SAND2013-1459C

Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs

Allan K. Porterfield
RENCI
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC, USA
akp@renci.org

Stephen L. Olivier
Scalable System Software Dept.
Sandia National Laboratories
Albuquerque, NM, USA
slolivi@sandia.gov

Sridutt Bhalachandra and Jan F. Prins
Dept. of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC, USA
{sriduttb, prins}@cs.unc.edu

***Abstract*—**

Understanding on-node application power and performance characteristics is critical to the push toward exascale computing. In this paper, we present an analysis of factors that impact both performance and energy usage of OpenMP applications. Using hardware performance counters in the Intel Sandybridge X86-64 architecture, we measure energy usage and power draw for a variety of OpenMP programs: simple micro-benchmarks, a task parallel benchmark suite, and a hydrodynamics mini-app of a few thousand lines. The evaluation reveals substantial variations in energy usage depending on the algorithm, the compiler, the optimization level, the number of threads, and even the temperature of the chip. Variations of 20% were common and in the extreme were over 2X. In most cases, performance increases and energy usage decreases as more threads are used. However, for programs with sub-linear speedup, minimal energy usage often occurs at a lower thread count than peak performance.

Our findings informed the design and implementation an adaptive run time system that automatically throttles concurrency using data measured on-line from hardware performance counters. Without source code changes or user intervention, the thread scheduler accurately decides when energy can be conserved by limiting the number of active threads. For the target programs, dynamic runtime throttling consistently reduces power and overall energy usage slightly (around 3%).

I. Introduction

The trade-off between performance and energy usage has been a constraint for several generations of commodity microprocessors. Decreasing voltage and clock frequency is one common mechanism to reduce power that can result in substantial energy savings for some applications (see Section V). Increasing frequency, e.g., using Intel's Turbo Boost or AMD's TurboCore, can save energy by completing the problem faster (but typically drawing higher power). Intel's Sandybridge chip provides multiple hardware techniques to control frequency and hardware performance counters to dynamically monitor the chip's energy usage. With these tools the runtime system can actively participate in the energy/performance trade-off.

By limiting the number of active threads, the runtime can reduce the instantaneous power demand, but the effect on overall energy usage depends on how overall execution time is changed by running on fewer hardware threads. When shared hardware resources (last-level cache, memory or network bandwidth) are oversubscribed, reducing the number of threads will not significantly affect total execution time. In this case fewer active threads can result in energy savings. However, if there is little interaction between threads and hardware resources are sufficient to support their execution, decreasing the number of threads will result in longer execution time and an increase in the overall energy utilization for the application.

Our work focuses understanding energy usage and run time performance of OpenMP multicore applications. To that end, the contributions of this paper are 1.) an analysis of factors that impact both performance and energy usage and 2.) an adaptive run time system that automatically throttles concurrency based on online measurements of system performance data.

The evaluation of several OpenMP programs, including micro-benchmarks, an OpenMP task benchmark suite, and a mini-app of several thousand lines, reveals substantial variation in energy usage depending on the application, the compiler, the optimization level, the number of threads and even the chip temperature. We analyze the effects of these variables on both power and performance, observing the interaction between execution time and energy usage. On a two socket system, 10% to 20% variation in power draw between applications was common (120 - 150 Watts); in the extremes the variation was over 2X (59.0 to 158.7 Watts). Within a given application, compiler optimizations can decrease time to completion with a similar power draw for a net decrease in total energy usage, often by a factor of two, but ranging from less than 1X to more than 5X. Performance increases as parallelism increases. Generally this reduces overall energy consumed, but for several tests that exhibit imperfect parallel speedup, energy consumed was minimized before maximum parallelism was reached.

Using dynamic performance measurements, the hierarchical scheduler [1] of the Qthreads lightweight threading runtime system [2] is modified to perform active power management for OpenMP applications. Dynamic Voltage and Frequency Scaling (DVFS) requires tens of thousands of cycles to adjust voltage[3]. By modifying the duty cycle register setting, cycle frequency can be adjusted quickly and easily. Integration of a simple model into the Qthreads scheduler enables it to accurately decide when limiting the number of threads will have little to no effect on performance yet reduce the overall energy usage of the program significantly. For applicable programs, dynamic concurrency throttling by the runtime system successfully reduces power and overall energy usage up to 3%.

II. Evaluating Energy Usage

For large exascale systems, the runtime system will perform energy management on-node and probably across nodes.

To better understand the number and scope of energy variables available to the runtime system, we evaluate various OpenMP programs. OpenMP programs can implement many different parallel algorithms and can provide single node multithreading as part of larger parallel applications, e.g., using MPI.

The test programs fall into three groups. The first are locally-written micro-benchmarks to examine several different types of algorithms. These simple programs implement fundamental algorithms such as matrix multiplication and sorting. They are not tuned and represent default implementations of generic algorithms. The second group of tests applications are benchmarks from the Barcelona OpenMP Task Suite (BOTS) [4] that exercise various aspects of task parallelism in OpenMP. They include protein alignment, sparse LU decomposition, Strassen matrix multiply, simulation of a health system, integer sorting, Fibonacci number calculation, and the nqueens problem. Some of these applications are similar to our micro-benchmarks but include key optimizations. Several have cutoff thresholds limiting the amount of generated parallelism so that the granularity of the tasks is coarse enough to amortize scheduling overhead costs. Two of the benchmarks have alternate versions to compare task generation patterns. For more information, see [5]. The final test is a proxy application, or mini-app, that emulates a larger application that will run on exascale systems. LULESH is a mini-app of about 3000 lines of code that represents the behavior of a production hydrodynamics application at Lawrence Livermore National Laboratory (LLNL) [6]. It uses a Lagrangian method to solve the Sedov blast wave problemin three dimensions. We used the OpenMP version of LULESH from the LLNL web site.

