Rethinking Query Processing for Energy Efficiency: Slowing Down to Win the Race

Paper # 286

ABSTRACT

The biggest change in the TPC benchmarks in over two decades is currently underway – namely the addition of an energy efficiency measure when reporting TPC benchmark numbers. This change is fuelled by the growing, real and urgent demand for energy-efficient database processing. As we show in this paper, database query processing engines must truly become energy aware, else they risk missing many opportunities for significant energy savings. The focus of this paper is on the design and evaluation of a new framework for query optimization that considers both traditional performance and energy consumption as first-class optimization criteria. Our method recognizes and exploits the evolution of modern computing hardware that allows hardware components to operate in different energy and performance states. Our optimization framework considers these energy efficiency profiles and uses an energy consumption model for database query operations. We have built these models for an actual commercial DBMS. We demonstrate that our models accurately predict the power consumption for a wide variety of join queries across different system states with an average error rate of about 3% and a peak error rate of 8%. Using these models the query optimizer can now pick query plans that meet traditional performance goals (e.g., specified by SLAs), but result in lower energy consumption. Our experimental evaluations show that our energy savings can be big – e.g., 23% reduction in energy for only a 5% increase in response time for an equijoin query.

1. INTRODUCTION

Energy management has become a critical aspect in the design and operation of database management systems (DBMSs). The emergence of this new paradigm as an optimization goal is driven by the following facts: a) Servers consume tremendous amount of energy – 61B kilowatt-hours in 2006 alone and doubling by 2011 [8]; b) The energy component of the total cost of ownership (TCO) for servers is already very high, and growing rapidly. The server energy component of the three-year TCO is expected to dwarf its initial purchase cost [15]. A big contributing factor to this trend is that processors are expected to continue doubling in the number of cores every 18 months, but the performance per watt doubles at a slower rate of once every two years [14]; c) To make matters worse, typical servers are over provisioned to meet peak demands, and as a result are idle or underutilized most of the time. Barroso and Hölzle [12] have reported average server utilization in the 20–30% range; d) Unfortunately, when servers are idle, or nearly idle, they tend to consume energy that is disproportional to their utilization – for example, an idle server often consumes more than 50% of its peak power [12].

With these rising energy costs and energy-inefficient server deployments, it is clear that there is a pressing need for DBMSs to evolve to directly consider energy efficiency as a first class operational goal. In fact, driven by requests from its customers, the Transaction Processing Performance Council (TPC) has started to move in this direction. As of the preparation of this paper, the TPC has begun to release an interim specifications of their newest upcoming benchmark: TPC-Energy. This specification introduces the rules and methods for reporting an energy metric in their existing TPC benchmarks. Database and hardware vendors that wish to report future TPC results will have a keen interest in minimizing their "power/performance" results. The challenge will be to reduce the energy consumption of a DBMS while maintaining the performance levels typically expected and accepted by database users. While the first version of this benchmark has resulted in a compromise that makes this energy reporting optional, the organization clearly expects that "*Competitive demands will encourage test sponsors to include energy metrics as soon as possible.*" Thus, we believe that there is huge opportunity for the database community to take on this challenge head-on and find methods to make DBMSs more energy-efficient. In this paper, we tackle the query processing component and develop a framework for energy-efficient query processing.

To begin, one might think that perhaps doing business as usual might work for energy-aware query processing. Specifically, we already know how to optimize queries for response time/throughput metrics. So it is natural to pose the following question: Is processing (optimizing and executing) the query as fast as possible also the most energy-efficient way to operate a DBMS? Unfortunately, the answer to this question is no.

The reason for this negative answer has to do with typical server operational characteristics and the power/performance characteristics of hardware components in modern servers.

First, recall that as discussed above, typical servers often run at low utilization. This means that a server has many opportunities to execute the query slower, if the additional delay is acceptable. For example, this situation may occur if the Service Level Agreement (SLA) permits the additional response time penalty. Since typical SLAs are written to meet peak or near peak demand, SLAs often have some slack in the frequent low utilization periods, which could

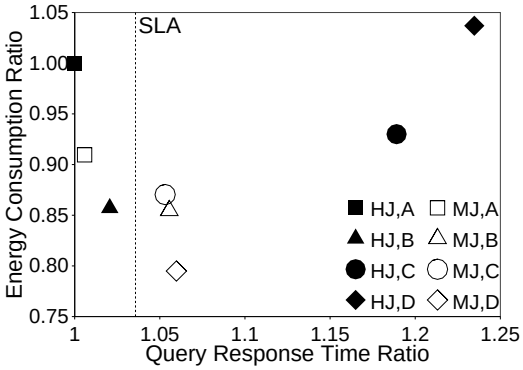


Figure 1: Energy Response Time Profile (ERP) for an equijoin query on two 50M tuple (5GB) Wisconsin Benchmark relations, on the attribute $four$, and a 0.01% selection on both relations. The energy and response time values are scaled relative to the stock settings (HJ, A) that is currently used by DBMSs.

be exploited by the DBMS. (The incentive for doing this is particularly high when the DBMS server is running as a hosted cloud service and any dollar savings that do not violate the SLAs directly and immediately adds to the bottom line.)

Second, components on modern server motherboards offer various power/performance states that can be manipulated from software. Of these components, the CPU is currently the most amenable for transitioning between such power/performance states, but nearly every power–hungry component connected to the server motherboard (e.g. memory chips and network cards) is moving towards providing multiple power/performance states for more energy-efficient operation.

This paper proposes a new way of thinking about processing and optimizing queries. In our framework, we assume that queries have some response time goal, potentially driven by an SLA. The query optimization problem now becomes: *Find and execute the most energy efficient plan that meets the SLA.*

To enable this framework, we propose extending existing query optimizers with an energy consumption model, in addition to the traditional response time model. With these two models, the enhanced query optimizer can now generate what we call an *Energy Response Time Profile (ERP)*. The ERP is a structure that details the energy and response time cost of executing each query plan at every possible power/performance setting under consideration. The ERP of a query can then be used to select the appropriate "energy-enhanced" startegy to execute the query.

To show the potential and validity of this approach, in Figure 1 we show an actual ERP for a single equijoin query on two Wisconsin Benchmark relations [19]. The plot shows the system energy consumption and response time measurements for executing the query using two different query plans – hash join (HJ) and sort-merge join (MJ) at different system settings (labeled A–D), on an actual commercial DBMS. The energy measurement is the actual energy that is drawn by the entire server box (i.e. we measure the energy drawn from the wall socket by the entire system.) System setting A corresponds to the "stock" (high power/performance) system settings. This is the setting currently used by DBMSs. System settings B and C are medium power/performance settings that produce power savings by reducing the front side bus (FSB) frequency and the memory capacity respectively. System setting D is a low power/performance setting where both the FSB frequency and memory capacity are reduced. Data point (HJ, A) is the default setting that the commercial DBMS would normally use. In this figure, we have plotted all the other data points proportionally relative to (HJ, A) for both the energy consumption (on the y-axis), and the response time (on the x-axis). Traditional query optimizers often use response time as the primary metric so they would choose HJ and system setting A to execute the query, since this is the fastest query plan. However, by choosing the sort-merge (MJ) plan and system setting D, we can reduce the energy consumption by more than 20% for only a 6% increase in response time. Optionally, by choosing MJ and system setting A we can achieve nearly 10% energy savings in lieu of a negligible (<1%) increase in response time.

However, it may not always be possible to choose a lower energy setting to run the query due to performance constraints. For example lets say an SLA exists between the client and the server, which only allows for an additional 4% delay over the optimal response time (HJ, A). This SLA is represented in Figure 1 as a vertically dotted line. Now the most energy-efficient choice is HJ and system setting B, which reduces the energy consumption by 14% and still meets the SLA! This ability to optimize for energy while maintaining performance constraints makes our framework particularly attractive for DBMSs.

To keep the scope of this work limited to a single paper, here we focus on an important class of queries – namely, single join queries with selection and projections. However, our framework can be extended for more complex query processing, using the single join queries as essential building blocks. As the reader will see, our research points to many open research problems in this emerging field of energy-aware data management, only part of which we can address in this paper. Other recent papers have also made the observation that there are many opportunities for new and challenging research opportunities in this emerging area of energy-aware data procesing [26, 29], which we echo in this paper.

