Energy Aware Algorithmic Engineering

Swapnoneel Roy
School of Computing
University of North Florida
Jacksonville, Florida 32224
Email: s.roy@unf.edu

Atri Rudra
Department of CSE
University at Buffalo, SUNY
Buffalo, New York 14260
Email: atri@buffalo.edu

Akshat Verma
IBM Research – India
New Delhi, India
Email: akshat.verma@in.ibm.com

***Abstract*—In this work, we argue that energy management should be a guiding principle for design and implementation of algorithms. Traditional complexity models for algorithms are simple and do not aid in design of energy-efficient algorithms. In this work, we conducted a large number of experiments to understand energy consumption for algorithms. We study the energy consumption for popular vector operations, matrix operations, sorting, and graph algorithms. We observed that the energy consumption for any given algorithm depends on the memory parallelism the algorithm can exhibit for a given data layout in the RAM with variations up to 100% for many popular algorithms. Our experiments validate the asymptotic energy complexity model presented in a companion paper [1] and brings out many practical insights. We show that reads can be more expensive in terms of energy than writes, and different data types can lead to different energy consumption. Our most important result is a theoretical and experimental quantification of the impact of parallel data sequences on energy consumption. We also observe that high memory parallelism can also increase energy consumption with multiple concurrent access sequences. We use insights from our experiments to propose algorithmic engineering techniques for practical energy efficient software.**

I. Introduction

Energy has emerged as a first class computing system resource that needs to be carefully managed in server class systems as well as personal devices. Energy management has traditionally focused on hardware and operating systems or hypervisors. Energy-aware hardware design focuses on designing hardware to run existing benchmark with much less energy consumption. This directly leads to reduced energy consumption for existing workloads without any change in the software stack. Research in this area has also led to providing various knobs that allow hardware to modify its performance profile (and by extension power consumption) to adapt to changes in the performance requirement of the workload. System software like hypervisors and operating systems use these knobs to save energy. The key idea employed is to measure the current workload intensity (e.g, CPU idle percentage) and use it to trigger hardware mechanisms (e.g., lower the operating frequency of the cores).

In this work, we argue that energy management should be a guiding principle for application coding as well as design of algorithms. Energy-aware hardware and system software can reduce the energy consumption for legacy application software. However, energy-aware algorithmic engineering provides a complementary technique to reduce energy consumption beyond what hardware and system software can achieve. Traditional complexity models for algorithms assume a fairly simplistic model for the hardware and are designed for minimizing compute or memory cost. Understanding the energy consumed by modern hardware requires a more careful look at individual compute elements as well as layout of memory across parallel banks. In [1], we presented a theoretical energy complexity model and algorithms optimal in the energy model. However, engineering algorithms for energy efficiency on real-life systems requires more careful investigation, which is the focus of this work.

The goals of this work are multi-fold. Firstly, designing energy efficient algorithm requires additional work and we study under what circumstances engineering algorithms to be energy efficient is worth the effort. Secondly, the energy complexity model in [1] requires individual algorithms to be modified to support memory parallelism. We investigate if there are practical generic ways to achieve this effect. Finally, complexity model like the one in [1] ignores constants and we investigate if practical algorithmic engineering needs to consider aspects not covered in the theoretical model.

Our Contributions: Our work makes four key contributions. First, we present a detailed experimental evaluation of energy consumption for a large class of algorithms. Our work experimentally validates the key concepts of the energy model proposed in [1]. We also present a generic way to achieve any desired degree of memory parallelism for a given access pattern, which can be employed by software engineers to implement energy efficient algorithms. We observe that the increased parallelism reduces running time along with energy. Second, we identify many new characteristics of algorithms that impact energy consumption equally, if not more than the key aspects identified by the energy complexity model in [1]. We show both experimentally and theoretically that multiple data sequences can increase energy consumption and anti-parallel sequences are preferred over parallel sequences. Third, we identify algorithmic patterns for which energy-efficient design leads to significant energy savings and algorithmic patterns for which energy awareness does not materially impact energy consumption. We show that it is more critical to parallelize reads than writes. Similarly, optimizing algorithms with multiple parallel sequences does not lead to significant energy savings. We observe that there is a tradeoff between parallelizing access across all banks by a data sequence versus partitioning disjoint memory banks for different independent data sequences. Our final contribution is a set of practical recommendations, derived from our experimental study, for energy-aware algorithmic engineering.

The rest of the paper is structured in the following manner.

Section II abstracts the key requirement for energy-efficiency described in [1] and presents a generic technique to achieve this requirement in practice. We describe our experimental setup in Section III and present our key experimental results in Section IV. A careful theoretical and experimental analysis of our key observation on parallel and anti-parallel sequences is presented in Section V. We discuss practical algorithmic engineering approaches for energy-efficiency in Section VI and related work in Section VII.

II. Engineering Algorithms for Energy Efficiency

A. Energy Complexity Model

An asymptotic energy complexity model for algorithms was proposed in [1]. Inspired by the popular DDR3 architecture, the model assumes that the memory is divided into P *banks* each of which can store multiple blocks of size B. In particular, P blocks in P *different* memory banks can be accessed in parallel. The main contribution of the model in [1] was to highlight the effect of parallelizability of the memory accesses in energy consumption. In particular, the energy consumption of an algorithm was derived as the weighted sum $T + (PB) \cdot I$, where T is the total time taken and I is the number of *parallel* I/Os made by the algorithm. However, this model measures the energy consumption asymptotically and in particular, ignores constants in order to derive the simple model above.

B. P-way Parallelism for a Vector

Energy optimal algorithms proposed in [1] require data to be laid out in memory with a controlled degree of parallelism. We first propose a generic way to ensure desired memory parallelism for a given input access pattern or vector.

Given any input data (vector), we created a logical mapping which ensures access to the vector in P-way parallel fashion, where P ranges from 1 to 8. The memory controller on our hardware allocates memory in strides across banks. We ensure P-way parallelism by first creating blocks of contiguous memory of size B in one bank. We then use a page table, which ensures that blocks are allocated from banks in the specified P-way. For the first part, we implement the following mapping function. Consider an input vector $\mathbf{V}$ size of N. Our mapping function logically converts $\mathbf{V}$ into a matrix $\mathbf{M}$ of dimensions $\frac{N}{B} \times B$, where B is the size of each block. Each block in $\mathbf{M}$ actually consists of $\frac{B}{s}$ (logical) strides each of size s. We logically assign strides to $P \times B$ blocks at a time in our function. Also we make sure, for any block, all its strides lie in the same bank. We store the mappings in matrix $\mathbf{M}$, where $\mathbf{M}_{ij}$ denotes the j^{th} stride of the i^{th} block.