All tests were performed on an M620 Dell blade with two Intel Xeon E5-2680 (Sandybridge) CPUs and 64GB of memory. The default clock speed of the processors is 2.70GHz. Intel's Turboboost feature was disabled in the BIOS. The blade runs a 3.5.0 pre-release version of the Linux kernel to support additional hardware counter access. Each test was repeated 10 times and the lowest execution time is presented in our results. Modern processors have enough internal heterogeneity that execution times often vary by several percent run to run. Different implementations of OpenMP have potentially large variations between runs particularly for task-based algorithms, but repeating experiments mitigates this variation.

A. Intel Sandybridge RAPL interface

The Intel Sandybridge architecture added the Running Average Power Limit (RAPL) interface. The RAPL interface supports client power limit control, providing the ability to manage power from the supervisor level. It provides mechanisms for proactive or reactive response to thermal events. It also tracks power usage and allows control over the maximum power draw of the chip. For this work, the MSR_PKG_ENERGY_STATUS counter was used to track energy usage by each socket. It is frequently updated but should be accessed less often to smooth jitter in the power usage. and counts in 15.3 microJoule units. Since the counter is only 32 bits wide it can wrap around in a few minutes. The measurement tools monitor the number of wraps to obtain valid application energy consumption numbers.

B. RCRdaemon

The Resource Centric Reflection (RCR) daemon runs at supervisor level and provides performance information to various clients through a self-describing hierarchical data structure in a shared memory region. It tracks hardware performance counters within a core, e.g., floating point operations, and resources shared by multiple cores, e.g., L3 cache misses, and resources shared by multiple sockets, e.g., NIC utilization. The overhead of running the daemon is about 16% of one of the 16 cores in the 2-socket test system.[1]

The RCRdaemon information is available to the programmer through a simple API that delineates a code region for measurement with a start and end call. As currently implemented the code run time must be at least 0.1 second. When the second call is reached, the elapsed time, the amount of energy used (in Joules), the average power (in Watts) and the most recent temperature of each chip (from IA32_THERM_STATUS) is output. Each test program has been modified to include the calls either explicitly in the source or implicitly through the Qthreads runtime.

C. Evaluation

We first evaluate the effects of static decisions on overall energy and performance. These decisions include choice of compiler (ICC versus GCC), algorithm design and implementation, compiler optimization level, and number of threads used. We only explore a small subset of the potential energy effects possible. The number of compilers tested could be larger and the number of compiler optimization options is almost endless. One effect we observed previously and tried to mitigate is a noticeable reduction in energy usage when the system is cold[2]. With that exception, power generally correlated with execution time. All numbers reported here are from experiments run on a warm system.

1) Compiler Variation: Two different compiler/OpenMP runtimes were tested: Intel's ICC/OpenMP and GNU's GCC/OpenMP. Table I shows that differences in energy usage between the two compilers do exist (at optimization level O2), but not consistently enough to favor one compiler over the other. For 12 of the 14 test applications, GCC used less average power but for 5 tests ICC's better execution time resulted in lower total energy cost for the program. ICC for Fibonacci (with and without cutoffs) runs faster, but GCC's advantage running BOTS with cutoff (96.5 W vs 157.0 W) resulted in GCC using less total energy (639 J vs 899 J). This advantage was not consistent, for instance, ICC used 11% less average power for the BOTS Strassen matrix multiply.

Historically, optimizations in both compilers have been focused on completion time and not concerned with energy usage. As power concerns become paramount for exascale systems, understanding the individual differences between the compiler families (e.g., how GCC uses 2% less energy on dijkstra) will become a factor in choosing the proper compilation strategy to use for a particular application.

2) Algorithm Variation: Differences in the underlying algorithms and the optimization techniques used in their implementations result in large variations in performance and power.

[1] In future work, we plan to substantially reduce the overhead by eliminating data compaction at each update. A non-compacted structure will use more shared memory but allow simple load and stores for reading and updates.

[2] Of 100 tests run on an initially cold system, the first run always used less energy and drew less power. For example, on the first run the NAS benchmark BT.C used 3.2% less energy (24666J vs 25477J) and lower power (151.0W vs 155.8W) than later runs with the same execution time.

Application	GCC			ICC		
	Time	Total Joules	Ave Watts	Time	Total Joules	Ave Watts
reduction	75.6	10201	134.9	77.1	10422	135.1
nqueens	5.5	649	118.0	6.0	714	119.0
mergesort	22.5	1364	60.6	20.5	1211	59.1
fibonacci	77.0	7115	92.3	13.5	1935	143.2
dijkstra	4.5	574	127.6	4.5	589	130.9
BOTS alignment-for	1.5	187	124.3	2.1	276	130.7
BOTS alignment-single	1.5	195	129.4	2.0	261	130.1
BOTS fib w/cutoff	6.6	639	96.5	5.7	899	157.0
BOTS health w/cutoff	1.6	216	134.5	1.5	205	135.8
BOTS nqueens w/cutoff	2.0	249	124.2	1.9	242	126.7
BOTS sort w/cutoff	1.5	188	124.9	1.4	189	134.1
BOTS sparselu-single	6.8	996	145.9	6.8	1010	147.7
BOTS strassen w/cutoff	24.1	3700	153.7	25.2	3483	138.3
LULESH mini-app	48.6	7064	145.4	14.5	2242	154.5