This paper makes the following contributions.

- We present a new design for query optimizers that uses both energy and performance as optimization criteria.
- We present the notion of an Energy Response Time Profile (ERP) that can be used by our query optimization framework to explore all the combinations of system power/performance settings that the hardware provides. The ERP can then be used by the optimizer in picking an "energy-enhanced" query plan that meets any existing response time targets.
- We present energy cost models that can be used to generate the ERP. These energy models are simple and portable across a variety of hardware.
- We validate the energy model using an actual commercial DBMS and demonstrate the end-to-end benefit of our approach, which can result in significant energy savings.

The reminder of this paper is organized as follows. In Section 2 we present our new query optimization framework. Section 3 describes the energy cost models that are used to generate ERPs. Section 4 contains our experimental results. Related work is discussed in Section 5, and Section 6 contains our concluding remarks and directions for future work.

2. FRAMEWORK FOR ENERGY AND RESPONSE TIME OPTIMIZATION

In this section we present a general framework of a system that uses both energy and response time as the optimization criteria. The questions we tackle are: (1) How to (re-)define the job of the query optimizer in light of its new responsibility to optimize for energy consumption? (2) Given this new role, how do we design a query optimizer? We discuss answers to these questions below.

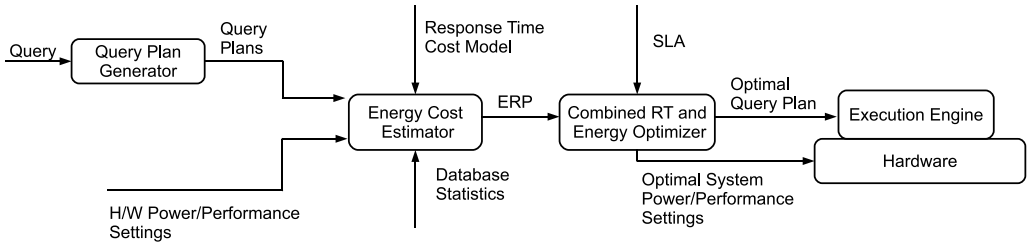


Figure 2: An overview of the framework that optimizes for both energy and response time. The energy cost estimator is described in more detail in Figure 3.

2.1 New Role of the Query Optimizer

The traditional query optimizer is concerned with one thing only: Optimize the query for response time performance. The optimizer's new goal now is to find query plans that have an *acceptable* response time, but consume *as little energy as possible.* As shown in Figure 1, we want the query optimizer to return an energy-enhanced query plan that is to the left of the SLA-dictated performance requirement, and as low as possible along the y-axis. The main challenging task here is to generate ERP plots like that shown in Figure 1, *without* actually running the different plans and actually measuring the response time and energy consumed. To generate these plots, the query optimizer needs to *quickly* and *accurately* estimate both the response time and the energy consumption for each query plan. Query optimizers today are fairly good at predicting the response time, so the key new challenge is enhancing optimizers to allow them to predict the energy consumption of a query plan. To predict the energy consumption, the query optimizer has to understand the power/performance settings that the hardware offers.

2.2 Hardware States, Optimization, and ERP

Driven by the need for energy-efficient computing, hardware components are increasingly becoming energy-aware, and offer multiple power/performance settings that can be manipulated directly from software at runtime. For instance, modern CPUs have multiple performance states (p-states) that allow the CPU to perform at varying frequency and voltage (and hence varying power consumption states). An example of this capability is p-state capping, which caps the maximum CPU p-state. So a p-state capping at 75% increases the response time of a CPU-bound query by approximately 25%. Furthermore, modern CPUs have the ability to "park" entire cores (making them inactive), and thereby dropping the power drawn by the processor. This ability is not limited to CPUs, and the ability to "park" main memory DIMMs is on the horizon, as memory has been identified as a crucial power-hungry component on the motherboard. Such products have been created [2], and memory vendors are actively working on making this technology widespread.

The "energy-enhanced" query optimizer must be provided with a set of system operating settings, where each system operating state is a combination of different individual hardware component operating settings. The optimizer then uses an energy prediction model along with its current response time model to produce an ERP for that query, like the example shown in Figure 1.

An ERP consists of a set of energy-enhanced query plans for the query being optimized. Given a query Q with a set of possible logical plans $P = \{P_1, P_2, ..., P_n\}$ that is to be executed on a machine with system operating settings $H = \{H_1, H_2, ..., H_m\}$, the ERP contains one point for every plan for every system operating setting (and hence an ERP has $n \times m$ points). Thus we expand the search space that is explored by traditional query optimizers by m. Note that heuristics could be developed to prune the system operating settings H so that only a small subset of the m settings are explored for specific queries – e.g. some operating settings may have a low chance of being interesting for certain query types and could be pruned heuristically. (An interesting direction for future work is to explore this option.)

Several methods can be envisioned to predict the energy cost of a query plan. One simple method is to use the response time (which current optimizer can estimate) to estimate the energy consumption as: Energy α Response Time. But, as our paper shows (Section 4) such simple models are not very accurate. Note that an accurate energy estimation model is essential, so that the optimizer does not make a wrong choice – such as choosing query plan *HJ* and system setting *D* in Figure 1, which incurs penalties for both response time and energy consumption.

To enable this new query optimizing framework, in this paper, we develop an accurate analytical model to estimate the energy consumption cost for evaluating a query plan. We discuss this model in more detail in Section 3.

2.3 Our Implementation

In this section we discuss our implementation of the framework described above. We had two implementation options – the first one was to integrate this framework into an existing query optimizer in a open source DBMS like MySQL, and the other to implement it as an external module for a commercial DBMS. We have found that the commercial DBMS that we are using is appreciably faster than the open-source alternative – often we have seen 10X speedup for complex queries. Consequently, we decided to use the commercial DBMS for our implementation. This choice also forced us to think of a design that is generic and likely to be more portable across other systems, as we have to build the framework using only the high level interfaces that the DBMS provides. (Besides, if one is concerned about energy efficiency, potentially starting with a system that is significantly faster is likely to be a better choice.)

Figure 2 gives an overview of our implementation. The query is supplied as input to the query plan generator in the commercial DBMS, which is then requested to list (not execute yet) all the promising query plans that the optimizer has identified. These query plans along with the information about available system settings is provided as input to the Energy Cost Estimator, which generates the ERP using an analytical model for predicting the energy consumption (see Section 3). The generated ERP is then used by the combined energy-enhanced query optimizer to choose the most energy efficient plan that meets any SLA constraints. Then, a command to switch to the chosen system operating state is sent to the hardware, followed by sending the optimal query plan to the execution engine.

3. ENERGY COST MODEL

In this section we develop a analytical model that estimates the energy cost of executing a query at a particular power/performance system setting. Our model abstracts the energy cost of evaluating a query in terms of system parameters that can be learnt through a "training" procedure, and query parameters that can be estimated from the available database statistics. This section is organized as follows: In Section 3.1 we provide a overview of our model. In Section 3.2 we discuss four different models that abstract the energy cost in terms of system and query parameters. The operator model that can be used to estimate the query parameters is discussed in Section 3.3 and the training procedure used to learn the system parameters is discussed in Section 3.4.

3.1 Model Overview

We want to develop a simple, portable, practical, and accurate method to estimate the energy cost of a query plan. Unfortunately prior techniques used to estimate energy consumption fail to satisfy one or more of these goals. For example a circuit level model of hardware components [1, 6, 23] can accurately predict the energy consumption, but these models also have a high computational overhead which make them impractical for query optimization. On the other hand, a high level model that treats the entire system as a black box, though simple and portable, is not very accurate.

In our approach, we use an analytical model that offers a compromise between these two extremes. In our model, the power drawn by different hardware components are abstracted into learnable system parameters. The intuition behind this approach is simple. At a particular power/performance system setting, the power consumption of a hardware component, which is in a specific power state, can be assumed to be a constant, independent of the query being executed. For example, let us assume that the CPU can exist only in two power states *active* and *idle*. The CPU is in active mode when it has instructions to execute in its queue, and is idle when it is waiting for a (memory or disk) I/O. Now the power drawn by the CPU at the active (P_{active}) and idle states (P_{idle}) are independent of the query that is being executed. Only the fraction of execution time that the CPU spends in each state depends on the workload. And these can be estimated from the available database statistics (Section 3.3). For example, suppose we have executed some query Q and have learnt T_{active} and T_{idle}, the length of time the CPU is *active* and *idle* respectively. Now, let $Energy = P_{active}T_{active} + P_{idle}T_{idle}$. By executing many queries of the same class as Q and measuring $Energy$, we can solve for the unknown P values. So, if we have enough measurements we can get a reasonably accurate estimate of the power drawn by the hardware components at each power state. We discuss this process in more detail in Section 3.4.