For the second part, we define a *page table* vector $\mathbf{T}$ of size $\frac{N}{B}$ that contains the *ordering* of the blocks. The ordering is based on the way we want to access the blocks (P-way would mean a full parallel access). The page table is populated by picking blocks with jumps. For a 1-way access, we select jumps of P ensuring the consecutive blocks are in the same bank. For a P-way access, we selects jumps of 1 i.e. the blocks are picked from banks in round robin order. Fig. 1 presents an example with 4-way and 2-way parallel access.

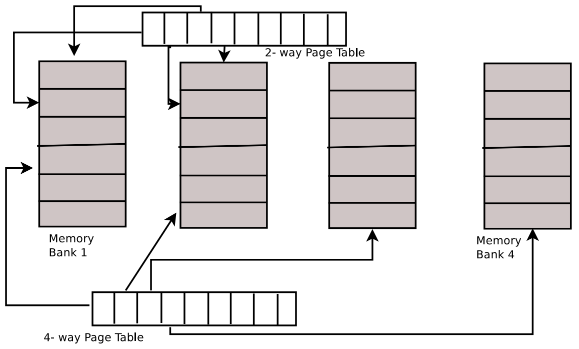


Figure 1. Memory layout for achieving various degree of parallelism for $P = 4$.

```
Input: Page table vector $\mathbf{V}$, jump amount $jump$.
$factor = 0$;
for $i = 0$ to $\frac{N}{B} - 1$ do
    if $i > 1$ and $((i \times jump) \mod (\frac{N}{B})) = 0$ then
        $factor = factor + 1$;
    end
    $\mathbf{V}_i = (i \times jump + factor) \mod (\frac{N}{B})$;
end
```

Algorithm 1: The function to create an ordering among the blocks

1) Code Optimization: Our memory layout strategy requires additional data structures as well as mapping of logical data blocks to physical data blocks. (Let us call this mapping `Map`.) This introduces a significant constant overhead, which may offset any variance in power induced by the memory layouts. We implemented few optimizations to minimize the overhead of this remapping.

1) **Reduce the number of calls to `Map` function:** The key overhead in our experiments is that we need to invoke the `Map` function, once per memory access. To minimize this overhead, we made this function inline. More importantly, we used the fact that the `Map` function has spatial locality (consecutive elements in a logical stride are placed on consecutive memory locations). Hence, we call the `Map` function only once per s elements, significantly minimizing the overhead.
2) **Usage of bit shifts:** Since most of our input were in powers of 2, we used bit shifts in place of operations like multiplication, division, and $\mod$.
3) **Usage of register variables:** The variables that were required to be computed a large number of times were replaced by register variables. This brought down the running time in many cases.

The benchmark code was written in C and was compiled using `gcc` compiler.

C. Engineering an Algorithm

Given an algorithm $\mathcal{A}$, we use the memory layout scheme to ensure desired memory parallelism. We consider the input to the algorithm and identify the most common access sequence. We ensure the required level of parallelism for the vector formed by the desired access sequence using the technique described earlier. For algorithms that iterate over the input with different access sequences in each iteration, we try to find sub-sequences of size PB or a multiple of PB that are accessed in the same order for each iteration. If such sub-sequences can be identified, we can create a large vector by adding the sub-sequences in any arbitrary order. If such sub-sequences can not be found, the vector is created by selecting the most likely sequence of the input data. For algorithms that use multiple input vectors, the dominant vector is used for parallelizing the memory layout.

III. Experimental Setup and Methodology

We conducted a large number of experiments to understand the implications of various implementation choices for an algorithm in its overall energy consumption.

A. Benchmarks Studied

We considered a mix of benchmarks, which together form some of the most frequent computations performed in work-loads. In order to understand the implications of various data types, we conduct these experiments with both integer and double data (except for the graph benchmarks where we only use integers). Our experiments are also designed to validate the model proposed in [1]. Hence, we implement variants of algorithm that allow us to use 1, 2, 4, or 8 memory banks in parallel on DDR3 memory with 8 memory banks. Below we list the benchmarks used in this study.

1) **Write**: Given a vector A, we write a random value in every element of A. The vector is parallelized using the method described in Sec. II-B.
2) **Copy**: Given two vectors A and B, we copy all elements from A to B. Both vectors A and B are parallelized.
3) **saxpy**: SAXPY (Single-precision real αX Plus Y) is a Level 1 (vector) operation in the Basic Linear Algebra Subprograms (BLAS) package, and is a common operation in computations with vector processors. SAXPY is a combination of scalar multiplication and vector addition. Both X and Y are parallelized.
4) **Sorting**: Given a vector A, sort all elements of A in increasing order. We evaluate selection sort, quick sort, and merge sort for this study. The input array is parallelized for these experiments.
5) **Matrix Transpose**: Given a matrix M, transform the matrix to M' s.t. $M'_{i,j} = M_{i,j}$ for all elements $i, j < n$. The matrix in row-major form is parallelized.
6) **Graph Traversal**: Given a graph G in adjacency list format, traverse through all connected components of G. The adjacency list is parallelized. Internal data structures (e.g., array to keep track of visited nodes) are not parallelized.
7) **Shortest Path**: Given an edge-weighted graph G in adjacency list format and two nodes s and t, compute the shortest path from s to t in G. The adjacency list is parallelized. The auxiliary data structures are not parallelized.

Each experiment has been repeated 10 times and the means are reported in this work. The variations in the energy readings that is, deviations from the average values were small (less than 8%) for all the experiments performed. We present the variations in the energy readings for the read experiment (of Fig. 3(a)) in Fig. 2.

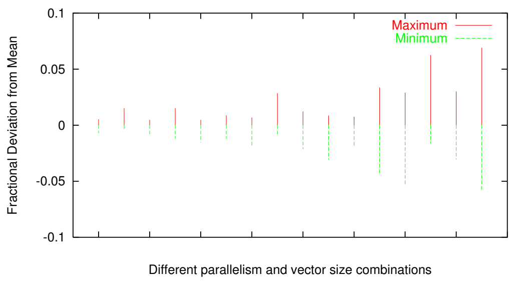


Figure 2. Variation for Read Experiment

B. Hardware Setup

The experiments were run on a Mac note book with 4 Intel cores, each of which operates at a frequency of 2 GHz. The laptop has 4GB DDR3 memory with 8 banks and is running Mac OS X Lion 10.7.3. The sizes of L2 Cache (per Core), and L3 Cache are respectively 256 KB and 6 MB. The disk size of the machine is 499.25 GB. During any experimental run, all non-essential processes were aborted to ensure that the power measured by us can be attributed primarily to the application being tested. We measured the (total) power drawn using the Hardware Monitor tool [2] which include the processor power, memory (RAM) power, and a background (leakage) power. The Hardware Monitor tool reads sensor data directly from Apple's System Management Controller (SMC), which is an auxiliary processor independent of the main computer. Hence, energy consumed by the monitoring infrastructure does not add to the sensor readings, ensuring that the reported measurements are fairly accurate.