TABLE I. EXECUTION TIME AND ENERGY USAGE (16 THREADS): OPTIMIZATION -O2, WITH ICC -IPO FOR SPARSELU

Application	-O0			-O1			-O2			-O3		
	Time	Joules	Watts	Time	Joules	Watts	Time	Joules	Watts	Time	Joules	Watts
reduction	79.1	10578	133.7	77.1	10360	134.3	75.6	10201	134.9	76.6	10302	134.4
nqueens	14.5	1962	135.2	6.5	800	123	5.5	649	118.0	6.5	846	130.1
mergesort	77.0	4752	61.7	23.0	1390	60.4	22.5	1364	60.6	22.5	1359	60.3
fibonacci	83.1	8012	96.4	83.6	8031	96.1	141.6	13806	97.5	77.1	7115	92.3
dijkstra	8.5	1195	140.5	5.0	657	131.3	4.5	574	127.6	4.5	572	127.2
BOTS alignment-for	5.9	895	151.0	1.8	244	135.1	1.5	187	124.3	1.6	207	128.7
BOTS alignment-single	5.7	864	150.9	1.8	245	135.7	1.5	195	129.4	1.5	193	128.1
BOTS fib w/cutoff	21.2	2157	101.8	14.2	1416	100.0	6.6	639	96.5	10.1	1014	99.9
BOTS health w/cutoff	1.6	224	139	1.6	218	135.4	1.6	216	134.5	1.6	217	134.6
BOTS nqueens w/cutoff	5.6	835	148.5	2.0	252	125.3	2.0	249	124.2	1.9	238	124.6
BOTS sort w/cutoff	2.8	389	138.2	1.5	186	123.1	1.5	188	124.9	1.5	182	121.0
BOTS sparselu-single	35.6	5517	154.8	18.3	2577	141.0	6.8	996	145.9	6.8	1001	146.5
BOTS strassen w/cutoff	34.5	5509	159.6	24.3	3702	152.3	24.1	3700	153.7	24.1	3679	152.3
LULESH mini-app	79.6	12134	152.4	48.6	7078	145.7	48.6	7064	145.4	47.6	6939	145.8

TABLE II. OPTIMIZATION LEVEL EXECUTION TIME AND ENERGY USAGE (GNU GCC 16 THREADS)

For GCC (O2), the power draw varied from 60.6W for an unoptimized mergesort application to 153.7W for a modestly optimized Strassen matrix multiply. Most applications drew between 120W and 145W, a 20% variation. The range for ICC was similar (59.0W to 158.0W). A better understanding of algorithms and optimization could allow higher level schedulers to co-locate high power work with lower power work to limit thermal intervention across a system, or to schedule high power work on sockets with better cooling. General conclusions are hard to make; algorithms that overlap memory traffic with computation require more peak power than memory- or computationally-bound algorithms, but overlapping often reduces execution time and total energy costs.

3) Compiler Optimization: Compiler optimizations affect execution time, average power usage, and total energy consumption. For GCC, Table II, there is no simple relationship between increasing optimization level and energy use or average power. Increasing the optimization level generally improves performance, but O2 outperforms O3 in several cases, e.g., reduction and nqueens. Using O3 generally draws less power than O1 or O2, but in the case of nqueens O3 uses substantially more (130.1W vs. 118.0W). For energy consumption, there is no clear winner between O2 and O3.

For ICC, Table III, level O0 generally uses the most time, average power, and total energy. There are exceptions like fibonacci where O0 execution time is the same as other optimizations levels and it uses slightly less power. As with GCC, execution time differs minimally between O2 and O3 for most of our test applications. Using O3 generally draws less power (1.4W for mergesort), but minor execution variations can mask any power improvements.

Overall the decrease in energy use from optimization is substantial, typically a factor of 2 or 3 less in the optimized code. The difference in energy use between the specific optimization levels is less clear and it is difficult to pinpoint a single "best" set of optimizations to minimize energy use for all applications. Finding the optimal compiler optimizations for any given application will require autotuning. While such methods are worthwhile for some heavily used kernels and applications, for general applications a dynamic (runtime) mechanism will likely be required.

4) Thread Count: The OS and runtime system can control the number of threads available for the execution of each application. We studied the effect of varying parallelism by limiting the number of OpenMP threads to less than the available hardware limit of 16. If thread executions are completely independent, the reduction of parallelism should increase execution time proportionally. In modern processors, many hardware resources, particularly in the memory subsystem, are shared between hardware threads. These shared resources, e.g., shared caches, can improve performance by reducing memory latency, but contention for them can limit performance.

The simple micro-benchmarks have not been tuned generally and scale poorly as shown by the left-hand graphs in Figures 1 and 2. Nqueens scales to 16 threads, dijkstra scales to 8, and mergesort only scales to 2 threads. Serial versions of Fibonacci and reduction both outperform any parallel version. 16 threads of Fibonacci took 50% longer than the serial time and reduction time increased by 220%. Both are elided from the graph to preserve scale for readability. Most of the BOTS tests (left-hand graph in Figures 3 and 4) have near linear speedup with their curves overlaying each other. Three BOTS

Application	-O0			-O1			-O2			-O3		
	Time	Joules	Watts	Time	Joules	Watts	Time	Joules	Watts	Time	Joules	Watts
reduction	80.1	10892	135.9	77.1	10337	134.0	77.1	10422	135.1	77.6	10512	135.4
nqueens	15.5	2143	138.1	6.0	710	118.3	6.0	714	119.0	6.0	710	118.3
mergesort	112.1	6963	62.1	20.5	1234	60.1	20.5	1211	59.0	21.5	1239	57.6
fibonacci	13.5	1928	142.7	13.5	1933	143.0	13.5	1935	143.2	13.5	1938	143.4
dijkstra	7.5	1054	140.4	4.5	595	132.2	4.5	589	130.9	4.5	589	130.7
BOTS alignment-for	5.6	859	152.8	2.4	322	133.7	2.1	276	130.7	2.2	290	131.3
BOTS alignment-single	5.5	845	153.0	2.3	308	133.4	2.0	261	130.1	2.1	279	132.2
BOTS fib w/cutoff	10.5	1612	154.1	7.7	1162	150.3	5.7	899	157.0	5.7	894	156.2
BOTS health w/cutoff	1.6	228	141.9	1.5	205	135.8	1.5	205	135.8	1.5	204	135.0
BOTS nqueens w/cutoff	5.0	773	154.0	2.3	295	127.6	1.9	242	126.7	1.9	231	121.0
BOTS sort w/cutoff	2.0	297	147.5	1.3	175	134.0	1.4	189	134.1	1.3	176	134.3
BOTS sparselu-for	30.4	4829	158.7	6.7	999	148.4	6.8	1014	148.4	6.6	986	148.6
BOTS sparselu-single	30.2	4788	158.4	6.7	997	148.1	6.8	1010	147.7	6.6	983	148.0
BOTS strassen w/cutoff	37.2	5482	147.3	25.8	3761	145.8	25.2	3483	138.3	24.8	3498	140.0
LULESH mini-app	52.1	8132	156.2	15.5	2360	152.1	14.5	2242	154.5	14.5	2233	153.8

