Energy Management Schemes for Memory-Resident Database Systems

Jayaprakash Pisharath, Alok Choudhary
Electrical & Computer Engineering Department
Northwestern University
Evanston, IL 60208
{jay, choudhar}@ece.northwestern.edu

Mahmut Kandemir
Department of Comp. Sci. & Engineering
Pennsylvania State University
University Park, PA 16802
kandemir@cse.psu.edu

ABSTRACT

With the tremendous growth of system memories, memory-resident databases are increasingly becoming important in various domains. Newer memories provide a structured way of storing data in multiple chips, with each chip having a bank of memory modules. Current memory-resident databases are yet to take full advantage of the banked storage system, which offers a lot of room for performance and energy optimizations. In this paper, we identify the implications of a banked memory environment in supporting memory-resident databases, and propose hardware (memory-directed) and software (query-directed) schemes to reduce the energy consumption of queries executed on these databases. Our results show that high-level query-directed schemes (hosted in the query optimizer) better utilize the low-power modes in reducing the energy consumption than the respective hardware schemes (hosted in the memory controller), due to their complete knowledge of query access patterns. We extend this further and propose a query restructuring scheme and a multi-query optimization. Queries are restructured and regrouped based on their table access patterns to maximize the likelihood that data accesses are clustered. This helps increase the inter-access idle times of memory modules, which in turn enables a more effective control of their energy behavior. This heuristic is eventually integrated with our hardware optimizations to achieve maximum savings. Our experimental results show that the memory energy reduces by 90% if query restructuring method is applied along with basic energy optimizations over the unoptimized version. The system-wide performance impact of each scheme is also studied simultaneously.

Categories and Subject Descriptors:
H.2.2 [Database Management]:Physical Design – *Access Methods*
H.3.2 [Information Storage and Retrieval]: Information Storage
B.3.1 [Memory Structures]: Semiconductor Memories – *DRAM*

General Terms: Design, Performance

Keywords: database, DRAM, energy, hardware energy scheme, power consumption, query-directed energy management, multi-query optimization

1. INTRODUCTION

Memory-resident databases (also called in-memory databases [6]) are emerging to be more significant due to the current era of memory-intensive computing. These databases are used in a wide range of systems ranging from real-time trading applications to IP routing. With the growing complexities of embedded systems (like real-time constraints), use of a commercially developed structured memory database is becoming very critical [5]. Consequently, device developers are turning to commercial databases, but existing embedded DBMS software has not provided the ideal fit. Embedded databases emerged well over a decade ago to support business systems, with features including complex caching logic and abnormal termination recovery. But on a device, within a set-top box or next-generation fax machine, for example, these abilities are often unnecessary and cause the application to exceed available memory and CPU resources. In addition, current in-memory database support does not consider embedded system specific issues such as energy consumption.

Memory technology has grown tremendously over the years, providing larger data storage space at a cheaper cost. Recent memory designs have more structured and partitioned layouts in the form of multiple chips, each having *memory banks* [30]. Banked memories are energy efficient by design, as per-access energy consumption decreases with decreasing memory size (and a memory bank is typically much smaller compared to a large monolithic memory). In addition, these memory systems provide *low-power operating modes,* which can be used for reducing the energy consumption of a bank when it is not being used. An important question regarding the use of these low-power modes is when to transition to one once an idleness is detected. Another important question is whether the application can be modified to take better advantage of these low-power modes. While these questions are slowly being addressed in architecture, compiler, and OS communities, to our knowledge, there has been no prior work that examines the energy and performance behavior of databases under a banked memory architecture. Considering increasingly widespread use of banked memories, such a study can provide us with valuable information regarding the behavior of databases under these memories and potential modifications to DBMSs for energy efficiency. Since such banked systems are also being employed in high-end server systems, banked memory friendly database strategies can also be useful in high-end environments to help reduce energy consumption.

Our detailed energy characterization of a banked memory architecture that runs a memory-resident DBMS showed that nearly 59% of overall energy (excluding input/output devices) in a typical query execution is spent in the memory, making this component an important target for optimization (see Figure 1). Moreover, for any system, memory power and energy consumption have become critical design parameters besides cost and performance. Based on these observations, this paper evaluates the potential energy benefits that memory-resident database queries can achieve by making use of banked memory architectures supported with low-power operating modes. Since each memory bank is capable of operating independently, this opens up abundant avenues for energy and performance optimizations.

In this paper, we focus on a banked memory architecture and study potential energy benefits when database queries are executed. Specifically, we focus on two important aspects of the problem:

• *Characterizing energy benefits of banked memories using hardware and software techniques:* To see whether query execution can make use of available low-power modes, we study both hardware and software techniques. The hardware techniques detect the idleness of memory banks and switch the inactive (idle) banks (during

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'04, November 8–13, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-874-1/04/0011 ...$5.00.