IV. Experimental Results

A. Vector Operations

In our first set of experiments, we compare three key operations - reading an integer vector, writing an integer vector, and copying one integer vector to another. We observe (Fig. 3(a), (b) and (c)) that memory parallelism has significant impact on energy consumption. This is in line with energy model proposed in [1]. However, if we only look at write operations (Fig. 3(b)), the impact of memory parallelism is muted. Copy (and read) operations benefit by a factor of 2 with parallelism, whereas writes benefit by less than 10%. The result may seem surprising at first but is a direct consequence of the memory hierarchy. DRAM memory is often write back allowing the memory controller to re-sequence writes to introduce parallelism in memory access. Hence, even if the original access sequence uses only 1 bank at a time, memory controller sequences the buffered writes in a way that allows all P banks to be used in parallel. On the other hand, reads can not be delayed by the memory controller and unoptimized read sequences lead to high energy consumption.

It is also important to note that the *copy* benchmark uses 2 different arrays. Hence, both arrays can not individually

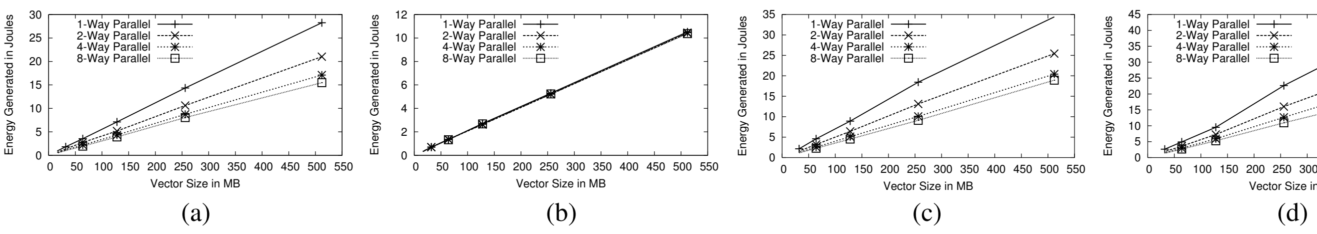


Figure 3. Energy consumed by (a) Read (b) Write (c) Copy and (d) Saxpy

achieve isolated 8-way parallel performance (the two arrays may access the same bank). However, we still see best performance with 8-way parallel access. This can be again explained by the fact that write operations can be deferred. Hence, the read operations are able to achieve full parallelism. The same observation is reinforced with the saxpy benchmark in Fig. 3(d), where 8-way parallelism leads to least energy consumption.

B. Matrix Transpose

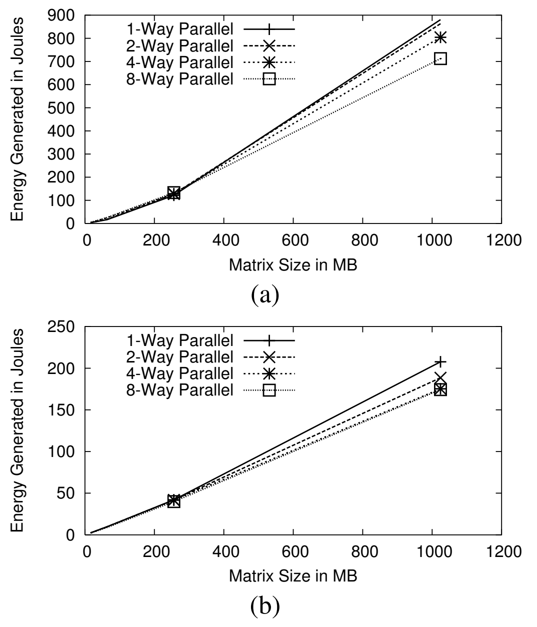


Figure 4. Energy consumed by (a) Naive and (b) Blocked Matrix Transpose

We next study the matrix transpose problem. We use two algorithms for transposing a matrix $\mathbf{A}$. The first algorithm is naive transpose, where we exchange $\mathbf{A}_{ij}$ and $\mathbf{A}_{ji}$ for input matrix $\mathbf{A}$. This algorithm essentially consists of two data access sequences for the matrix - the first one is a row major scan and the second one is a column major scan of the matrix. Since the matrix is P-way parallel only in the row major form, only one of the two scans is P-way parallelized. In the second algorithm, we divide the matrix $\mathbf{A}$ (of size $N \times N$) into sub-matrices of size of size $B \times B$. We transpose the sub-matrices in place, and swap each sub-matrix with its *swap partner* to get $\mathbf{A}^T$. We call this algorithm as blocked transpose.

Fig. 4 captures the energy consumed respectively by naive transpose and by blocked transpose. We observe that blocked transpose consumes less energy than naive transpose. For both algorithms, we observe a moderate impact of using optimal data access pattern in the algorithm. Overall, our experiments validate the energy model in [1], while highlighting the fact that if an algorithm has multiple data access sequences, then the impact of parallelization is moderated.

C. Sorting

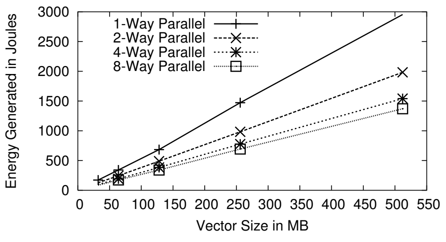
We next consider the sorting problem. Sorting is a data-dependent problem and the exact data access pattern depends on the underlying data. Hence, it is not possible to statically layout the data with a fixed degree of parallelism. We use the observation that sorting algorithms works in iteration, where each iteration consists of one or more sequential data sequences that are accessed in a data-dependent manner. Hence, we parallelize the input array with the required degree of parallelism. We consider selection sort, quick sort, and merge sort for this study.

Selection sort algorithm with an input size of n performs $n - i$ reads (comparisons) in its i^{th} round. After i rounds, we have the first i elements sorted. Executing i rounds of selection sort (on an input of n integers) therefore is expected to consume almost equal amount of energy as done by i times reading the input (of n integers). This is assuming i to be a constant with $i \ll n$.

