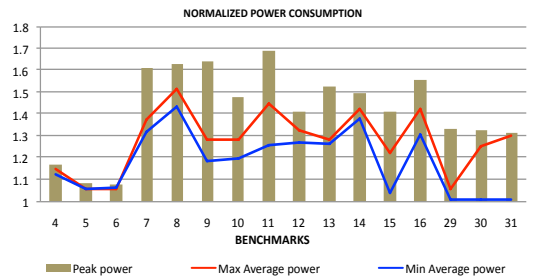
We have isolated five optimizations, cache blocking, unroll&jam, scalar replacement, loop unrolling, and SSE vectorization, which aim to enhance the efficiency of memory, register and instruction scheduling within individual CPU cores, to study their respective impact on the performance and power efficiencies of the sequential implementations of our benchmarks. The following presents our results of using the 8-core Intel machine to evaluate the sequential benchmarks compiled with the icc compiler using *-O2*, combined with POET optimization scripts when appropriate.

7.1 Cache Blocking

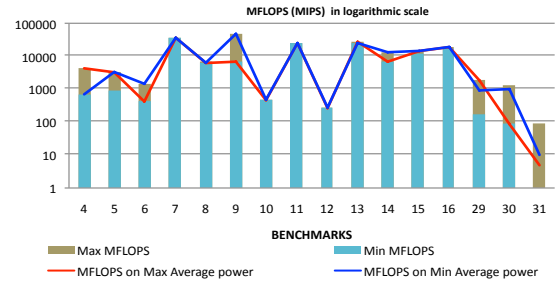
For the six matrix/stencil kernels (benchmarks 1-6) and the three LoneStar benchmarks (29-31), we used POET optimization scripts to either apply loop blocking (for the matrix and stencil kernels) or automatically revise the chunk sizes of task scheduling (for the LoneStar benchmarks) to enhance cache reuse. All benchmarks are evaluated using only a single thread. Note that although the LoneStar benchmarks use amorphous data parallelism, evaluating them using one thread with different chunk sizes for task scheduling would effectively serve the purpose of cache blocking.

Figure 13 shows the impact of applying cache blocking to the sequential implementations of these benchmarks with varying blocking factors, where *Max (or min) average power* stand for the highest (or lowest) average power consumption across different blocking factors, *MFLOPS on max (or min) average power* plot the MFLOPS rates when incurring the corresponding power consumption, *Max (or min) MFLOPS* stand for the highest (or lowest) MFLOPS rates achieved across different blocking factors, and *Average power on max (or min) MFLOPS* plot their corresponding average power consumption rates.

From the graphs, different cache blocking factors have resulted in significant variations in both performance and

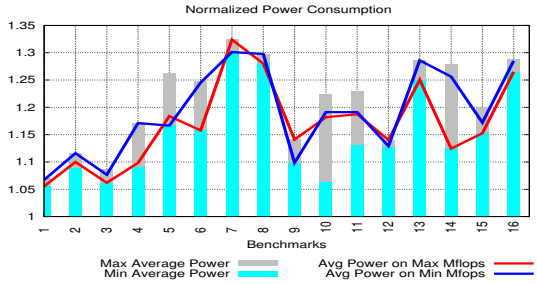


(a) Power normalized against machine idle power(360W)

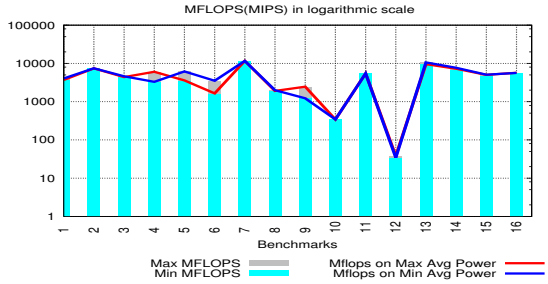


(b) *MFLOPS* of benchmarks (benchmarks 29-31 use *MIPS*)

Figure 10: Impact of scheduling policies on 32-core AMD



(a) Power normalized against machine idle power(154W)



(b) *MFLOPS* of benchmarks

Figure 11: Impact of different thread affinites on 8-core Intel

power consumptions of these benchmarks. In particular, the variations are 5-18% in power consumption and 15-60% in performance (MFLOPS rates). Further, attaining max MFLOPS has required higher power consumption than that required by attaining min MFLOPS for all benchmarks except benchmark 4 (the matrix multiplication kernel), where close to max MFLOPS performance can be achieved while incurring the min average power consumption.