TABLE III. OPTIMIZATION LEVEL EXECUTION TIME AND ENERGY USAGE (INTEL ICC 16 THREADS), WITH -IPO FOR SPARSELU

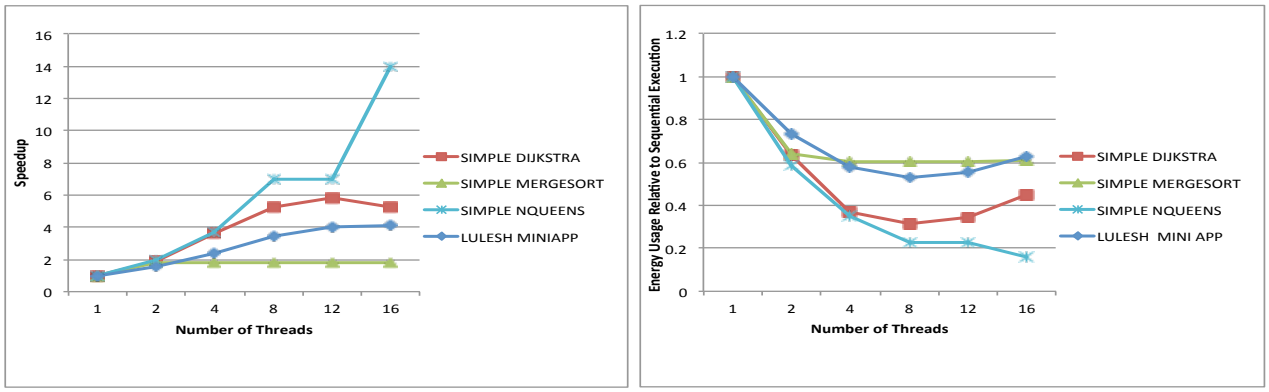

Fig. 1. SIMPLE/LULESH GCC Speedup and Normalized Energy Consumption

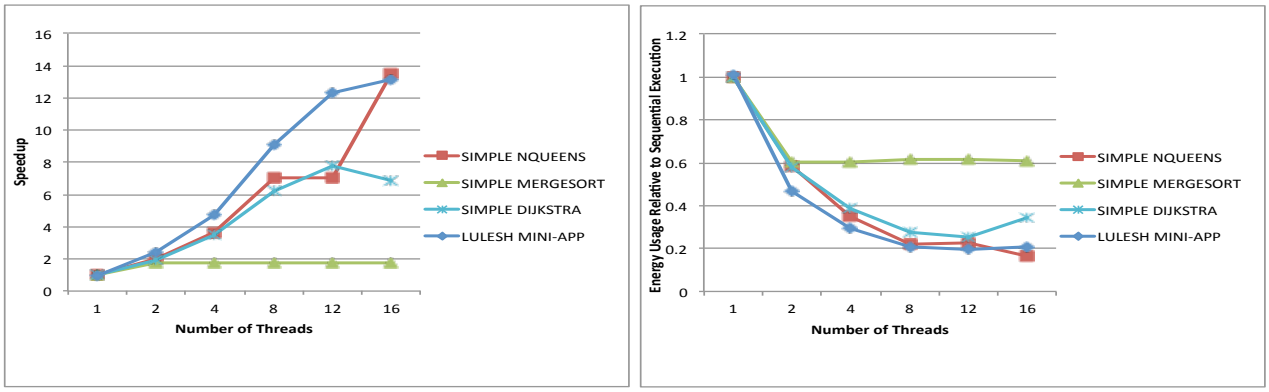

Fig. 2. SIMPLE/LULESH ICC Speedup and Normalized Energy Consumption

tests, health (6.7), sort (12.6), Strassen (4.9) and lulesh (4.0) have speedups below 15. Each of these programs experiences some form of contention limiting its performance.

The energy consumption of a processor is driven by many factors, including frequency, voltage, number of active transistors, leakage current, etc. A large portion of each processor is devoted to the hardware structures shared between cores. Thus each additional active core uses much less energy than the first core. As long as performance increases proportionally, adding cores improves overall energy consumption. Figures 1 and 2 show that for most of the example applications that was the case. The four programs that scaled poorly have unusual energy consumption curves. As thread counts increase above 8 for the programs, the additional energy usage from each addtional thread is not matched by a corresponding execution time reduction and the overall energy consumption of the program rises. The increase ranges from 17% for lulesh to 30% for dijkstra.