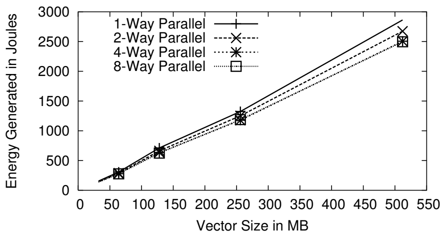
We performed $i = 100$ rounds of selection sorting for inputs ranging from sizes 32MB ($n = 8388608$) through 512MB ($n = 67108864$). This according to our conjecture is energy wise equivalent to performing 100 read operations on the input arrays each time. The results of Fig. 3(a) and Fig. 5(a) pretty much validate this. We compare the value of a particular point, e.g. 1-way parallel for input size 256MB of Figs. 3(a) with 5(a). Fig. 3(a), the read shows an energy generation of about $15J$, while Fig. 5 shows about $1500J$ for the same point. This is pretty much along the expected lines.

We observe that energy consumed by energy optimal layout (8-way parallel) of selection sort performs much better compared to non-optimal (e.g., 1-way parallel) as shown in Fig. 5(a). Quick sort (Figure 5(b)) and Merge Sort (Figure 5(c)) show moderate savings over changes in parallelization due to energy-awareness.

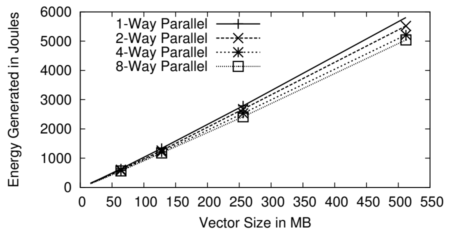
In order to understand this better, we take a close look at the three sorting algorithms. Again, the selection sort algorithm at each iteration i, finds the i^{th} minimum element of the input and puts it in its *proper* place in the array. Hence at step i, the algorithm executes a linear sequence of length $n - i$ where n is the input size. Since we have a single linear sequence executing during an iteration, the impact of memory parallelism expresses itself in significant savings in energy. In quick sort,



(a)

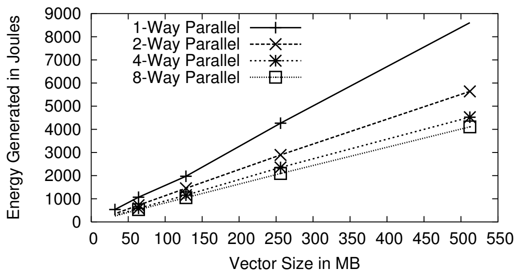


(b)

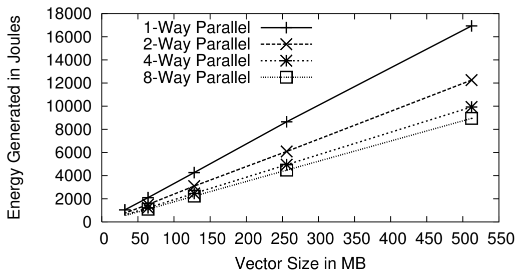


(c)

Figure 5. Energy consumed by integer (a) selection sort (b) quicksort and (c) mergesort with random input.



(a)



(b)

Figure 6. Energy consumed by quicksort on integers with (a) sorted and (b) reverse sorted input.

```
Input: Vector $\mathbf{A}$ containing integers, end positions $low$
       and $high$, pivotal element $pivot = \mathbf{A}[high]$.
Output: Vector $\mathbf{A}$ partitioned w.r.t $pivot$.
$pivot = \mathbf{A}[high]$;
$i = (low - 1)$;
$j = 0$;
for $j = low\ to\ high - 1$ do
    if $\mathbf{A}[j] \leq pivot$ then
        $i = i + 1$;
        if $i \neq j$ then
            $swap(\mathbf{A}[i], \mathbf{A}[j])$;
        end
    end
end
$swap(\mathbf{A}[i+1], \mathbf{A}[high])$;
```
Algorithm 2: Partition

we partition the input based on a pivotal element chosen for each (sub)array we sort. In our implementation of quick sort, we always choose the last element of the (sub)array as the pivotal element. Both quick sort and merge sort either partition or merge smaller sequences in each iteration (other than the last iteration). Further, merge sort merges two sequences in parallel, which are completely interspersed with each other. So, the impact of parallelism is extremely subdued. Algorithm 2 shows our partition algorithm. We have 2 linear sequences (i and j) executing on the array at any point of time. The point to note is that the two parallel sequences execute in the same direction. However, while sequence j covers all the elements one by one, sequence i might have jumps. Hence, the interference between the two sequences is smaller then merge. Hence, quick sort is able to extract some benefit of higher parallelism.

We also experimented quick sort with sorted and reverse sorted input. For these two scenarios (Fig. 6((a) & (b)), the optimal energy-aware version of quick sort shows significant benefit. For the sorted version, the linear sequences i and j merge into a single sequence (since their locations at any point of time is the same). So effectively we have a single linear sequence in case of the sorted input. For the reverse input, index i does not move at all. Again hence effectively we have a single linear sequence in case of the reverse input.

Our experiments show that the number of parallel sequences used by an algorithm plays a significant role in energy consumption. Multiple parallel sequences can interfere with each other and prevent parallel access to memory. Even though this interference only leads to a constant factor impact on energy, the increase leads to perceivable performance problems in practice. We do a careful experimental and theoretical analysis of this phenomenon in Sec. V.

D. Graph Algorithms

We next consider graph algorithms, which have recently emerged as a popular framework for solving social media and big data problems [3]. Many graph algorithms perform some kind of traversal, which can exhibit very weak locality with an adjacency list data structure. Hence, it is extremely difficult to create static data sequences, that can be parallelized across the banks. We use the following trick to introduce some degree of parallelism. We store the graphs in a non-standard adjacency list format. We store the array of linked lists in one "flattened" array of size $2m$ called *edge list* that just lists all the edges (ordered by one end-point). This ensures parallelizability of the operation to enumerate all neighbors of a given vertex, which is possibly the most common operation for the algorithms we consider. We also store two additional arrays of size n each. One array lists the degrees of each vertex and the second captures the index in the *edge list* the first edge incident on the vertex starts. When changing the parallelizability of the

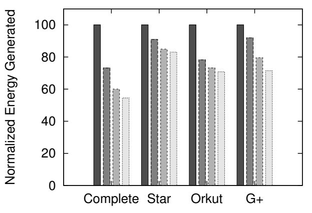
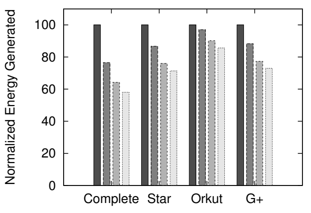
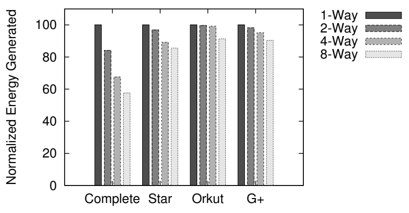

(a)

(b)

(c)

Figure 7. Energy consumed by (a) BFS (b) DFS and (c) Dijkstra's on various graphs