Figure 3 gives an overview of the Energy Cost model that we have designed. The operator model takes as input the query plan and uses the database statistics available about the input relations to estimate the query parameters that is required to estimate the energy cost. From our example, this is when we estimate the values of T_i. The most complex model that we describe in Section 3.2 requires four query parameters I_Q, the estimated number of CPU instructions, M_Q, the estimated number of memory accesses, R_Q, the estimated number of disk read requests and W_Q, the estimated number of disk write requests that will be made during the execution of the query. We discuss a model to estimate these parameters for selected database operations (selection, projection, joins) in Section 3.3. The hardware abstraction model takes as input the query parameters estimated by the operator model plus the response time model (since response time is dependent on system settings) and estimates the energy cost of evaluating a query using a particular query plan at a particular power/performance system setting.

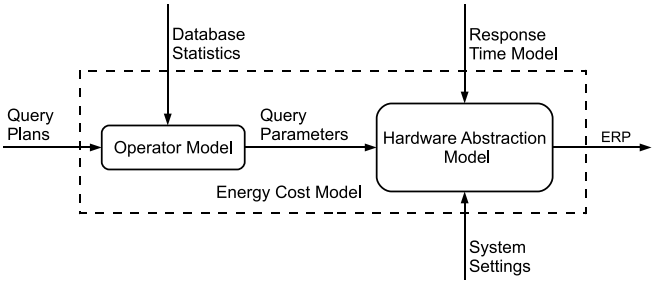


Figure 3: An overview of the Energy Cost model used in our implementation. The operator model estimates the query parameters required by the hardware abstraction model for each query plan from the database statistics available. The hardware abstraction model uses the query parameters estimated and the system parameters learnt to accurately estimate the energy cost.

The ERP is generated by estimating the energy cost for each query plan and at each power/performance system operating setting.

We discuss four models that can be used to estimate the energy cost in Section 3.2. The advantages of the energy cost models that we propose here are:

- **Simplicity:** The models require no additional database statistics other than those used in traditional query optimizers for response time cost estimation. Also, minimal overhead is incurred by the query optimizer in calculating the most energy efficient power/performance operating setting and query plan. The computational complexity is $O(|H| * |P|)$ where H is the set of valid power/performance system settings and P is the set of query plans for the query.
- **Portability:** The model makes few assumptions about the underlying hardware or the database engine internals and can be ported across many DBMSs and machines. In fact, we have been able to implement this model on top of a commercial DBMS, treating it as a black box, and the model still achieves high accuracy.
- **Practical:** Detailed system simulators like DRAMSim [1] and Sim-Panalyzer [6] model hardware components at the circuit level to estimate power consumption. This process though accurate is computationally very expensive, and is hence not practical for use in a query optimizer. In our model we abstract the power drawn by different components into learnable system parameters.
- **Accuracy:** As we show in Section 4 our model is very accurate, and in our tests it has an average error rate of around 3% and a peak error rate of 8%.

3.2 Hardware Abstraction Model

In this section, we propose four models to predict the energy required to execute the query.

We use a simple energy model, namely[1]:

$$E = P_{av} * T$$

where T is the response time of the given query plan, P_{av} is the average wall power drawn during the query execution, and E is the energy consumed during query execution. Commercial and open-source database engines already have techniques that provide good

[1]More complex models could be used that don't assume average power drawn, but these can get complicated. Our goal here is to produce a simple and usable model. An interesting direction of future work is to explore more complex energy models.

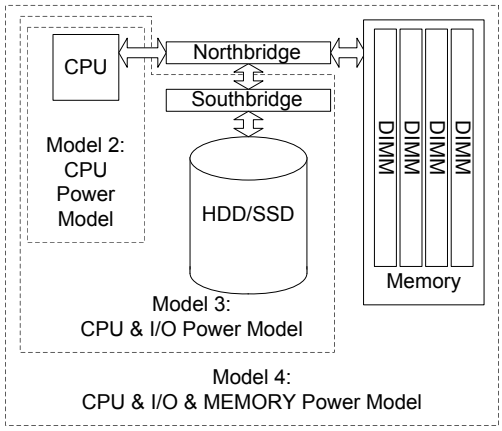


Figure 4: Progression of complexity in the models. The power drawn of each component within the demarcation line is modelled explicitly. All consumption outside a particular model demarcation line is assumed constant in that model.

estimates for T, hence we assume that this information is available and only present a method for estimating P_{av}.

We begin with a simple model to estimate P_{av}, and then incrementally build on top of this simple model to obtain more accurate models. Later we show that these more sophisticated models are needed. Figure 4 depicts the incremental building up of the model. Essentially, we start with a static constant power model (not shown in Figure 4) where we assume no power fluctuations during query execution. Gradually, as shown in Figure 4, we introduce the power fluctuations of various components (CPU, storage disks, and memory) into the models. For simplicity we assume that the database engine is the only active process and that only the query under consideration is executing in the database engine. Extending the framework that we present here for multiple queries opens up new issues, which would be an interesting direction for future work.

Each of the models presented in this section depend on a different set of query and system parameters. The query parameters depend on the query in question and are estimated by the operator model discussed in Section 3.3. The system parameters are independent of the query and are constant for a given power/performance system setting. These parameters are estimated for each possible power/performance system setting using the training procedure described in Section 3.4.

3.2.1 Model 1: Constant Power Model

We begin with the simplest model possible. This model assumes that P_{av} is a constant for a given power/performance operating setting and is independent of the query being executed. As per this model:

$$P_{av} = C_{other}$$

where C_{other} is a constant for a given power/performance operating setting. The weakness of this model is that it assumes that all the hardware components in the system remain at the same power state during query execution. This is often not true as during query execution, the CPU may wait for I/O or memory operations and the wait time depends on the query. We rectify this flaw in Model 2.

3.2.2 Model 2: CPU Power Model

This model takes into account the possibility that the average power consumption of the CPU varies across different queries. As shown in Figure 4, in this model we model the changes in the power drawn by the CPU, but assume that all other hardware components draw constant power.

Let P_{cpu} be the power drawn by the CPU, and P_{other} be the power drawn by the other system components. This gives:

$$P_{av} = P_{cpu} + P_{other} \quad (1)$$

Modern CPU can dynamically change their power/performance states (called p-states). For simplicity we assume that the CPU can operate only in the two extreme p-states – namely *active*, when it has instructions to execute and *idle* when it has been blocked by (memory or disk) I/O. When the execution of the query is blocked by I/O, the CPU is in the idle state as the query execution is assumed to be the only active process in the system. Let T_C be the CPU time (amount of time the query uses the CPU), P_{act} be the average power drawn by the CPU when it is in the active state, and P_{idle} be the average power drawn by the CPU when it is in the idle state. Now, the average CPU power draw is:

$$P_{cpu} = P_{act} * \frac{T_C}{T} + P_{idle} * \frac{(T - T_C)}{T}$$

Ignoring the power drawn when the CPU transitions from one state to the other, and grouping together terms in the equation above gives:

$$P_{cpu} = (P_{act} - P_{idle}) * \frac{T_C}{T} + P_{idle}$$

Now let I_Q be the number of CPU instructions that has to be executed to evaluate the query Q. I_Q can be estimated by the operator model from existing database statistics. (Section 3.3) In addition, let T_I be the time taken to execute a single instruction. This gives:

$$T_C = T_I * I_Q$$

Using $C_{cpu} = (P_{act} - P_{idle}) * T_I$ (which is a constant dependent only on the power/performance setting) and substituting in equation 1 for P_{cpu} we get:

$$P_{cpu} = C_{cpu} * \frac{I_Q}{T} + P_{idle} \quad (2)$$

Substituting P_{cpu} back in equation 1 and putting $C_{other} = P_{other} + P_{idle}$ we get:

$$P_{av} = C_{cpu} * \frac{I_Q}{T} + C_{other} \quad (3)$$

This model has two system parameters C_{cpu}, C_{other} that must be learnt by the training procedure described in Section 3.4, and one query parameter I_Q that needs to be estimated by the operator model (Section 3.3).