Since a high degree of cache reuse can be made possible via cache blocking for the matrix multiplication kernel (benchmark 4), the high degree of cache reuse has resulted in both low power consumption (less memory traffic) and high performance. For the other benchmarks, as the degree of reuse within cache cannot fully compensate the underlying memory traffic resulted from blocking, relatively high power consumption may be required. Note that incurring max power consumption does not necessarily lead to better performance, as manifested by *MFLOPS on max average power* in Figure 13(b), as efficient use of memory traffic is required to convert the extra Watts consumed into meaningful MFLOPS.

7.2 Utilization Of Registers

We have used POET scripts to parameterize the application of unroll&jam and scalar replacement optimizations to the three matrix computation kernels (rows 4-6 in Table 1). Figure 14 shows the attained performance and power consumption when optimizing these kernels with different unroll&jam factors and selectively replacing different array references with scalar variables to promote the use of registers. In particular, *Max (or Min) average power* shows the highest (or lowest) average power consumption across all the tuning evaluations, and *MFLOPS on max (or min) average power* show the MFLOPS rates attained when incurring the corresponding average power consumption.

From Figure 14(a), the register level optimizations are able to effect 5-10% of difference in average power consumption. From Figure 14(b), the optimized code that achieves the lowest power consumption had simultaneously achieved the highest MFLOPS across all tuning evaluations.

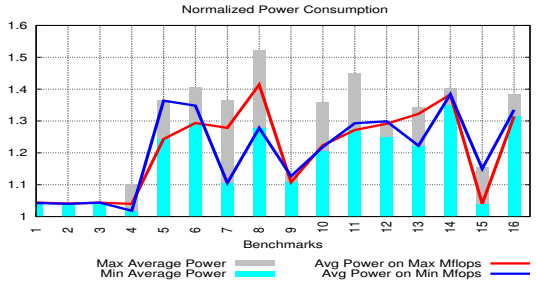
The results in Figure 14 confirm our belief that efficient utilization of registers not only improves application performance but also reduces overall power and energy consumption, since fewer memory references naturally result in both higher speed of execution and lower power consumption in the caches and memories. Therefore reducing memory usage is an effective optimization strategy to simultaneously enhance both performance and power efficiency.

7.3 Loop Unrolling And SSE Vectorization

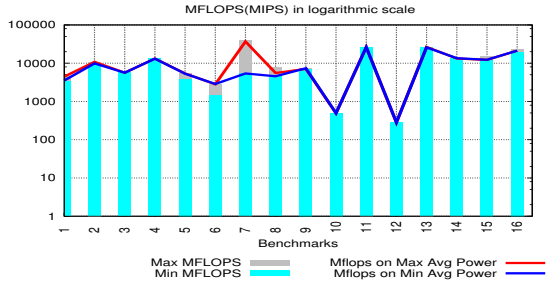
The Intel icc compiler provides two command-line options, *funroll-loops* and *msse4.1*, which instruct the compiler to apply loop unrolling and SSE vectorization respectively to optimize a given input file. To investigate the impact of these two optimizations, both of which aim to enhance the concurrency of instruction scheduling within a single CPU core, we combined them with the *-O2* flag (which includes neither of the optimizations) to compile all the serial benchmark implementations.

Figure 15 shows the resulting performance and power efficiencies of the differently optimized codes, where *Peak Power* shows the highest peak power consumption across all combinations of compilation flags, and *Max MFLOPS* shows the highest MFLOPS attained. For each combination of the *icc* compilation flags, the average power consumption and MFLOPS rates attained for each benchmark are additionally plotted. Note that the *-funroll-loops* and *-msse4.1* flags seem to frequently interfere with each other when both are specified, and we removed this combination to avoid obfuscation of the individual impact of each optimization flag.