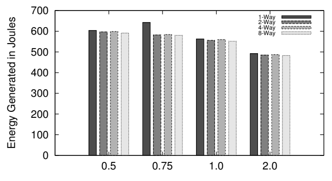
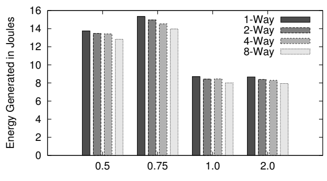
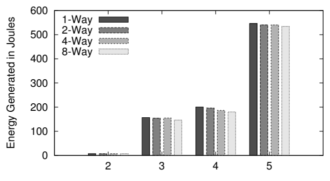
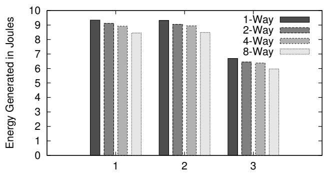

(a)

(b)

(c)

(d)

Figure 8. Energy consumed by Dijkstra's algorithm with different covariance in edge weights for (a) Orkut graph and (b) GPlus graph. Average energy consumed by Dijkstra's algorithm with different distances between (s, t) pairs for (c) Orkut graph and (d) GPlus graph

Type	Nodes	Edges	Dia	Average degree of top 10% nodes
Complete	8192	33550336	1	8191
Star	16777216	16777215	2	11
Orkut	3072441	117185083	9	318
Gplus	107614	13673453	6	1648

Table I. CHARACTERISTICS OF GRAPHS USED

algorithms, we only changed the layout of the edge list array (from the usual 1-way to 8-way). For weighted graphs we also store the edge weights in the edge list. We study the energy consumed by traversal algorithms (Breadth-first-search and Depth-first-search) and Dijkstra's shortest path algorithm.

We experimented on the graphs in Table I. Two of those graphs are from the SNAP dataset collection [4]: social circles from Google plus (henceforth Gplus) and Orkut online social network (henceforth Orkut). All these graphs are unweighted. For the shortest path problem, we need edges weights, which we generate from the Gaussian distribution. We vary the ratio of standard deviation to the mean (which is called covariance) to obtain different edge weight distributions. We took the absolute value of the weights (no negative weights). The results are presented in Figs. 7 and 8.

We ran the three algorithms above (DFS, BFS and Dijkstra's) on the four graphs (complete, star, Gplus and Orkut). The results are presented in Fig. 7. The gains in going from 1-way to 8-way for the complete graph is in line with our previous experiments, though the gain are lower by about 20% because the graph algorithms have a random jump after each operation to get all neighbors. Similar behavior though with diminishing gains are seen for the star and Gplus graph (both of which are much sparser than the complete graph and thus the energy cost of accessing the auxiliary arrays become more significant).

The trends in energy savings is the same for Orkut graph expect for Dijkstra's algorithm. This is because the number of nodes n is of the same order as the cache size. The auxiliary array sizes of the DFS and BFS algorithm however are just small enough to still fit into the cache. However, for Dijkstra's the fact that the auxiliary array sizes are three times larger makes it spill beyond the cache size, which results in the access to the auxiliary data being more significant than accesses to the edge list. In particular, we now have to access more than one data structure in the main memory and even if each of these accesses are fully parallelized, they will interfere with each other. Further, the relatively smaller average node degree of dense nodes implies that parallelism does not aid the *neighbor list* operation significantly. We also experimented with different distance values and different covariance values for generating the edge weight distribution. The results (Fig. 8) exhibit the same overall trend over the entire range of experiments.

Our graph experiments lead to the following recommendation. If a graph has high average degree and all auxiliary data structures can fit in cache, then parallelizing the layout leads to energy savings. However, if either of these conditions are violated, memory parallelism does not lead to energy savings.

E. Impact of data type

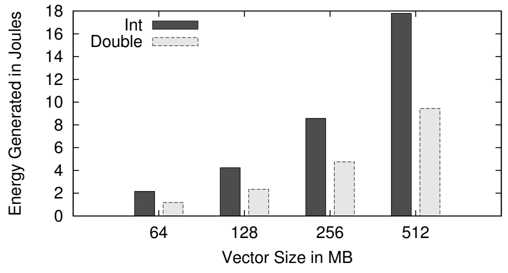
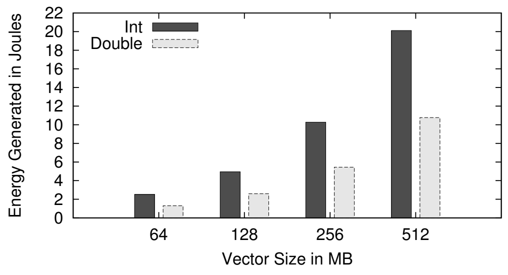
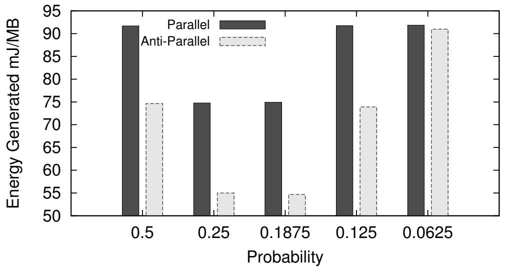

(a)

(b)

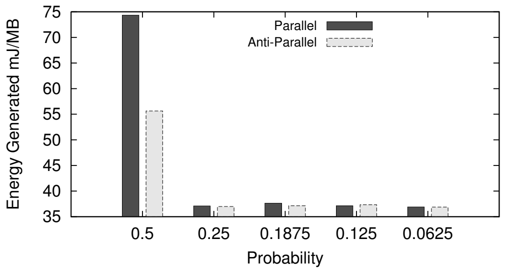
Figure 9. Energy consumed by (a) Copy (b) Saxpy

Next, we study the impact of various data types by performing all our experiments so far with doubles. We observe that using a larger data type leads to significantly lower energy for copy and saxpy operations. Using optimal algorithms for the data types, we find up to 40% lower energy consumption for both these algorithms (see Fig. 9). The data presented in Fig. 9 compares the energy consumed in the experiments (copy and saxpy) for the 8-way parallel case using `int` and `double` data types. Similar observations hold for quick sort as well as selection sort (omitted for lack of space). This savings is comparable to the typical savings we have observed by using optimal variants of an algorithm against non-optimal versions. Hence, engineering the data type can lead to significant differences in energy consumption.