The observed increase in energy usage as thread parallelism increases exposes potential runtime energy savings. Simple runtime models can use hardware performance counters to recognize many forms of dynamic contention that limits scaling. Run time systems have the potential to change the number of active hardware threads over time using hardware mechanisms to idle one or more threads and greatly reduce the power consumed. A system that dynamically adjusts parallelism to

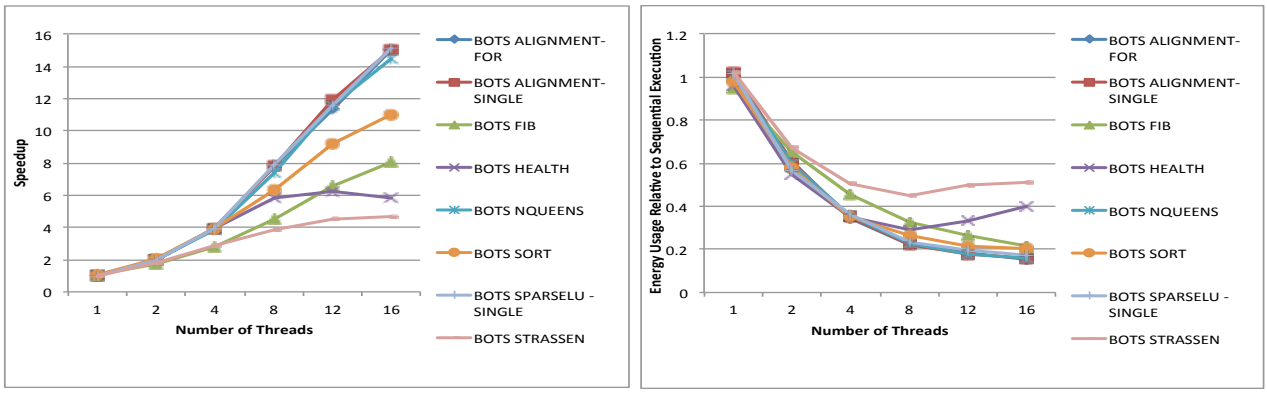

Fig. 3. BOTS GCC Speedup and Normalized Energy Consumption

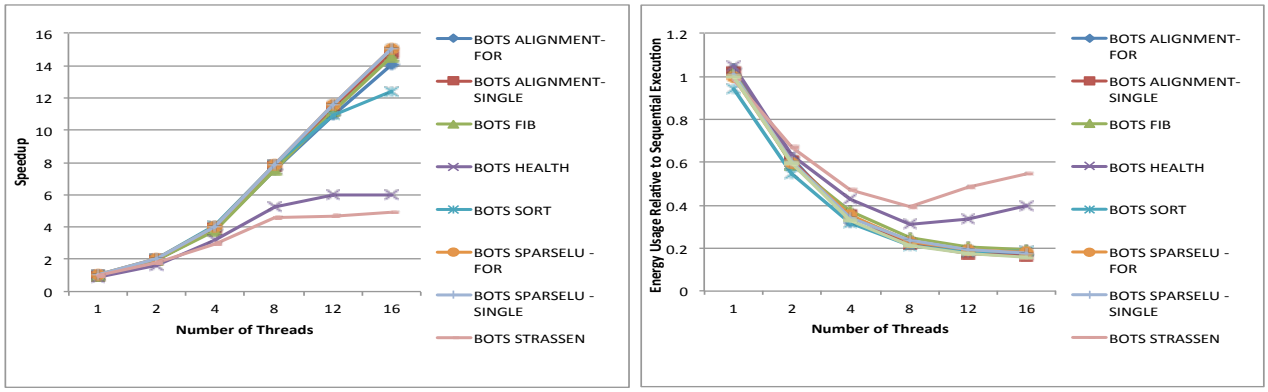

Fig. 4. BOTS ICC Speedup and Normalized Energy Consumption

control energy consumption can have the ability to limit energy consumption with little impact on execution time.

III. Using Qthreads to Execute OpenMP

Qthreads [2] is a cross-platform general-purpose parallel run time library designed to support lightweight threading and synchronization in a flexible integrated locality framework. Qthreads directly supports programming with lightweight threads and a variety of synchronization methods, including non-blocking atomic operations and potentially blocking full/empty bit (FEB) operations. The Qthreads lightweight threading concept and its implementation are intended to match future hardware environments by providing efficient software support for massive multithreading.

In the Qthreads execution model, lightweight threads (*qthreads*) are created in user-space with a small context and small fixed-size stack. Unlike heavyweight threads such as pthreads, qthreads do not support expensive features like per-thread identifiers, per-thread signal vectors, or preemptive multitasking. Qthreads are scheduled onto a small set of worker pthreads. Logically, a qthread is the smallest schedulable unit of work, such as a set of loop iterations or an OpenMP task, and a program execution generates many more qthreads than it has worker pthreads. Each worker pthread is pinned to a processor core and assigned to a locality domain, termed a *shepherd*. There may be multiple worker pthreads per shepherd, and shepherds may be mapped to different architectural components, e.g., one shepherd per core, one shepherd per shared L3 cache, or one shepherd per processor socket.

OpenMP is supported by Qthreads through the ROSE source-to-source compiler and its XOMP interface [7]. Although Qthreads XOMP/OpenMP support is incomplete, it accepts every OpenMP program accepted by the ROSE compiler. OpenMP directives are outlined and mapped to functions and data structures in the Qthreads library. Explicit tasks and chunks of loop iterations are implemented as qthreads

A. MAESTRO Extensions to Qthreads

The MAESTRO project [8] has implemented alternative thread scheduling mechanisms and polices within the Qthreads runtime. The Sherwood hierarchical scheduler [1] recognizes that on non-uniform memory access (NUMA) machines some threads share a last-level cache and a local memory. Those threads can take advantage of that locality by sharing a LIFO work queue. Constructive cache sharing avoids high-latency accesses and saves memory bandwidth. Work stealing among the queues provides system-wide load balancing.