The weakness of this model is that it assumes that the power drawn by the disk is a constant across all queries. This is often not true as the fraction of execution time that is spent on disk I/O can vary widely from one query to another. The idle power drawn by modern disk can also be significantly smaller than the power drawn when they are active (especially the ones designed for energy efficiency like the Seagate disks with PowerChoice [3]). We rectify this limitation in Model 3.

3.2.3 Model 3: CPU and I/O Power Model

This model takes into account the possibility that the average power drawn by the I/O unit varies varies across different queries.

Term	Description
P_R	Power drawn by the disk during a read operation
P_W	Power drawn by the disk during a write operation
T_R	Time taken to read a page from disk
T_W	Time taken to write a page to disk
P_{didle}	Power drawn by an idle disk
P_{bus}	Power drawn by I/O bus to transfer data to disk
B_{bus}	Bandwidth of the I/O bus
T_{bus}	Time to transfer a page from disk to memory
$P_{(I/OC)}$	Power drawn by the I/O controller when active
T	Response time of query Q
N_{act}	Number of active memory DIMMs
$P_{R/W}$	Power drawn by a DIMM in read/write state
P_{act}	Power drawn by a DIMM in active state
T_M	Memory access latency
P_{fsb}	Power drawn by the FSB during data transfer
B_{fsb}	Bandwidth of the FSB
T_{fsb}	Time to transfer a page from memory to CPU
P_{MC}	Power drawn by the memory controller
R_Q	Number of disk page read requests when executing query Q
W_Q	Number of disk page write requests when executing query Q
I_Q	Number of CPU instructions when executing query Q
M_Q	Number of page requests (r/w) when executing query Q

Table 1: Hardware abstraction variables used in our discussion.

In Figure 4, we have now included persistent storage such as traditional disks or newer solid state disks (SSDs) as capable of having variable power draw. According to this model

$$P_{av} = P_{cpu} + P_{I/O} + P_{other} \tag{4}$$

The CPU power model described in equation 2 can be used as is in this model, so here we focus on estimating $P_{I/O}$. In this model $P_{I/O}$ accounts for the power drawn by the disk drives used for permanent storage, the southbridge which acts as the I/O Controller, and the data bus used to carry data to and from the disk. We here assume that the system has only one disk drive for simplicity. We also ignore the power consumption of the bus and the I/O controller when they are idle. Consider Table 1 which gives us the terms used in Model 3.

Using these terms, we can provide the power model for the disk as:

$$\begin{aligned} P_{I/O} &= P_R * \frac{R_Q * T_R}{T} + P_W * \frac{W_Q * T_W}{T} + \\ & P_{didle} * \frac{T - R_Q * T_R + W_Q * T_W}{T} + \\ & P_{bus} * \frac{(R_Q + W_Q) * T_{bus}}{B_{bus} * T} + \\ & P_{I/OC} * \frac{R_Q * T_R + W_Q * T_W}{T} \end{aligned}$$

In this model we differentiate between disk read and write operations as they could be asymmetric and the power consumed during these operations could be vastly different. For example in SSD drives, while pages are read in units of flash pages, writes are in units of flash blocks, which consists of several flash pages and are far more expensive. Now grouping together terms and putting $C_R = (P_R - P_{didle}) * T_R + P_{bus} * T_{bus}/B_{bus} + P_{I/OC}$ and $C_W = (P_W - P_{didle}) * T_W + P_{bus} * T_{bus}/B_{bus} + P_{I/OC}$ in the equation above we get:

$$P_{I/O} = C_R * \frac{R_Q}{T} + C_W * \frac{W_Q}{T} + P_{didle} \tag{5}$$

C_R and C_W are constant across all queries for a given operating setting. Now substituting for $P_{I/O}$ and P_{cpu} (from equation 2) in equation 4 and putting $C_{other} = P_{other} + P_{idle} + P_{didle}$ we get:

$$P_{av} = C_{cpu} * \frac{I_Q}{T} + C_R * \frac{R_Q}{T} + C_W * \frac{W_Q}{T} + C_{other} \tag{6}$$

This model has four system parameters: C_{cpu}, C_R, C_W and C_{other}, which are learnt by using the training procedure described in Section 3.4. This model also has three query parameters: I_Q, R_Q and W_Q, which are estimated by the operator model (Section 3.3).

Now the weakness of this model is that it assumes that the average power drawn by the main memory unit is constant across different queries. This is not true as the number of memory requests placed per second during query execution varies widely across different queries and the energy consumed by the memory module varies based on the memory activity [2, 20]. We rectify this limitation in the final model below.

3.2.4 Model 4: CPU, I/O and Memory Power Model

In this model, P_{mm} represents the power drawn by the main memory unit during query execution, so:

$$P_{av} = P_{cpu} + P_{I/O} + P_{mm} + P_{other} \tag{7}$$

The CPU and I/O power models described in equation 2 and equation 5 respectively can be used directly for this model. So, we only concentrate on modeling P_{mm}. The main memory unit consists of the RAM memory modules (DIMMs), the memory controller and the front side bus, which carries data between the CPU and the northbridge.

The main memory modules available today have several power states that they operate in [2], but for simplicity we assume that the memory modules only operate in the following three power states: read/write, active and powerdown. In the powerdown state, the memory modules consume negligible power and cannot be used. The active state is the default state for a memory module. When a request for a page read/write arrives all the active memory modules transition to the read/write power state. (Though the page may exist in only one of the DIMMs all the DIMMs are transitioned to the read/write power state in current memory architectures [9].) Once the memory access is complete, if there is no request pending, then the DIMMs would transition back to the active state. For simplicity we ignore the energy consumption due to the transition of the DIMMs from one state to another. Now the power drawn by the main memory module is given by Table 1 of terms and equation for P_{mm}:

$$\begin{aligned} P_{mm} &= N_{act} * [P_{R/W} * \frac{M_Q * T_M}{RT} + P_{act} * \frac{(T - M_Q * T_M)}{T}] \\ & + P_{fsb} * \frac{M_Q * T_{fsb}}{B_{fsb} * T} + P_{MC} * \frac{M_Q}{T} \end{aligned}$$

Now, putting $C_{mm} = N_{act} * (P_{R/W} - P_{act}) * T_m + P_{fsb} * T_{fsb}/B_{fsb} + P_{MC}$ in the equation above we get:

$$P_{mm} = C_{mm} * \frac{M_Q}{T} + P_{act} \tag{8}$$

C_{mm} is independent of the query, and only depends on the current power/performance operating setting. Now substituting back for P_{mm}, P_{cpu} (from equation 2) and P_{disk}(from equation 5) in

equation 7 and putting $C_{other} = P_{other} + P_{idle} + P_{didle} + P_{act}$, we get:

$$P_{av} = C_{cpu} * \frac{I_Q}{T} + C_R * \frac{R_Q}{T} + C_W * \frac{W_Q}{T} + C_{mm} * \frac{M_Q}{T} + C_{other} \quad (9)$$

This model has five system parameters C_{cpu}, C_{mm}, C_R, C_W and C_{other}, that need to be learnt by the learning procedure described in Section 3.4, and four query parameter I_Q, M_Q, R_Q and W_Q that needs to be estimated by the operator model (Section 3.3).

3.3 Operator Model

In this section we discuss methods to estimate the query parameters I_Q, M_Q, R_Q and W_Q for the basic query operations: selection, projection and equijoins. Recall that these query parameters correspond to: the estimated number of CPU instructions, the estimated number of memory accesses, the estimated number of disk read requests and the estimated number of disk write requests, respectively. Query parameters for more complex query operations can be derived in a similar fashion, and is part of future work.