V. Analysis of Results



(a)



(b)

Figure 10. Energy Consumption (mJ/MB) with 2 Sequences on (a) 1GB input (b) 512MB input

Our experimental study establishes high variation in energy consumed for algorithms, which are equivalent in classical complexity models. Overall, these results establish the need for an energy model based on work and parallel I/O complexity, as proposed in [1]. However, energy consumption differs between algorithms that are asymptotically equal even in the energy model. One of the key aspects highlighted by our experiments is the reduced impact of parallelism, when multiple data sequences are present. We define two simple operations that occurred frequently in our experiments and analyze their performance both experimentally and theoretically.

Definition 1. *Circular Sequence: Given P banks, create a circle of banks from 1 to P in clock wise order. If an algorithm accesses data in a way that follows the circle clock-wise or anti clock-wise, the algorithm is defined as a* Circular Sequence *algorithm.*

We define two common variants of Circular Sequence next.

Definition 2. *Parallel Circular Sequence: An algorithm is said to be a parallel circular sequence (or just parallel sequence) algorithm if the data access pattern of the algorithm can be represented as a set of circular sequences, all of them either clock-wise or anti clockwise.*

Definition 3. *Anti-Parallel Circular Sequence: An algorithm is said to be an anti-parallel circular sequence (or just anti-parallel sequence) algorithm if the data access pattern of the algorithm can be represented as a set of circular sequences, with half of them clock-wise and the other half anti clockwise.*

We conducted micro experiments with 2 parallel and anti-parallel sequences. Since we observed in our experiments that different data sequences may not progress at the same rate, we defined a probability q to capture the likelihood that the first sequence would progress at any given time. The second sequence progresses with probability $1 - q$. We experimentally study the energy consumed by parallel and anti-parallel sequences in Fig. 10. We observe that at $q = 1/2$, the parallel sequence consumes significantly higher energy than anti-parallel or reverse sequences. Further, as one of the sequence starts to dominate the other, the energy consumption decreases and the difference between parallel and anti-parallel sequences reduce. At $q = 0.0625$, the difference between the two sequences is insignificant. Hence, we can conclude that if there is a choice between parallel and anti-parallel sequences that are accessed at similar rates, using the anti-parallel sequence is more energy efficient.

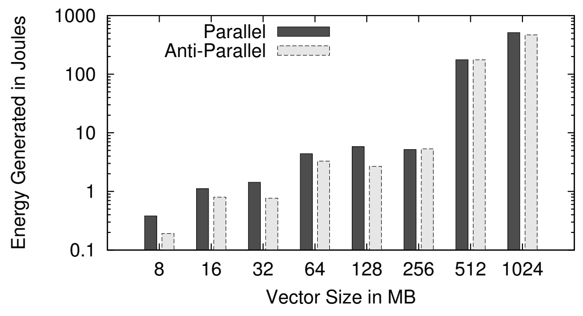


Figure 11. Energy consumption in joules with partition on two sequences (on log scale)

We validate the above finding using the partition algorithm. We implement a variant of partition algorithm, where we create two anti-parallel sequences. The first sequence, called *forward* sequence scans from the start of the array till it the finds an element larger than the pivot. The *reverse* sequence then proceeds from the end of the array and stops when it finds an element smaller than the pivot. The two elements are swapped and the *forward* sequence continues its scan. Fig. 11 plots the energy consumed by the two variants of partition algorithm. We observe that anti-parallel partition has significantly lower energy consumption than parallel partition validating our conjecture. We next attempt to theoretically understand the reason behind different energy consumption for parallel and anti-parallel sequences.

Definition 4. *Circular Iteration: A circular iteration for a circular sequence algorithm captures data accesses that equal P block I/Os.*

Given a sequence of N I/Os performed by an algorithm, it is easy to see that the algorithm has N/P disjoint circular iterations and needs to perform at least N/P parallel I/Os.

Definition 5. *Circular Loss (L): Given any circular iteration, the number of blocks that can not be read in one parallel I/O is defined as the circular loss for that iteration.*

It is easy to see that if we allow circular iterations to be defined at the start of each parallel I/O, then the total circular loss across all the iterations is the extra number of I/Os performed by an algorithm. Further, the algorithm needs to perform at least L/P additional parallel I/Os over and above the lower bound (of N/P parallel I/Os). We next consider anti-parallel circular sequences with 2 sequences, one clock-wise and one anti clock-wise. This e.g. models the behavior of the partition code, where we go "inwards" from both end of the array.

We prove the following results.

Theorem 1. *The total loss in an anti-parallel sequence with* 2 *sequences is bounded by* P *or* 1 *parallel I/O only.*

Proof: We first note that the two sequences cross each other only once in exactly P steps. Further, in these P time steps, the two sequences access blocks in different banks. In other words, all the P access between two consecutive crossings are to distinct banks and thus, we have only one parallel I/O. Thus, the only places where we might not have a parallel access with P I/Os is at the beginning if the two sequences do not start at the same bank, which leads to the claimed result. ■

Next, we will see that the situation can be much worse for the parallel circular sequence case. Hence, partition has better constants than merging, which has parallel sequences.

A. The parallel case

We now consider the case of parallel sequence with two sequences. Unlike the clean result of Theorem 1, the situation for the parallel case is much different. In particular, it is easy to come up with a sequence that has no extra parallel I/Os and at the same time one can show parallel sequence such that all the parallel I/Os have exactly one I/O (i.e. the effective parallelizability is 1 instead of the idea P).

Thus, we need to consider classes of parallel sequence. In particular, we consider the following natural parallel sequence:

Definition 6. *Let* $0 < q \leq 1/2$ *be a real. At every step, with probability q the first sequence advances and with probability* $1-q$ *the second sequence advances.*[1] *Let us call this the q-random parallel sequence.*

We will prove the following result:

Theorem 2. *Let* $0 < q < 1/2$ *such that* $\log P/q$ *and* $\sqrt{P \log P}/(1-2q)$ *are both* $o(P)$. *Then for large enough* P *and for any* $\alpha < 1-q$, *the number of extra parallel I/Os needed by the q-random parallel sequence is at least* $\left(\frac{1-\frac{\alpha}{1-q}}{1+\left\lceil\frac{1-\alpha}{1-2q}\right\rceil} - o(1)\right)\cdot\frac{N}{P}$, *with probability at least* $1-1/P^{O(1)}$.

[1]Note that by changing the roles of the first and second sequences, we can also deal with the case of $1/2 \leq q < 1$.

By picking $q=\epsilon$ and say $\alpha = 3\epsilon$, we get the following:

Corollary 1. *For small enough constant* ϵ, *the* ϵ*-random parallel sequence has parallelizability at most* $P\cdot\left(\frac{1}{2}+O(\epsilon)\right)$.

In the rest of the section, we will prove Theorem 2.