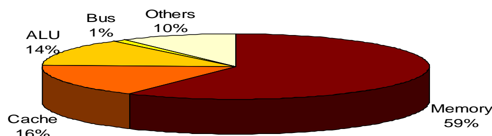

Figure 1: Breakup of the energy consumption for various system components. The results are based on the average energy consumption of TPC-H benchmarks [35] executed on a memory-resident DBMS.

query execution) to low-power operating modes. We also present a query-based memory energy optimization strategy, wherein the query plan is augmented by explicit bank turn-off/on instructions that transition memory banks into appropriate operating modes during the course of execution based on the query access pattern. We experimentally evaluate all the proposed schemes and obtain energy consumptions using an energy simulator. Our experiments using TPC-H queries [35] and a set of queries suitable for hand-held devices clearly indicate that both hardware-based and query-directed strategies save significant memory energy.

• *Query restructuring for memory energy savings:* We propose a query restructuring scheme and a multi-query optimization strategy to further increase energy benefits coming from using low-power operating modes. The idea behind these schemes is to increase bank inter-access times so that more aggressive low-power modes can be employed and a memory bank can stay in a low-power mode longer once it is transitioned. Our experimental evaluation indicates that this query restructuring strategy does not only reduce energy consumption, but also helps improve overall performance (execution cycles).

Apart from providing useful input for database designers, our results can also be used by hardware designers to tune the behavior of low-power modes so that they handle query access patterns better. Similar to the observation that creating a lightweight version of a disk-based database will not serve as a suitable in-memory database, our belief is that taking an in-memory database system and using it on a banked architecture without any modification may not generate the desired results. Therefore, the results presented in this work also shed light on how database design and memory architecture design interact with each other.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 elaborates on the memory database that we built and also on the memory banking scheme that we employ for our experiments. Section 4 presents in detail the proposed hardware and query-directed energy optimization techniques. The results of our energy evaluation of these schemes are discussed in Section 5. Our experiments also account for the performance overhead incurred in supporting our schemes. Section 6 presents our query restructuring and regrouping scheme, and Section 7 discusses its energy/performance benefits within the context of our banked memory architecture. Finally, Section 8 summarizes the results.

2. RELATED WORK

In the past, memory has been redesigned, tuned or optimized to suit emerging fields. Need for customized memory structures and allocation strategies form the foundation for such studies. Copeland et al proposed SafeRAM [11], a modified DRAM model for safely supporting memory-resident databases alike disk-based systems, and for achieving good performance. In PicoDBMS [27], Pucheral et al present techniques for scaling down a database to a smart card. This work also investigates some of the constraints involved in mapping a database to an embedded system, especially memory constraints and the need for a structured data layout. Anciaux et al [3] explicitly model the lower bound of the memory space that is needed for query execution. Their work focuses on light weight devices like personal organizers, sensor networks, and mobile computers. Boncz et al show how memory accesses form a major bottleneck during database accesses [7]. In their work, they also suggest a few remedies to alleviate the memory bottleneck.

An et al analyze the energy behavior of mobile devices when spatial access methods are used for retrieving memory-resident data [2]. They use a cycle accurate simulator to identify the pros and cons of various indexing schemes. In [1], Alonso et al investigate the possibility of increasing the effective battery life of mobile computers by selecting energy efficient query plans through the optimizer. Although the ultimate goal seems the same, their cost plan and the optimization criterion are entirely different from our scheme. Specifically, their emphasis is on a client-server model optimizing the network throughput and overall energy consumption. Gruenwald et al propose an energy-efficient transaction management system for real-time mobile databases in ad-hoc networks [16]. They consider an environment of mobile hosts. In [22], Madden et al propose TinyDB, an acquisitional query processor for sensor networks. They provide SQL-like extensions to sensor networks, and also propose acquisitional techniques that reduce the power consumption of these networks. It should be noted that the queries in such a mobile ad-hoc network or a sensor environment is different from those in a typical DBMS. This has been shown by Imielinksi et al in [19]. In our model, we base our techniques on a generic banked memory environment and support complex, memory-intensive typical database operations. There are more opportunities for energy optimizations in generic memory databases, which have not yet been studied completely. The approach proposed in this paper is different from prior energy-aware database related studies, as we focus on a banked memory architecture, and use low-power operating modes to save energy.

Gassner et al review some of the key query optimization techniques required by industrial-strength commercial query optimizers, using the DB2 family of relational database products as examples [15]. This paper provides insight into design of query cost plans and optimization using various approaches. In [23], Manegold studies the performance bottlenecks at the memory hierarchy level and proposes a detailed cost plan for memory-resident databases. Our cost plan and optimizer mimics the PostgreSQL model [12, 14]. We chose it due to its simple cost models and open source availability.