From the graphs, applying loop unrolling in addition to *-O2* in *icc* made little difference in performance in Figure 15(b)



(a) Power normalized against machine idle power(360W)



(b) *MFLOPS* of benchmarks

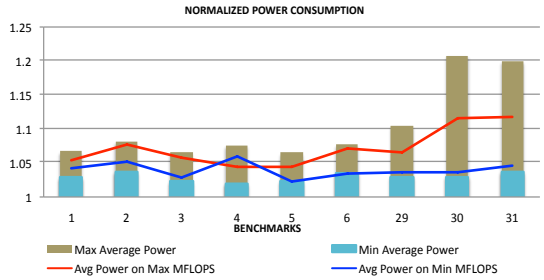
Figure 12: Impact of different thread affinites on 32-core AMD

while incurring up to 4% of variation (benchmark 4) in average power consumption in Figure 15(a) for the sequential benchmarks. For a majority of benchmarks, the average power consumption increases when loop unrolling is additionally applied, although a reduction in power is resulted in a few cases (benchmarks 5, 11, 40). We suspect that the lack of speedup by loop unrolling may have resulted in extra activities within the CPU nonetheless, which contributed to its negative impact in the power consumption.

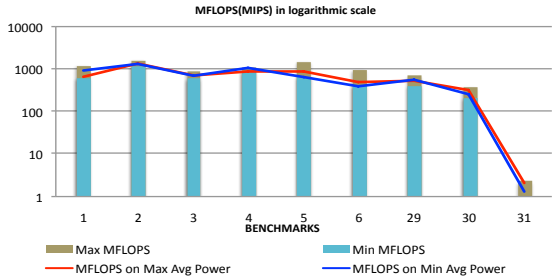
On the other hand, SSE vectorization makes slightly more significant impact in performance for several MiBench applications. Further, it increases power consumption by up to 2% for 6 benchmarks but reduces power consumption by up to 2% for 7 other benchmarks. Therefore, it seems to have a positive impact on both the performance and power efficiencies on our benchmarks. This is expected as SSE vectorization, when applied successfully, allows a single instruction stream to operate on a number of data items simultaneously, reducing both activities inside the CPU and references to the memory or cache.

8. CONCLUSIONS

This paper presents an extensive study of the impact of application level optimizations on both the performance and power efficiencies of applications from a wide variety of scientific and embedded systems domains. We observe that application-level optimizations often have a much bigger impact on performance than on power consumption. However, optimizing for performance does not necessarily lead to better power consumption, and vice versa. Compared to sequential applications, multithreaded applications give more room for both performance and power improvements. Additionally, a number of optimizations, including both loop and thread affinity optimizations, have shown great potential in supporting collective enhancement of both performance and



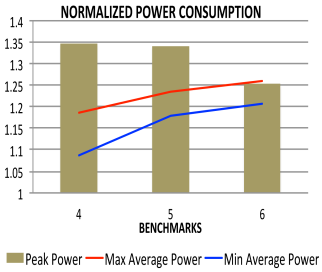
(a) Power normalized against machine idle power(154W)



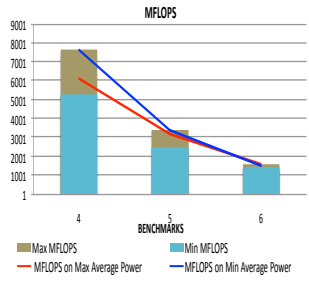
(b) *MFLOPS* of benchmarks (benchmarks 29-31 use *MIPS*)

∗ All benchmarks are optimized with default configurations using POET and compiled with icc -O2.