To simplify the proof somewhat we will change the setup a bit: instead of the sequences running in a circular fashion we will assume that they run on a line with N points– we will use i and j to denote the location of the first and second sequence respectively. To convert this to the circular framework we can just replace i and j by $(i \mod P+1)$ and $(j \mod P+1)$ respectively. Thus, the state of the system at any point of time can be represented by (i,j). Note that with probability q, the state changes to $(i+1,j)$ and with probability $1-q$, the state changes to $(i,j+1)$.

We start with some simple observations, which follow by noting that the probability that i does not increment for ℓ consecutive steps is $(1-q)^{\ell}$ and by the (additive) Chernoff bound respectively.

Lemma 1. *Given a state* (i,j), *with probability at least* $1-1/P^{O(1)}$, i *will get incremented in* $O\left(\frac{\log P}{q}\right)$ *steps.*

Lemma 2. *Let the current state be* (i,j). *Then in* ℓ *steps with probability at least* $1-1/P^{O(1)}$, *the values of i and j change by* $\ell\cdot q \pm O(\sqrt{P\log P})$ *and* $\ell(1-q)\pm O(\sqrt{P \log P})$ *respectively.*

Call a state (i,j) to be *good* if $j \mod P$ is behind $i \mod P$ by at most αP (i.e. if we think of both i and j as going clockwise on the circular banks, then if we keep i stationary then it will take j at most αP steps to get to i). Otherwise call the state *bad*. We now argue the following.

Lemma 3. *Let* (i,j) *be good. Then with probability at least* $1-1/P^{O(1)}$, *within* $\frac{\alpha P}{1-q}+O\left(\frac{\log P}{q}+\sqrt{P\log P}\right)$ *steps both* i *and* j *would have taken the same value* $\mod P$ *(though not necessarily at the same time).*

Proof: By Lemma 1, within $O\left(\frac{\log P}{q}\right)$, the index j would have gone ahead by one to $j' = j+1$. Then by Lemma 2 in at most $\frac{\alpha P}{1-P}+O\left(\sqrt{P\log P}\right)$, i steps, i would attain a value i' such that $i' \mod P = j' \mod P$. Summing up the two bounds gives the claimed result. ■

Next, we argue that it does not take long for a bad state to get converted into a good state.

Lemma 4. *Let* (i,j) *be a bad state. Then with probability at least* $1-1/P^{O(1)}$, *the state will turn good in at most* $\frac{(1-\alpha)P}{1-2q}+O\left(\frac{\sqrt{P\log P}}{1-2q}\right)$ *steps.*

Proof: Note that if (i,j) is bad, then by definition if $i-j$ increases by $(1-\alpha)P$, then the resulting state will be good. Lemma 2 implies that with probability $1-1/P^{O(1)}$, in ℓ steps $i-j$ increases by at least $\ell(1-2q)-O(\sqrt{P\log P})$. Picking ℓ as claimed completes the proof. ■

Remark 1. *It turns out that in subsequent calculations, we can ignore* $o(P)$ *terms and thus, from now on for both Lemma 3 and 4, we will ignore the* $o(P)$ *terms.*

Define $x = \left\lceil \frac{1-\alpha}{1-2q} \right\rceil$. Now divide the N items into contiguous chunks of size $(x+1)P$. Call a chunk *good* if either (1) (i,j) is good and Lemma 3 holds or (2) (i,j) is bad and first Lemma 4 holds and then Lemma 3 holds. Note that by the definition of q-random parallel sequence, Lemmas 3 and 4 and the union bound, every chunk is good independently with probability at least $1 - 1/P^{O(1)}$. Note that by the Chernoff bound this implies that with the probability $1 - 1/P^{O(1)}$, all but $o(N/(xP))$ chunks are good.

Now consider a good chunk. In this case the loss in the chunk is $P - \frac{\alpha P}{1-q}$. This is because in the worst case we have a good state (i,j) within xP steps and then by Lemma 3, we will have a parallel I/O of size $\frac{\alpha P}{1-q}$. Thus, so even if all the first xP steps are fully parallel, we would still have the loss as claimed above. Thus, the total loss overall is $\left(P - \frac{\alpha P}{1-q}\right) \cdot \left(\frac{N}{(x+1)P} - o(N/P)\right)$. Thus even if the total loss were fully parallelizable, we would have at least as many extra parallel I/Os as claimed in Theorem 2, as desired.

Remark 2. *Theorem 2 needs $q < 1/2$ and the result also deteriorates as q approaches $1/2$. However, we believe that this is a shortcoming of our proofs and not due to some inherent reason. In fact, we conjecture that for* every *value of q, the maximum parallelizability that one can get out of parallel sequences is roughly $P/2$.*

VI. Discussion and Conclusion

In this work, we experimentally validated the energy complexity model proposed in [1]. We identified scenarios where the energy model is valid as well as aspects of energy consumption that are not captured by the model. We now use insights obtained from our experiments and theoretical analysis to propose practical algorithmic engineering ideas for implementation of energy-aware algorithms.

Read Versus Write: We have observed that write buffers can help memory controllers use all memory banks in parallel. Hence, write intensive workloads do not need energy-awareness as much. We recommend using simple algorithms with small constants for write-intensive workloads and leave memory controllers to achieve high parallelism.

Data Sequences: We observed that algorithms that have a single data sequence active at any given time can achieve optimal energy using memory parallelism. On the other hand, algorithms with parallel sequences can lead to high energy consumption and memory parallelism does not help at all. Finally, anti-parallel sequences can use memory parallelism better and reduce energy consumption. Hence, we propose that given a choice an algorithm that uses a single data sequence should be preferred. If an algorithm requires two or more data sequences, preference should be given to anti-parallel sequences (for example, anti-parallel partition code is preferred over parallel version, quick sort is preferred over merge sort). Further, if an algorithm has multiple parallel sequences, then effort should be made to introduce bias so that one sequence dominates others. If we implement an algorithm with multiple balanced parallel sequences, then there is no need to ensure memory parallelism as its impact is not significant.

Data Dependent Algorithms: For data dependent algorithms, we should identify a primary sequence that is deterministic and parallelize it (e.g., input array in selection sort). If a primary sequence can not be identified, we can identify long sub-sequences and try to ensure that they can be parallelized (e.g., in graphs we optimize the sub-sequence used for neighborhood operation). If such sub-sequences also can not be identified, there is no advantage of parallelizing memory layout.

Auxiliary data structures: We observed that auxiliary data structures introduce noise and reduce the benefit of memory parallelism. Hence, we should engineer algorithms so that additional data structures can fit in cache and do not interfere with memory parallelism. If possible, one can also consider using a disjoint set of banks for auxiliary data structures and ensure parallelism across remaining banks for the primary data structures (e.g., 2 banks can be reserved for auxiliary data structures and 6 banks can be reserved for primary input).