3.3.1 Notation and Assumptions

For our discussion we assume that we have two relations **R** and **S**, where **S** is the larger relation. Let $||R||$, $||S||$ denote the cardinality of the relations and $|R|$, $|S|$ denote the number of DBMS pages occupied by the relation. Let **M** denote the workspace allocated to the execution of the query in main memory. sel refers to the selectivity of the predicates in the query and is assumed to be available [43]. For simplicity we assume that all the queries are run "cold", i.e., no part of the tables reside in main memory initially. Also, we assume that no indices exists on the tables and that the result of each query is stored in another table on disk.

In order to estimate the query parameters required for power estimation, we extend the model used by Shapiro [44]. Tasks performed by the CPU is abstracted into four CPU operations - $comp$, $hash$, $swap$ and $save$.

For simplicity we assume that all memory/disk requests are in multiples of $PageSize$, the DBMS page size. Each disk read is assumed to account for two memory accesses, one to write the data to memory from disk, and one to read the memory location to the CPU cache. Similarly each disk write also accounts for two main memory accesses, one to write to memory from the CPU cache and one to read from memory to write to disk.

3.3.2 Selection Operation

A selection operation extracts tuples from a relation that satisfies a specified predicate. A selection query can be evaluated by scanning the entire relation, checking the condition on each tuple and adding it to the result if the predicate is satisfied. In order to do this, the CPU has to perform one comparison for each tuple and one save operation for each tuple that satisfies the predicate.

Assume a selection query on the relation R. Since the query is run cold, we need $|R|$ disk reads to read each page of the input relation into memory. The output relation occupies $sel * |R|$ pages that need to be written to disk, where sel is the selectivity of the predicate. Since each disk read/write accounts for two memory accesses each, the total number of memory accesses is twice the sum of disk reads and writes. For this query, the query parameters can be estimated as:

$$I_Q = ||R|| * comp + sel * ||R|| * save$$

$$M_Q = 2 * |R| + 2 * sel * |R|$$

$$R_Q = |R|$$

$$W_Q = sel * |R|$$

3.3.3 Projection Operation

A projection operation extracts specified columns from a relation, say R. A projection operation can be evaluated by scanning the entire relation and adding the desired attributes from each tuple to the result. In order to do this, the processor has to perform one $save$ operation for each tuple in the relation. Once again $|R|$ disk reads are required to read each page of the input relation into memory. The number of pages occupied by the output relation is given by $\frac{sizeof(attrs)*|R|}{sizeof(R)}$, where $attrs$ is the set of projected attributes. The result tuples need to be written to disk, and once again the total number of memory accesses is twice the sum of disk reads and writes. For this query, the query parameters can be estimated as:

$$I_Q = ||R|| * save$$

$$M_Q = 2 * |R| + 2 * \frac{sizeof(attrs) * |R|}{sizeof(R)}$$

$$R_Q = |R|$$

$$W_Q = \frac{sizeof(attrs) * |R|}{sizeof(R)}$$

3.3.4 Equijoin Operation

Equjoins are the most common type of joins that are executed by DBMSs, and there are several join algorithms for evaluating equijoins (e.g. [44]). To keep this paper manageable, here we focus only on two of the most popular equijoin algorithms – namely sort-merge join and hash join. We discuss two basic implementations of these algorithms suggested by Shapiro [44] and present the query parameters estimated in each case. These estimates could be easily modified to account for the many optimizations suggested in literature for these algorithms, to match the specific DBMS implementation.

Sort-Merge Join: This algorithms sorts both relations on the join attribute and computes join results during the merging phase. The sorting step requires a scan over each relation in order to produce the sorted runs. This step requires $||R||log_2||R|| + ||S||log_2||S||$ comparisons and swaps. We here assume that the amount of available memory $|M| \geq \sqrt{|S|}$, so that the number of sorted runs is at most $\sqrt{|S|}/2$. The initial read of the input tables requires $|R| + |S|$ disk requests. Portion of the sorted runs that do not fit in memory, need to be written to disk and read back in the next phase, requiring $|R| + |S| - min(|R| + |S|, |M| - sqrt|S|)$ disk reads and writes.

The merging phase merges these sorted runs and if a tuple from **R** matches one from **S**, it is added to the result. So the merging phase requires $||R|| + ||S||$ comparsion operations to perform the merge, and $sel * ||R|| * ||S||$ save operations to produce the result. The output relation occupies $sel * (||R|| * ||S||) * sizeof(R + S)/PageSize$ pages that need to be written to disk.

For this join algorithm, the query parameters can be estimated as:

$$I_Q = (||R|| + ||R||log_2||R|| + ||S|| + ||S||log_2||S||) * comp + (||R||log_2||R|| + ||S||log_2||S||) * swap + sel * ||R|| * ||S|| * save$$

$$M_Q = 6*(|R|+|S|) - 2*min(|R|+|S|, |M| - sqrt|S|) + 2*\frac{sel*(||R||*||S||)*sizeof(R+S)}{PageSize}$$

$$R_Q = 2*(|R|+|S|) - min(|R|+|S|, |M| - sqrt|S|))$$

$$W_Q = |R|+|S| - min(|R|+|S|, |M| - sqrt|S|) + \frac{sel*(||R||*||S||)*sizeof(R+S)}{PageSize}$$

Hash Join: Here we model the simple Grace hash join algorithm which first partitions **R** and **S** using a hash function. The hash values are partitioned in such a way that the smaller relation **R** is partioned into $\sqrt{F|R|}$ subsets of approximately equal size, where F is the fudge factor. This phase requires $||R||+||S||$ hash and save operations. The initial disk read of the tables requires $|R|+|S|$ disk accesses. The fraction of the partitioned tables that cannot fit into memory have to be written to disk and read back for the second phase, which adds $|R|+|S| - min(|R|+|S|, |M|\sqrt{F|R|})$ disk reads and writes.

In the second phase, each partition of **R** is read into memory and a hash table is constructed on each partition. Now each tuple in the corresponding partition in **S** is read and probed for a match. If there is a match, the tuple is added to the result. This phase requires $||R||$ hash operations, $||S|| * F$ comparisons, and $sel * ||R|| * ||S||$ operations to save the output tuples. The output table occupies $\frac{(||R||*||S||)*sizeof(R+S)}{PageSize}$ pages in memory that need to be written to disk.

The query parameters for hash join can be estimated as:

$$I_Q = (2*||R||+||S||)*hash + ||S||*F*comp + (||R||+||S||+sel*||R||*||S||)*save$$

$$M_Q = 6*(|R|+|S|) - 2*min(|R|+|S|, |M|\sqrt{F|R|}) + 2*\frac{sel*(||R||*||S||)*sizeof(R+S)}{PageSize}$$

$$R_Q = 2*|R|+|S| - min(|R|+|S|, |M|\sqrt{F|R|})$$

$$W_Q = |R|+|S| - min(|R|+|S|, |M|\sqrt{F|R|}) + \frac{sel*(||R||*||S||)*sizeof(R+S)}{PageSize}$$

Discussion: While we have presented a basic implementation of the join algorithms in this section, DBMSs often optimize these algorithms for better performance. One such optimization, implemented by the commercial DBMS that we tested our framework on, is to build a bloom filter on the join attribute values of the smaller relation during the initial scan and then using this bloom filter to filter out tuples from the larger relation. This technique is used for both sort merge and hash join. For optimization like these, we have to extend the model that we present above to better reflect the underlying implementation. Thus some customizations of these models may be needed when porting to specific DBMSs. (We have extended our method to incorporating bloom filters, but in the interest of space we omit this model – an extended version of our paper will detail this enhanced model.)

3.4 Training Procedure to Estimate the System Parameters

In this section we discuss methods to estimate the system parameters of the hardware abstraction models descibed in Section 3.2. The system parameters are C_{cpu}, C_{mm}, $C_{I/O}$, and C_{other}. These parameters are independent of the query and depend only on the power/performance operating setting. These parameters could be easily estimated if the hardware vendors provided the power drawn by each component at the different power/performance states that they can operate in. Unfortunately, hardware vendors currently do not divulge this information. One way to get around this drawback is to implement a per-component direct energy measurement system as described by Ryffel et al. [40]. In our implementation we use an alternate method where we learn these parameters by executing a set of "training" queries. The advantage of this method is that it requires no hardware modification (e.g. taps on the motherboard), and can be ported to a wide variety of machines.