The Sherwood scheduler has been extended to allow scheduling decisions to be made based on current contention for memory bandwidth. It was integrated with the Resource Centric Reflection system [9] to modify scheduling policies to account for dynamic utilization of various shared hardware resources across a multi-socket multi-core node. The driving idea is that thread performance will be impacted not only by the code running on that thread, but also by the code running on other threads sharing hardware resources. Based upon on-line measurements of system resource usage, the run time system changes the number of worker threads active at

any thread initiation point (the beginning of a parallel loop iteration chunk or task instantiation). Internal mechanisms are implemented in Qthreads to allow the number of active threads to vary dynamically to support this ability.

IV. Automatic Dynamic Concurrency Throttling

In Section II, we showed that for some programs, total energy consumption is reduced by running with less than the maximum number of available threads. Qthreads/MAESTRO has the ability to manipulate thread scheduling policies based on hardware counter information. Combined with the RAPL interface in the Intel Sandybridge architecture, the infrastructure exists to build a scheduler that automates dynamic thread throttling. It will limit the amount of active parallelism in regions of code where power usage is high and contention for a shared resource limits execution performance.

Automatic throttling for Qthreads is implemented using two daemons: the system RCRdaemon, described in Section II-B, and, inside the Qthreads runtime, a user-level daemon that reads the shared memory region updated by RCRdaemon. The latter daemon activates every 0.1 seconds and uses very little CPU time. This granularity was chosen to allow fluctuations in the energy counters to dissipate. (DVFS could support this modulation frequency but could only slow all cores or none, whereas our duty cycle changes are per-core.) The polling frequency is adjustable to allow control of overhead versus responsiveness. It measures two metrics: current power utilization and memory bandwidth. The observed values are classified as High, Medium, or Low. When both conditions are High, a flag is set to activate throttling at the next opportunity. If both conditions are Low, throttling is disabled. The Medium range does not toggle throttling, but avoids hysteresis effects that occur when observed values hover near the threshold.

Inside the runtime, the scheduler was modified to check if throttling is active. Each shepherd maintains an counter of active worker threads. When a worker thread looks for work (either a new task or parallel loop iterations), if the active thread count for this shepherd is greater than the shepherd-local throttling limit, then that worker thread is placed in a spin loop. It waits for one of four conditions: throttling deactivation; application completion; parallel region termination; or parallel loop termination. In the spin loop, the thread does no productive work and is run in a low power mode.

Most previous work in this area, described in Section V, has used DVFS to run in a low power state. For this application, DVFS has two significant disadvantages. First, as currently implemented, it affects all cores on a processor. It also requires significant OS and hardware overhead to adjust the voltage without having instructions fail. Dynamic throttling will require individual cores to be rapidly placed in a low power state, so we use a different hardware mechanism. The duty cycle of individual cores can be modified using a control register (MSR)[3]. On the Sandybridge architecture, the effective frequency of the clock can be reduced to 1/32nd of the actual frequency. By slowing the processor in a tight memory spin loop, each thread saves about 3W. For our tests idling four threads saved over 12W (in one case 134W vs. 147W). Adjusting the duty-cycle is not instantaneous, but our measurements show it takes only the amount of time equivalent to approximately 250 memory operations (including call and OS overhead to access the MSR). Thus, duty-cycle modification provides low-overhead hardware-supported power control.

[3]Both DVFS and duty cycle modification require kernel permission level at the hardware, therefore dynamic power adjustment must be run as root.

Configuration	Time	Total Joules	Ave Watts
16 Threads - Dynamic	48.4	6860	141.7
16 Threads - Fixed	45.5	7089	155.9
12 Threads - Fixed	48.2	6341	131.5

TABLE IV. Execution Time and Energy Usage on LULESH using MAESTRO (optimization -O3).

Configuration	Time	Total Joules	Ave Watts
16 Threads - Dynamic	16.04	2262	140.9
16 Threads - Fixed	16.34	2306	141.0
12 Threads - Fixed	15.83	2236	141.2

TABLE V. Execution Time and Energy Usage on Dijkstra using MAESTRO (optimization -O3).

A. Throttling Models

The RCRdaemon records the average power drawn across intervals for each socket in the system. In our test two-socket system, we used the results from the application benchmark evaluation to set our high and low power benchmarks for each socket. Since only a few applications exceeded 150W for their entire execution, we chose 75W per socket as our metric for high energy usage. The power draw of almost all applications exceeded 100W during execution, so 50W per socket was chosen as our low power point. The low value was picked after looking at the 12 thread results, attempting to allow the high power applications to run with 12 threads, but to reset when average power regions were encountered.

When only average power is used to determine throttling, it often limits thread count for programs running at high efficiency and increased overall energy consumption. To reduce this behavior, we check the number of outstanding memory references in the memory subsystem. In previous studies [10], each processor was found to have an effective maximum outstanding memory references count, above which memory bandwidth does not increase but memory latency worsens. The high value is chosen to be 75% of the maximum achievable number and the low is 25% of that number.

B. Throttling Results

Four test programs showed power utilization curves for which throttling the amount of concurrency could result in a total reduction in energy consumed. We examine each to evaluate the effectiveness of our simple throttling implementation. On the other applications, which already scale well, our throttling implementation never detected the need to throttle and resulted in only minor overheads (up to 0.6%).

1) LULESH Mini-App: Table IV shows that concurrency throttling is partially effective. Limiting the parallelism reduces average power of lulesh by 14.2W at the cost of increased execution time (2.9 sec). This resulted in a total energy savings of 229J (or about 3.3%). The execution time matched the 12 thread case, but turning the threads off at the OS level saved an additional 10.2W and 519J.

2) Simple Microbenchmark Dijkstra: Table V shows that the energy decrease from 16 to 12 threads of the simple micro-

Configuration	Time	Total Joules	Ave Watts
16 Threads - Dynamic	1.33	173.0	130.0
16 Threads - Fixed	1.26	176.3	139.4
12 Threads - Fixed	1.35	166.9	123.0

TABLE VI. EXECUTION TIME AND ENERGY USAGE ON BOTS HEALTH USING MAESTRO (OPTIMIZATION -O3).