Choice of Data Type: We observed that double data type consumes less energy for a fixed number of bytes than integer data type. This is a direct consequence of using larger data units. Hence, wherever possible, data should be processed using larger data types.

VII. Related Work

Research in power modeling can be broadly classified into (i) Simulator-based [5], [6], (ii) CPU Utilization-based [7] (iii) Event or performance counters based [8] and (iv) Coarse-grained [9]. Early power management research used analytic power models based on voltage and frequency [9], which are fast, but only provide rough estimates. Coarse-grained estimates based on the type and state (active, off) of the processor have been used in [10]. However, with the increase in the dynamic power range of servers [11], a more accurate power prediction method is needed. Power models based on readily available system parameters like CPU utilization [7] are possibly the simplest to use for algorithm design. A CPU utilization based model is currently the most popular power estimation model used in practice [12], [13]. CPU utilization can possibly be estimated roughly using the computational complexity of an algorithm. However, different applications make differing use of various CPU units and other system resources like memory and a CPU utilization model is not accurate across wide application categories. Interestingly, the workload-sensitive nature of CPU-based models has been recently cited as a reason to go back to using detailed event counters in [14] for predicting processor and memory power usage under voltage scaling. Application-aware power modeling has the potential to assist energy aware algorithmic engineering. In [15], the authors create power profiles for each application and use it to estimate the power drawn by a consolidated server hosting the applications. *WattApp* [16] also uses power profiles for an application and estimates the power consumed with changes in workload as well. However, all these techniques are measurement-based, whereas algorithmic engineering needs energy models that are based on first principles. A good comparison of various system-level power models is presented in [17].

These problems have also garnered a lot of attention in the theory community: see e.g. the recent survey by Albers [18], which gives an overview of the various threads of energy related research in the theory community. These include speed-scaling [19], [20] and dynamic powering down a machine [21]. Both of these general approaches lead to scheduling problems in both the online and offline settings. However, all of these treat algorithms as black boxes while our work deals directly with the algorithms. We believe that our work complements this body of existing work.

The first asymptotic energy model for algorithms was presented in [1]. As is the norm with traditional algorithm design, the energy model is asymptotic. A couple of simulation results were presented, where it was shown that algorithms designed for the "traditional" algorithmic model can be transformed into algorithms whose memory accesses are highly parallelized and thus, consume much less energy than the naive implementation. These simulations led to design of energy optimal (in the asymptotic sense) algorithms for certain basic problems such as sorting and matrix transpose. However, due to the fact that the authors were only interested in the asymptotic behavior, the simulations usually ran the following trick: they only used $P/2$ banks for the actual computation and used the rest of $P/2$ banks for book keeping in their simulations (the simulations basically amounted to randomly changing the memory layout for the original algorithm and storing these random maps in $P/2$ banks so that one could also access the map with high parallelizability). Of course this implied that they were potentially losing a factor of 2 from the maximum parallelizability of P, which translates to similar loss in the energy consumed. However, implementations needs to be aware of the exact constants used in the complexity model for practical implementation.

Our model also has similarities with the cache oblivious model [22]. In this model, the goal is to minimize the number of cache misses (equivalent to the number of I/Os), while ensuring that the algorithm is also work optimal. Our model differs from this work in three ways: (i) The cache model assumes 'ideal cache' with high associativity whereas we have an associativity of 1 (ii) The tall cache assumption does not hold in our model due to the fact that only one block can be transferred from any memory back into its local cache (iii) We aim to minimize a linear combination of work complexity and number of parallel I/Os whereas the cache oblivious model tries to minimize both the work complexity and the number of sequential I/Os without worrying about parallelism.

VIII. Acknowledgments

Part of this work was done when SR was at University at Buffalo and supported by NSF grant CCF-0844796. AR's research is party supported by NSF grant CCF-0844796.

We thank the anonymous reviewers for comments that helped improve the presentation of the paper.

References

[1] S. Roy, A. Rudra, and A. Verma, "An energy complexity model for algorithms," in *ITCS 2013*.

[2] H. Monitor, "http://www.bresink.com/osx/HardwareMonitor.html."

[3] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *SIGMOD 2010*.

[4] J. Leskovec, "Stanford large network dataset collection," http://snap.stanford.edu/data/index.html.

[5] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. ISCA*, 2000.

[6] S. G. et al., "Using complete machine simulation for software power estimation: The softwatt approach," in *HPCA*, 2002.

[7] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, "Full system power analysis and modeling for server environments," in *WMBS*, 2006.

[8] F. Bellosa, "The benefits of event-driven enery accounting in power-sensitive systems," in *Proc. SIGOPS European Workshop*, 2000.

[9] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam, "Managing server energy and operational costs in hosting centers," in *Sigmetrics*, 2005.

[10] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat, "Ecosystem: Managing energy as a first class operating system resource," in *ASPLOS*, 2002.

[11] L. Barroso and U. Hlzle, "The case for energy proportional computing," in *IEEE Computer*, 2007.

[12] T. Heath, B. Diniz, E. V. Carrera, W. M. Jr., and R. Bianchini, "Energy conservation in heterogeneous server clusters," in *Proc. PPoPP*, 2005.

[13] P. Ranganathan, P. Leech, D. Irwin, and J. Chase, "Ensemble-level power management for dense blade servers," in *Proc. ISCA*, 2006.

[14] D. Snowdon, S. Petters, and G. Heiser, "Accurate on-line prediction of processor and memory energy usage under voltage scaling," in *EMSOFT*, 2007.

[15] J. Choi, S. Govindan, B. Urgaonkar, and A. Sivasubramaniam, "Profiling, prediction, and capping of power consumption in consolidated environments," in *MASCOTS 2008*.

[16] R. Koller, A. Verma, and A. Neogi, "Wattapp: an application aware power meter for shared data centers," in *ICAC 2010*.

[17] S. Rivoire, P. Ranganathan, and C. Kozyrakis, "A comparison of high-level full-system power models," in *HotPower*, 2008.

[18] S. Albers, "Energy-efficient algorithms," *Commun. ACM*, vol. 53, pp. 86–96, May 2010. [Online]. Available: http://doi.acm.org/10.1145/1735223.1735245

[19] F. F. Yao, A. J. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *FOCS*, 1995, pp. 374–382.

[20] N. Bansal, T. Kimbrel, and K. Pruhs, "Speed scaling to manage energy and temperature," *J. ACM*, vol. 54, no. 1, 2007.

[21] S. Irani, G. Singh, S. K. Shukla, and R. K. Gupta, "An overview of the competitive and adversarial approaches to designing dynamic power management strategies," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, pp. 1349–1361, December 2005. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2005.862725

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS*, Washington, DC, USA, 1999, pp. 285–297. [Online]. Available: http://dl.acm.org/citation.cfm?id=795665.796479