We use a basic training procedure to learn these parameters. We define the following to be the characteristics of a single join query: size of the larger relation, ratio of the size of the larger relation to the size of the smaller relation, and selectivity of the join predicates. We collect an arbitrary mixture of N join queries that cover a wide spectrum with respect to these characteristics. At each power/performance operating setting, we execute these queries for two query plans – one for hash join and the other for merge join – and measure the wall power drawn and response time in each case. Now we estimate the query parameters using the operator model described in Section 3.3. It is also possible to measure the query parameters using system tools like *logman* but as we show in the validation section (Section 4), the operator model estimates the query parameters with high accuracy. Now we have $2*N$ linear equations on these system parameters, from which we obtain the best fit by linear regression. The system parameters are plugged back into the hardware abstraction model so that it can estimate the energy cost.

4. VALIDATION AND RESULTS

In this section we validate and compare the models that we presented in Section 3. We then deploy our model and show end-to-end results that show the effectiveness of our method in producing significant energy savings.

4.1 System Under Test

The system that we use in this paper has the following main components: ASUS P5Q3 Deluxe Wifi-AP motherboard, Intel Core2-Duo E8500, 4x1GB Kingston DDR3 main memory, ASUS GeForce 8400GS 256M, and a 32G Intel X25-E SSD. The power supply unit (PSU) used was a Corsair VX450W PSU, which is labeled as an energy efficient PSU under 80plus.org. System power draw was measured by a Yokogawa WT210 unit (as suggested by SPEC power benchmarks) connected to a client measuring system. The operating system used was Microsoft Windows Server 2008, and we use a leading commercial DBMS (the name of the commercial DBMS is suppressed).

4.2 Validation Setup

In this section we discuss the method that we used to validate the energy cost model discussed in Section 3. Our validation process consists of two steps: operator model validation and hardware

S.No	Predicate	Selectivity
1	R.onePercent = S.onePercent	$0.01/\|\|S\|\|$
2	R.twenty = S.twenty	$0.05/\|\|S\|\|$
3	R.ten = S.ten	$0.1/\|\|S\|\|$
4	R.four = S.four	$0.25/\|\|S\|\|$
5	R.two = S.two	$0.5/\|\|S\|\|$
6	R.unique2 < 0.75*$\|\|R\|\|$	$0.75/\|\|S\|\|$
7	–	$1/\|\|S\|\|$

Table 2: The different predicates of our query set along with the corresponding selectivities.

abstraction model validation.

4.2.1 Operator Model Validation Setup

First, we compare the query parameters estimated using the operator model (see Section 3.3) with the actual values obtained from the DBMS and the *logman* Windows tool, and show that these estimates are accurate. For this test, we created 70 single join queries on the Wisconsin Benchmark [19] relations. The join queries are performed on two Wisconsin Benchmark like tables **R** and **S**, where **S** is the larger relation. We modified the Wisconsin Benchmark table by removing the string attributes stringu1 and stringu2 and reducing the capacity of string4 to 48 bytes which results in a tuple size of 100B. The join queries that we use have the following template:

```
SELECT * FROM R,S
WHERE R.unique1 = S.unique2 [AND <predicate>]
```

We vary the predicate in the WHERE clause to obtain seven different selectivities. These prediates and selectivities are shown in Table 2.

We also varied the ratio of the size of the larger relation **S** to the smaller relation **R** – we used two different values $||R||/||S||$ = 1 and $||R||/||S||$ = 0.4. In addition, we varied the size of **S**, using the following five values: 500 MB, 1 GB, 2 GB, 4 GB and 5 GB. This combination of seven selectivities with ten different database scale factors gives 70 single-join queries. Finally, we limit the the query plans to be either hash join or sort-merge join.

We validated our operator model for the query parameter estimation (e.g., instruction count (I_q), memory accesses (M_q), etc.) by comparing the operator model estimates with the actual DBMS and logman reported values. Then, using our estimates as input into the hardware abstraction models (Section 3.2), we verify that the accuracy of the hardware abstraction models increases as we increase the model complexity from Model 1 to 4 (see Section 3).

4.2.2 Hardware Abstraction Model Validation Setup

For this validation, we learnt the system parameters at four system power/performance operating states, using the training procedure described in Section 3.4. The four system settings (A, B, C, and D) are combinations of different component frequencies and memory sizes. Table 3 describes the combinations of reducing FSB frequency and reducing the main memory of the system under test. We chose to change these two settings as many prior efforts [18, 20, 29] have shown significant energy benifits that could be achieved by changing the FSB frequency and memory capacity. FSB frequency was altered using the ASUS SixEngine tool which allows changes to be made to the FSB frequency/speed in software. Other settings in the SixEngine tool were left at default.

We acknowledge that present day DRAM chips do not provide a software interface to control their power states. As a result, for this experiment, we physically remove the memory modules to reduce the main memory size as software control to turn off memory banks is still not available. However we expect this functionality to be available in the near future as energy consumption is a first class design criteria in many evolving memory architectures [2] and many recent efforts [20, 30] have shown that significant energy benifits could be achieved through software control of memory power states.

	Stock (A)	Frequency (B)	Memory (C)	Both (D)
FSB	100%	90%	100%	90%
Memory	4GB	4GB	2GB	2GB

Table 3: The four system settings that we use in our experimental analysis. FSB frequency reduction is done using the ASUS SixEngine tool, and memory reductions involved physically removing DIMMs.

Operator Parameters	Average Error Ratio			Peak Error Ratio		
	HJ	MJ	Both	HJ	MJ	Both
I_Q	7.0%	8.2%	7.6%	13.9%	12.8%	13.9%
M_Q	5.1%	5.1%	5.1%	10.2%	10.1%	10.2%
R_Q	2.4%	1.0%	1.7%	4.8%	3.6%	4.8%
W_Q	1.3%	1.3%	1.3%	10.5%	10.1%	10.5%

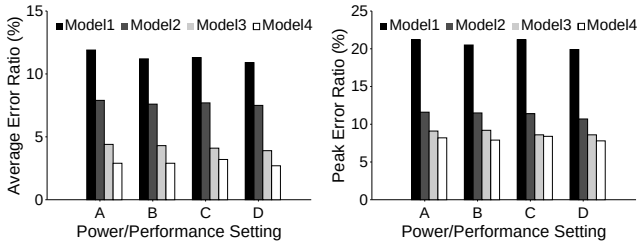
Table 4: Error ratios for the estimates made by the operator model for I_Q number of CPU instructions, M_Q, the number of memory accesses, R_Q, the number of disk reads and W_Q, the number of disk writes.

We note that we are showing cold numbers here, so changing memory capacity is simple as we don't have to worry about existing buffer pages. (There are some interesting issues related to how DBMS internals might change to adapt to evolving hardware features. For example, consider memory parking – A key question is how can the DBMS quickly relinquish a portion of memory? If the memory is used for working space (e.g. hash tables) then the problem is potentially easy. But, if the memory is being used for a buffer pool then things get more interesting. In read mostly environments, one can imagine a buffer management strategy that works in conjunction with the operating system's virtual memory manager to quickly find DIMMS with only clean pages, and using methods to quickly shrink the buffer pool. For updates, one can imagine various modifications to the buffer manager, such trying to keep updates on DIMMS that are the last ones to be powered down for energy saving. We plan on investigating such issues as part of future work.)

We picked 35 of the 70 join queries (Section 4.2.1) randomly for training, leaving 35 queries for testing. These 35 training queries were used to learn the system parameters at each power/performance operating state, as discussed in Section 3. We create two hardware abstraction models (hash and sort-merge join) per training query, where the system parameters (such as C_{cpu}, C_{mm}, etc.) are the remaining unknowns and are solved using linear regression.

In the testing phase, given the estimated query parameters for each query (having applied the methods from Section 4.2.1), and the trained hardware abstraction models, we estimate the energy cost at a particular system operating state. This estimate is then compared with the actual measured system energy consumption.

In both validations presented next, we will present error ratio, which is defined as the absolute difference between the measured (actual) value and the estimated value divided by the measured



(a) (b)

Figure 5: (a) average and (b) peak error ratios of the hardware abstraction models. The error ratios were calculated from 70 estimates made by the models at each power/performance system states shown in Table 3, and the two join algorithms.

value. We present this error ratio as a percentage value.