Configuration	Time	Total Joules	Ave Watts
16 Threads - Dynamic	23.7	3601	151.7
16 Threads - Fixed	24.1	3716	154.2
12 Threads - Fixed	26.9	3505	130.3

TABLE VII. EXECUTION TIME AND ENERGY USAGE ON BOTS STRASSEN USING MAESTRO (OPTIMIZATION -O3).

benchmark dijkstra was entirely the result of a quicker execution with fewer threads. The average power was effectively unchanged. The reduced thread count eased contention for some shared resources and allowed a faster execution. When throttling is enabled, the power remains about the same and about half of the lost performance is recovered resulting in a 1.9% power improvement (3.1% possible).

3) BOTS Benchmark Health: The BOTS benchmark health, Table VI, decreases execution time by 6%(0.9sec) but increases power by 12% (16.4W) as the thread count increases from 12 to 16 threads. The automated throttling recovers about half of the increased power (9.4W). This results in an execution 5% slower than 16 fixed threads using 7% less power and a small decrease in total energy (173J vs 176.3J).

4) BOTS Benchmark Strassen: The fastest execution for BOTS Strassen occurred with throttling enabled (see Table VII). The 1.6% execution time improvement combined with the 1.6% reduction in power resulted in the throttled version using 3.2% less energy. Most of the execution was done with 16 threads, resulting in energy and power usage comparable to 16 threads rather than the 18% lower power required with only 12 threads.

C. Overall Effectiveness

Throttling is only useful when the CPU is using a high percentage of the available power and execution is delayed due to contention a some point in the system. When programs effectively use all the threads, reducing the number of threads significantly increases total execution time, raising overall power utilization. When latencies are limiting performance, parallelism needs to stay high, even if memory contention occurs. If the applications in our study are representative of typical multicore applications, between a quarter and a third of programs (or program phases) may see energy savings from throttling. Reducing parallelism dynamically is effective without any modifications to the application. Duty-cycle modification by the runtime saves over half the energy that could be saved by having the OS put the hardware thread to sleep. The savings are mitigated by the longer execution times resulting in a net gain of only about 3%. With better hardware idle mechanisms the improvements could be larger.

V. RELATED WORK

Over the last decade or so there has been considerable research into power management, initially for embedded devices and then later for HPC systems and applications. The embedded community has responded to the power challenge through improvements to make the system as well as the applications power-aware [11], [12]. Embedded devices typically have stricter power constraints but less restrictive performance requirements compared to the HPC systems addressed here.

Power management on HPC systems has focused on using the available hardware mechanisms for controlling energy use. The most common mechanism has been voltage and frequency scaling, used in either inter-node [13], [14] or intra-node methods. Our technique is comparable to the intra-node efforts. Early intra-node work by Ge, *et al.* [15] explored opportunities to save energy at fixed frequencies for memory-bound applications. Freeh, *et al.* [16] used offline traces to manually divide the work into phases that are run at several frequencies do determine the most energy efficient choice. The Tiwari, *et al.* Green Queue [17] automates the process of finding phases and optimal frequencies using power models. A number of efforts use hardware performance counters [18], [19], [20] to compute optimal off-line settings. Several projects estimate energy usage based on hardware counters with direct correlation including cache access [21], MIPS [22] and CPU stall cycles [23]. Li, *et al.* [24] address DVFS and dynamic concurrency throttling for hybrid, multi-node MPI+OpenMP applications. They statically determine the best concurrency level for each OpenMP phase and explore dynamic algorithms to reduce power utilization in nodes that are idle due to load imbalance. All these approaches estimate energy usage, whereas MAESTRO measures energy using the new hardware-supplied energy counter. In contrast to our duty-cycle modification, DVFS-based work has the dual drawbacks of 1) large transition time overheads and 2) global effect on all cores on a chip (currently). Duty-cycle modification can be activated and deactivated very quickly and on a single core basis.

Recently an additional hardware mechanism, power clamping, has been introduced on Intel SandyBridge, along with similar mechanisms on IBM Power 6 and 7 (capping) and AMD Bulldozer (capping and thermal design power limits). Rountree, *et al.* [25] examine the effect of clamping for an HPC application (NAS MG). Their work addresses processor performance variation as HPC moves from performance scheduling to *power scheduling*. Concurrency throttling to match parallelism to available power would operate well within a multi-node power clamping environment. However, our current work seeks to reduce energy usage, not to respect a fixed power bound. Other work that saves energy by turning off components includes Dynamic Sleep Signal Generator by Youssef *et al.* [26], which uses off-line traces to predict when functional units can be put to sleep.

VI. CONCLUSIONS

Software optimization of energy usage is just beginning. In its early stages, users and developers often rely on brute force energy optimization strategies that resemble strategies used in early autotuning. Modern processors make the entire process opaque by aggressively idling temporarily unused hardware. We examined two compilers, along with a variety of algorithm and optimization levels, and discovered that there are no simple answers for energy usage for all different applications.

The general rule of thumb "hurry up and finish" works well for about 2/3 of the applications studied. The base power draw is high enough that increasing power to complete

execution more quickly is normally advantageous to the total energy consumed. In four programs whose concurrency scaled poorly, we found a degree of concurrency beyond which the effectiveness of additional parallelism no longer justifies the additional energy used. For these programs the minimum energy did not correspond with the number of threads to obtain maximum performance. As core counts increase on processors and hardware resources shared between cores become bottlenecks to execution performance, limiting parallelism to control energy costs will become more attractive.

The Qthreads runtime was modified to automate the detection of 'excess' parallelism, using the Intel Sandybridge hardware performance counters throughout execution. By reducing parallelism when power and memory bandwidth usage are both high, the energy consumption for all four programs with poor scaling was reduced. The savings are a fraction of the potential savings, due to current overheads in the software and the hardware mechanisms for idling the unused threads.