Figure 13: Impact of loop blocking on 8-Core Intel



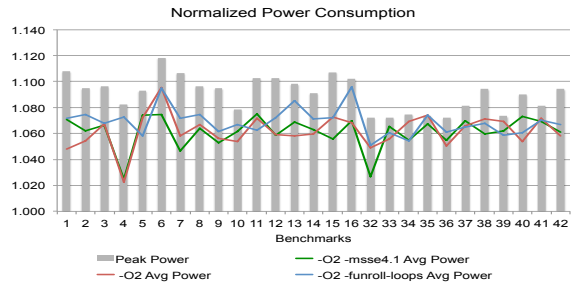
(a) Power normalized against machine idle power(154W)



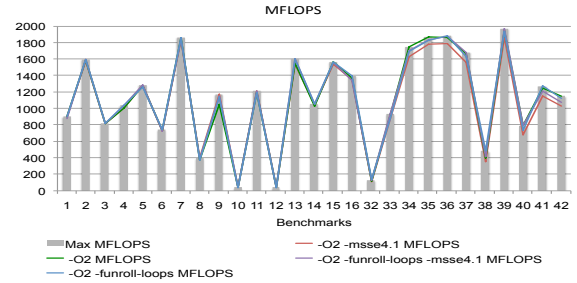
(b) MFLOPS of benchmarks

∗ For each kernel, the default configurations are used for other POET optimizations that are not being tuned, and OpenMP parallelization is disabled. The POET optimized codes are then compiled using icc with -O2 flag.

Figure 14: Result of tuning register optimizations on 8-core Intel



(a) Power normalized against machine idle power(154W)



(b) MFLOPS of benchmarks

Figure 15: Impact of CPU optimizations on 8-core Intel

power efficiency. Our experimental results provide several insights to help exploit these optimizations effectively.

9. REFERENCES

[1] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A simple power-aware scheduling for multicore systems when running real-time applications. In *IPDPS '08*.

[2] Y. Ben-Itzhak, I. Cidon, and A. Kolodny. Performance and power aware cmp thread allocation modeling. In *HIPEAC*, pages 232–246, Jan. 2010.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00*.

[5] D. J. Brown and C. Reams. Toward energy-efficient computing. *Commun. ACM*, 53:50–58, March 2010.

[6] G. Contreras and M. Martonosi. Power prediction for intel xscale®processors using performance monitoring unit events. In *ISPLED*, pages 221–226, New York, NY, USA, 2005. ACM.

[7] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 1998.

[8] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal. Analyzing the energy-time trade-off in high-performance computing applications. *TPDS '07*.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01*, 2001.

[10] H. Jin, M. Frumkin, , and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report, 1999.

[11] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler optimizations for low power systems. *Power aware computing*, pages 191–210, 2002.

[12] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09*, 2009.

[13] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai. Compiler optimization on instruction scheduling for low power. In *ISSS '00*, 2000.

[14] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and low-power scheduling techniques for embedded dsp software. In *ISSS '95*.

[15] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly Media, Inc., 1996.

[16] A. Parikh, S. K. Kim, M. Vijaykrishnan, and M. J. N. Irwin. Instruction scheduling for low power. *Journal of VLSI Signal Processing Systems For Signal Image And Video Technology*, (1):129–149, 2004.

[17] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, 2011.

[18] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *HIPEAC '11*.

[19] S. F. Rahman, Q. Yi, and A. Qasem. Understanding stencil code performance on multicore architectures. In *CF'11: ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2011.

[20] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[21] J. S. Seng and D. M. Tullsen. The effect of compiler optimizations on pentium 4 power consumption. INTERACT '03, Washington, DC, USA, 2003.

[22] W.-T. Shiue. Retargetable compilation for low power. In *CODES '01*.

[23] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. In *dasCMP*, 2008.

[24] M. Valluri and L. John. Is compiling for performance == compiling for power? INTERACT, 2001.

[25] wattsup? Power meters. https://www.wattsupmeters.com.

[26] H. Yang, G. R. Gao, and C. Leung. On achieving balanced power consumption in software pipelined loops. In *CASES '02*, pages 210–217, 2002.

[27] Q. Yi. Automated programmable control and parameterization of compiler optimizations. In *CGO*, Apr. 2011.

[28] Q. Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice & Experience*, 2011.