4.3 Validation Results

In this section we discuss the results of the validation process. Table 4 shows the error ratios in the estimates made by the operator model for the query parameters. As can be seen, our operator model estimates the parameters accurately. The average error ratio across all the parameter is less than 8.2% and the peak error ratio is less than 14%.

Figure 5 shows the average and peak error for the four hardware abstraction models proposed in Section 3.2 at the four power/ performance system states. The error rates were calculated from 70 estimates (35 queries X 2 algorithms) made by the cost models at each system state. As can be seen the error ratios (both average and peak) improve as the model complexity increases. Model 4 (varying CPU, memory, and I/O power draw), which is the most completes model we present in this paper, has the lowest average error ratio of 2.9% and the lowest peak error ratio of 8.1%.

4.4 End-to-End Results: ERP Effectiveness

In this section we present our end-to-end results using the techniques that we have proposed in this paper. We use the four system settings (shown in Table 3), and used a large number of join queries (going beyond the original 70 queries that we use above). For each query we generated the ERP using the methods proposed in this paper. We then found that there are four important classes of queries, which vary in how much they can benefit from the energy consumption and response time tradeoffs that is characterized by the ERP. In the interest of space, we only present only one representative query from each class here, see Table 5.

First, let us try to determine when switching to a lower power/ performance state would have little effect on the response time. Lets consider system setting A and C (see Table 3). Since the only difference between these settings is in the amount of available memory, we expect that a query whose peak main memory requirement is less than 2 GB would take approximately the same amount of time to execute, and hence would provide significant energy savings. Figure 6 (a) shows the ERP of such a single join query. The actual query is shown as query class (a) in Table 5. As we can see, in Figure 6 (a), system setting B is the most energy efficient state and *achieves close to 20% energy savings with negligible (1%) response time degradation.*

Now, if we consider system setting A and B, the only difference between them is in the FSB frequency, which affects the operational power and performance of the processor and main memory, but does not affect the performance of the disk. Therefore an I/O-intensive query with minimal computation could exploit this setting to achieve significant energy savings with minimal throughput

Class	Predicate	Size of R, S
a	`R.unique2 < 0.1*||R|| AND R.unique1 = S.unique2`	1GB, 1GB
b	`R.unique1 = S.unique2 AND R.onePercent = S.onePercent`	5GB, 5GB
c	`R.unique2 = S.unique2`	5GB, 5GB
d	`R.unique2 < 0.001*||R|| AND R.onePercent = S.onePercent`	1GB, 1GB

Table 5: All queries have the template (SELECT * FROM R, S WHERE <predicate>). The predicate portion is shown above. These queries are used for the ERP plotted in Figure 6. All relations are modified (100 byte tuples) Wisconsin Benchmark relations.

degradation. Figure 6 (b) gives the ERP of a single join query, Query (b) in Table 5, that satisfies this criteria. We are able to achieve *a 23% energy savings for this query with a 5% degradation in response time*, by operating at system setting B.

Similarly, system setting D would be best for a query that is I/O-intensive, requires minimal computation, and has a low peak memory requirement. One such query common in many benchmarks is the equijoin of two tables that are already sorted on their join attributes. Figure 6 (c) shows the ERP for a query that fits this criteria. This ERP is for query (c) in Table 5.In order to execute this query we create a clustered index on $unique2$ and used it as the join predicate. In this case, merge join (no sorting neccessary) is the overwhelming winner with respect to both response time and energy, which is why we suppress the hash join query plan for this query in Figure 6 (c). (The hash join plans are far outside the plot area that is shown.) In this case we are able to achieve nearly *16% energy savings with negligible (<1%) response time degradation* by switching to system setting D.

Now consider a query that has low I/O requirements, but has a high peak memory requirement. Such a query is not well suited for running on the low power states and would provide minimal energy savings, if any. Figure 6 (d) shows the ERP for such a query – Query (d) in Table 5. In this case, switching to a lower power system setting results in minimal energy savings, and significant response time degradation. For such queries, the DBMS should use the stock settings (A). In Figure 6 (d), we omit the merge join results because the have a significantly larger response time that is well beyond the plot area.

4.4.1 Summary

Figure 7 summarizes the discussion above. When the size of the input relations is large, we expect the query to be I/O-intensive, so reducing the FSB frequency makes sense. If the peak memory requirement is determined to be low, then reducing the memory size can provide significant energy savings with minimal response time degradation. Heuristics such as these can be used to prune the "energy-enhanced" plan space (noted $O(|H| \times |P|)$ in Section 3.1) by reducing $|H|$, the number of system states.

5. RELATED WORK

The area of energy management in the data management community has begun to grow [25, 26, 29]. Much of the drive has come because of the observation that the energy costs of their monthly total cost of ownership (TCO) of large server clusters have begun to grow faster than the hardware costs [28, 35]. Infrastructure-level energy management methods include efficient power distribution and conversion as well as unorthodox methods to cool and direct

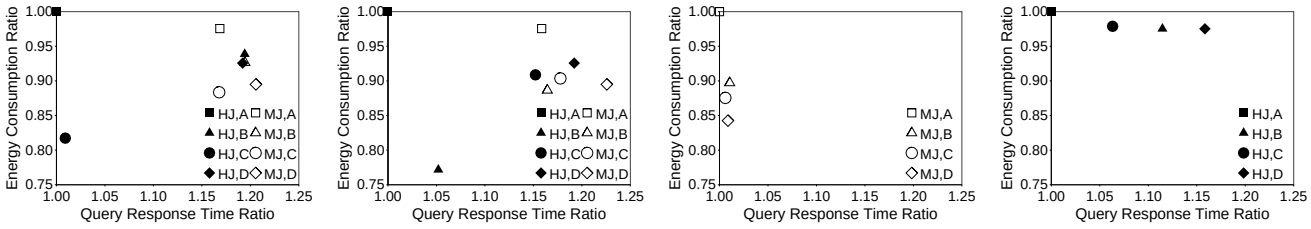


Figure 6: ERPs of four equijoin query classes: (a) Low memory requirement (b) I/O heavy (c) Low memory and I/O heavy (d) High memory and low I/O. Two join algorithms are used: hash join (HJ) and sort-merge join (MJ); along with 'stock' (A), low frequency (B), low memory (C), and low frequency and memory (D) system settings

airflow [22, 24, 25, 31, 33, 36, 41]. While these are certainly effective means of controlling growing energy costs, methods at the node level and cluster level (multiple nodes) must also be explored to achieve true energy efficiency.

There are many data center-wide methods for energy management. One popular method is to consolidate services into as few nodes as possible using virtual machines (VMs) [7]. Consolidation allows cluster nodes to be highly utilized and thus, as energy-efficient as possible [11, 17, 34, 37, 46, 47]. However, numerous interesting issues abound from using VMs for energy management such as performance degradation, storage overhead, and VM migration costs.

Cluster level methods can result in improvements in the energy efficiency of data centers and can largely be used orthogonally to "local-level" methods for reducing energy consumption. Local level techniques improve the energy efficiency of individual nodes and our work falls into this category.

Besides our methods to optimize the efficiency of a DBMS, other local-level techniques have been explored. Other work that directly target DBMS energy efficency include energy efficient execution engines [27]. While that work increases the energy efficiency of the DBMS, making changes to the execution engine is a complex and platform specific task that is not portable. We wish to develop a simple, readily deployable, and portable energy management solution.

The problem of modifying query optimizers to optimize for energy has largely been ignored by the database community. Alonso et al. [10] discuss a simple energy model which can be used to find the most energy efficient query plan to execute the query. However, their methods (studied over a decade ago) do not consider fluctuating power draw from the system. We have shown that such an assumption incurs high error. In this paper, we empower the query optimizer with a energy cost model so that it can choose not only the most energy-efficient query plan but also the best power/ performance system setting to execute the query.

Other local level methods have shown to achieve energy savings by controlling the power states of system components dynamically through software. Most of these efforts focus on achieving energy savings by switching memory modules [18, 20] and/or CPU [21, 32] into lower power states during periods of low workload. Other studies on components such as disk have also been done [4, 16, 42].

CPUs are the most energy efficient components in the system today [13]. Recently, a study on CPU power control and potential energy/ performance tradeoffs was presented [29]. While that work is similar to this study, its narrow scope on CPU energy efficiency leaves many open questions as to how much energy savings can we gain at the system level. Our work clearly answers this question and also provides a new framework for optimizing query for both energy and response time.