Concurrency throttling, as presented, is a mechanism for saving energy within a single node of a larger system. The interface to control active parallelism and monitoring of energy consumption made available by the runtime system will be useful to higher level tools that seek to control energy usage across multi-node systems. In the future, we envision energy tools that control not only parallelism but also processor speed and dynamic hardware resource allocation in response to system bottlenecks or load imbalance.

ACKNOWLEDGMENT

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *Intl. Journal of High Performance Computing Applications*, vol. 26, no. 2, May 2012.

[2] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads." in *IPDPS '08: Proc. of the 22nd IEEE Intl. Parallel and Distributed Processing Symposium*. IEEE, 2008.

[3] H. Kimura, M. Sato, Y. Hotta, T. Boku, and D. Takahashi, "Emprical study on reducing energy of parallel programs using slack reclamation by DVFS in a power-scalable high performance cluster," in *CLUSTER 2006: Proc. of the 2006 IEEE Intl. Conference on Cluster Computing*. IEEE, 2006.

[4] A. Duran and X. Teruel, "Barcelona OpenMP Tasks Suite," http://nanos.ac.upc.edu/projects/bots, 2010.

[5] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *ICPP '09: Proc. of the 38th Intl. Conference on Parallel Processing*. IEEE, Sept. 2009.

[6] Lawrence Livermore National Laboratory, "Hydrodynamics challenge problem," Technical Report LLNL-TR-490254, 2010, https://computation.llnl.gov/casc/ShockHydro/LULESH-files/spec.pdf.

[7] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries," in *IWOMP 2010: Proc. of the 6th Intl. Workshop on OpenMP*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds., vol. 6132. Springer, 2010.

[8] A. Porterfield, P. Horst, S. Olivier, R. Fowler, D. O'Brien, K. Wheeler, and B. Viviano, "Scheduling OpenMP for Qthreads with MAESTRO," RENCI, Technical Report TR-11-02, 2011, http://www.renci.org/wp-content/uploads/2011/10/TR-11-02.pdf.

[9] A. Porterfield, R. Fowler, and M. Y. Lim, "RCRTool design document; version 0.1," Tech. Rep. RENCI Technical Report TR-10-01, 2010.

[10] A. Mandel, R. Fowler, and A. Porterfield, "Modeling memory concurrency for multi-socket multi-core systems," in *ISPASS 2010: IEEE Intl. Symposium on Performance Analysis of Systems and Software*, White Plains, New York USA, March 2010.

[11] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *SOSP '99: Proc. of the 17th ACM Symposium on Operating Systems Principles*. ACM, 1999.

[12] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *SOSP '01: Proc. of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[13] V. W. Freeh, N. Kappiah, D. K. Lowenthal, and T. K. Bletsch, "Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs," *Journal of Parallel and Distributed Computing*, vol. 68, no. 9, 2008.

[14] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. K. Bletsch, "Adagio: Making DVS practical for complex HPC applications," in *ICS '09: Proc. of the 23rd Intl. Conference on Supercomputing*, 2009.

[15] R. Ge, X. Feng, and K. W. Cameron, "Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters," in *SC05: Proc. of the 2005 ACM/IEEE Conference on High Performance Networking and Computing*. IEEE Computer Society, 2005.

[16] V. W. Freeh and D. K. Lowenthal, "Using multiple energy gears in MPI programs on a power-scalable cluster," in *PPoPP 2005: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[17] A. Tiwari, M. Laurenzano, J. Peraza, L. Carrington, and A. Snavely, "Green queue: Customized large-scale clock frequency scaling," in *CGC '12: Proc. of the 2nd Intl. Conference on Cloud and Green Computing*, Nov. 2012.

[18] D. C. Snowdon, E. L. Sueur, S. M. Petters, and G. Heiser, "Koala: a platform for OS-level power management," in *Proc. of the 2009 EuroSys Conference*, 2009.

[19] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proc. of the 2004 Intl. Symposium on Low Power Electronics and Design*, 2004.

[20] C. W. Lively, X. Wu, V. E. Taylor, S. Moore, H.-C. Chang, C.-Y. Su, and K. W. Cameron, "Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems," *Computer Science - R&D*, vol. 27, no. 4, 2012.

[21] R. Ge, X. Feng, W. Feng, and K. Cameron, "CPU miser: A performance-directed, run-time system for power-aware clusters," in *ICPP 2007: 36th Intl. Conference on Parallel Processing*. IEEE, 2007.

[22] C. Hsu and W. Feng, "A power-aware run-time system for high-performance computing," in *SC05: Proc. of the 2005 ACM/IEEE Conference on High Performance Networking and Computing*. IEEE Computer Society, 2005.

[23] S. Huang and W. Feng, "Energy-efficient cluster computing via accurate workload characterization," in *CCGrid 2009: Proc. of the 9th IEEE/ACM Intl. Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2009.

[24] D. Li, B. R. de Supinski, M. Schulz, K. W. Cameron, and D. S. Nikolopoulos, "Hybrid MPI/OpenMP power-aware computing," in *IPDPS 2010: Proc of the 24th IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[25] B. Rountree, D. H. Ahn, B. de Supinski, D. K. Lowenthal, and M. Schulz, "Beyond DVFS: A first look at performance under a hardware-enforced power bound," in *HP-PAC 2012: Proc. of the 8th Workshop on High Performance, Power-Aware Computing*, May 2012.

[26] A. Youssef, M. Anis, and M. I. Elmasry, "Dynamic standby prediction for leakage tolerant microprocessor functional units," in *MICRO 39: Proc. of the 39th IEEE/ACM Intl. Symposium on Microarchitecture*, 2006.