Beyond component level energy optimization, there is OS level

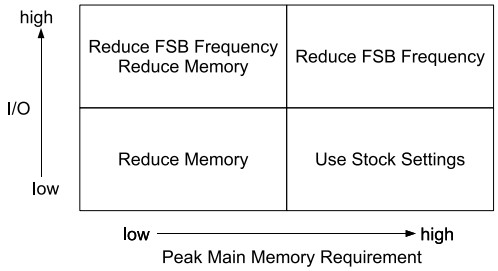


Figure 7: Summary of our analytical results: depending on I/O and memory requirements, certain system settings can be expected to provide favorable energy savings and minimal performance penalties

energy optimization in the form of the Tickless Kernel Project aims to increase idle efficiency [45]. Efforts to develop energy efficiency metrics have also been presented [5, 39] and the Joulesort benchmark reports the energy consumed during sort operations [38].

6. CONCLUSIONS AND FUTURE WORK

This paper presents a new framework for energy-aware database query processing. The framework augments query plans produced by traditional query optimizers with an energy consumption prediction, to produce an Energy Response Time Profile (ERP) for a query. These ERPs can then be used by the DBMS in various interesting ways, including finding the most energy-efficient query plan that meets certain performance constraints (dictated by SLAs). To enable the above framework, a DBMS needs an energy consumption model for queries, and we have developed a simple, portable, practical and accurate model for an important subset of database operations and algorithms. We have used our framework to augment an actual commercial DBMS and using actual energy measurements (energy drawn by the entire system from the wall socket), we have demonstrated that significant energy savings are possible using this framework.

This area of energy-aware data processing is in its inception, and there are many directions for future work. Some of these directions include extending our framework to more query operations/algorithms, considering more complex queries, considering breaking down "complex" operations such like hash join into smaller components (the build and the probe phase) and exploring if switching between hardware power/peformance states results in any benefits, considering multiple concurrent queries and dynamic switching of hardware power/performance states, designing new DBMS techniques to deal with new and evolving power-related hardware mechanisms, and working and influencing the development of new hardware features (e.g. memory) to make it more amenable for

DBMS to exploit for energy-aware query processing. We echo the sentiment expressed in [26] that the database community should embrace this new area of research, and reexamine every aspect of database management for energy-aware data processing.

7. REFERENCES

[1] DRAMSim: A Detailed Memory-System Simulation Framework. http://www.ece.umd.edu/dramsim/.

[2] RDRAM Memory Architecture. http://www.rambus.com/us/products/rdram/.

[3] Seagate Introduces Constellation: All-Star Enterprise Hard Drives With The World's Highest Capacity And Power Efficiency. http://seagate.com/ww/v/index.jsp?locale=en-US&name=null&vgnextoid=c7712f655373f110VgnVCM100000f5ee0a0aRCRD.

[4] SNIA Green Storage Initiative. http://www.snia.org/forums/green.

[5] SPEC Power. http://www.spec.org/power_ssj2008.

[6] The SimpleScalar-Arm Power Modeling Project. http://www.eecs.umich.edu/~panalyzer/.

[7] VMware Infrastructure Architecture Overview White Paper. http://www.vmware.com/pdf/vi_architecture_wp.pdf.

[8] Report To Congress on Server and Data Center Energy Efficiency. In *U.S. EPA Technical Report*, 2007.

[9] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi. Multicore dimm: an energy efficient memory module with independently controlled drams. *IEEE Comput. Archit. Lett.*, 8(1):5–8, 2009.

[10] R. Alonso and S. Ganguly. Energy efficient query optimization. Technical report, Matsushita Info Tech Lab, 1992.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.

[12] L. A. Barroso and U. Holzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 2007.

[13] L. A. Barroso and U. Holzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2009.

[14] C. Belady. In the Data Center, Power and Cooling Costs More than the IT Equipment it Supports. *Electronics Cooling*, 23(1), 2007.

[15] K. G. Brill. Data Center Energy Efficiency and Productivity. In *The Uptime Institute - White Paper*, 2007.

[16] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving Disk Energy in Network Servers. In *17th Annual International Conference on Supercomputing*, 2003.

[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Symposium on Networked Systems Design and Implementation*, 2005.

[18] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based dram energy management. pages 697–702, 2002.

[19] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[20] X. Fan, C. Ellis, and A. Loebeck. Memory controller policies for dram power management. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 129–134, New York, NY, USA, 2001. ACM.

[21] X. Fan, C. S. Ellis, and A. R. Lebeck. The synergy between power-aware memory systems and processor voltage scaling. In *In Workshop on Power-Aware Computing Systems*, pages 164–179, 2003.

[22] Green Grid. Seven Strategies to Improve Datacenter Cooling Efficiency. http://www.thegreengrid.org/gg_content.

[23] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Jane, I. N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John. Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach. pages 141–150, 2001.

[24] J. Hamilton. Where Does Power Go In DCs and How To Get It Back? http://mvdirona.com/jrh/TalksAndPapers/JamesRH_DCPowerSavingsFooCamp08.ppt, 2008.

[25] J. Hamilton. Cooperative Expendable Micro-slice Servers (CEMS): Low Cost, Low Power Servers for Internet-Scale Services. In *CIDR*, 2009.

[26] S. Harizopoulos, M. A. Shah, J. Meza, and P. Ranganathan. Energy Efficiency: The New Holy Grail of Database Management Systems Research. In *CIDR*, 2009.

[27] H. Hopfner and C. Bunse. Towards an energy aware dbms - energy consumptions of sorting and join algorithms. In *Grundlagen von Datenbanken*, volume CS-02-09, pages 69–73, 2009.

[28] J. G. Koomey. Estimating Total Power Consumption by Servers in the US and the World. http://enterprise.amd.com/Downloads/svrpwrusecompletefinal.pdf, 2007.

[29] W. Lang and J. M. Patel. Towards Eco-friendly Database Management Systems. In *CIDR*, 2009.

[30] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *In Architectural Support for Programming Languages and Operating Systems*, pages 105–116, 2000.

[31] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. *International Symposium on Computer Architecture*, 2008.

[32] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–61, New York, NY, USA, 2001. ACM.

[33] D. Nelson, M. Ryan, S. DeVito, K. V. Ramesh, P. Vlasaty, B. Rucker, and B. Nelson. The Role of Modularity in Datacenter Design. http://www.sun.com/storagetek/docs/EED.pdf.

[34] A. Noll, A. Gal, and M. Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. In *Workshop on Cell Systems and Applications*, 2008.

[35] C. D. Patel and A. J. Shah. Cost Model for Planning, Development and Operation of a Datacenter. http://www.hpl.hp.com/techreports/2005/HPL-2005-107R1.pdf.

[36] PG&E. High Performance Datacenters. http://hightech.lbl.gov/documents/DATA_CENTERS/06_DataCenters-PGE.pdf.

[37] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *ISCA*, 2006.

[38] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: a balanced energy-efficiency benchmark. In *SIGMOD*, 2007.

[39] S. Rivoire, M. A. Shah, P. Ranganathan, C. Kozyrakis, and J. Meza. Models and Metrics to Enable Energy-Efficiency Optimizations. *Computer*, 2007.

[40] S. Ryffel, T. Stathopoulos, D. McIntire, W. Kaiser, and L. Thiele. Accurate Energy Attribution and Accounting for Multi-core Systems. 2009.

[41] S Greenberg and E Mills and B Tschudi. Best Practices for Datacenters: Lessons Learned from Benchmarking 22 Datacenters. http://eetd.lbl.gov/EA/mills/emills/PUBS/PDF/ACEEE-datacenters.pdf, 2006.

[42] S. Sankar, S. Gurumurthi, and M. R. Stan. Intra-disk Parallelism: An Idea Whose Time Has Come. In *ACM International Symposium on Computer Architecture*, 2008.

[43] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM.

[44] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[45] S. Siddha, V. Pallipadi, and A. V. D. Ven. Getting Maximum Mileage Out of Tickless. In *Linux Symposium*, 2007.

[46] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering Energy Proportionality with Non Energy-Proportional Systems - Optimizing the Ensemble. In *HotPower*, 2008.

[47] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, 2002.