A query restructuring algorithm is proposed by Hellerstein in [18]. This algorithm uses predicate migration to optimize expensive data retrievals. In [10], Chaudhuri et al extend this approach to study user-defined predicates and also guarantee an optimal solution for the migration process. Sarawagi et al present a query restructuring algorithm that reduces the access times of data retrieval from tertiary databases [32]. Monma et al develop the series-parallel algorithm for reordering primitive database operations [24]. This algorithm optimizes an arbitrarily constrained stream of primitive operations by isolating independent modules. This work forms the basic motivation for our query restructuring algorithm. However, our paper is different from all of the above work in the sense that we reorder queries for reducing energy consumption. Moreover, our database is memory-resident, with the presence of banked memory that gives more freedom for optimizations.

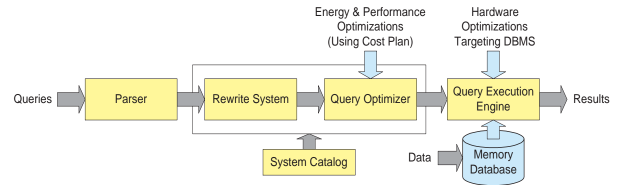

Figure 2: DBMS architecture.

3. SYSTEM ARCHITECTURE

3.1 DBMS

For our work, we modified the PostgreSQL DMBS to work with memory-resident data sets as its workload. The block diagram for our setup is shown in Figure 2. The core components are derived from PostgreSQL. The flow of our model is similar to PostgreSQL except that the database is memory resident. A query is parsed for syntax and then sent to the rewrite system. The rewrite system uses the system catalog to generate the query tree, which is then sent to the optimizer. The query optimizer derives the cost of the query in

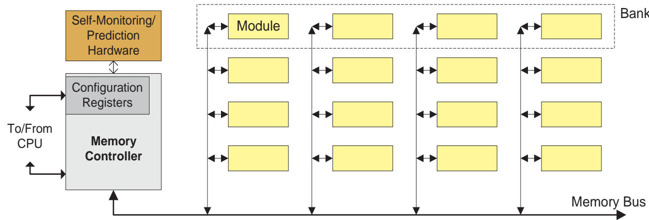

Figure 3: Banked memory architecture.

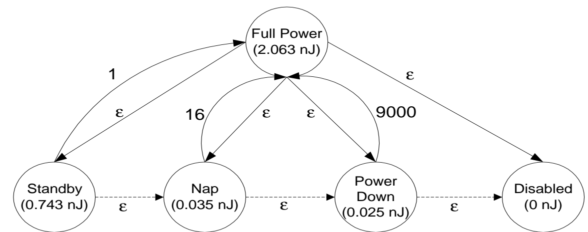

Figure 4: Available operating modes and their resynchronization costs.

multiple ways using the query tree and issues the best suited plan to the query execution engine. We incorporate our software-based techniques at the optimizer stage of the DBMS. These optimizations are based on the cost that is derived for each of the query plans (the discussion pertaining to the modified cost model is deferred till Section 4). Based on the final query execution plan, the execution engine executes the query by using the database. The database is entirely memory resident and the memory is organized in a banked format (elaborated in the following section). The executor recursively iterates the query plan and uses a per-tuple based strategy (pipelined execution, and not bulk processing) to project the output results. The proposed hardware optimizations are at the computer architecture level of the system. Since the base DBMS model is similar to PostgreSQL, we do not elaborate each component in detail ([26] provides an elaborate discussion). Instead, we highlight our contributions, and modifications to DBMS (shown in blue in Figure 2) in the following sections. Overall, our strategies require modification to the query optimizer, memory hardware, and system software components.

3.2 Memory Model

We use a memory system that contains a memory array organized as banks (rows) and modules (columns), as is shown pictorially in Figure 3 for a 4×4 memory module array. Such banked systems are already being used in high-end server systems [30] as well as low-end embedded systems [31]. The proposed optimizations will, however, apply to most bank-organized memory systems. Accessing a word of data would require activating the corresponding modules of the shown architecture. Such an organization allows one to put the unused banks into a low-power operating mode. To keep the issue tractable, this paper bases the experimental results on a sequential database environment and does not consider a multiprocessing environment (like transaction processing which requires highly complex properties to be satisfied). We assume in our experiments that there is just one module in a bank; hence, in the rest of our discussion, we use the terms "bank" and "module" interchangeably.

3.3 Operating Modes

We assume the existence of five operating modes for a memory module: *active, standby, nap, power-down,* and *disabled*[1]. Each mode is characterized by its *energy consumption* and the time that it takes to transition back to the active mode (termed *resynchronization time* or *resynchronization cost*). Typically, the lower the energy consumption, the higher the resynchronization time [30]. Figure 4 shows possible transitions between the various low-power modes (the dynamic energy[2] consumed in a cycle is given for each node) in our model. The resynchronization times in cycles (based on a cycle time of 3.3ns) are shown along the arrows (we assume a negligible cost ε for transitioning to a lower power mode). The energy and resynchronization values shown in this figure have been obtained from the RDRAM memory data sheet (512MB, 2.5V, 3.3ns cycle time, 8MB modules) [30]. When a module in standby, nap, or power-down mode is requested to perform a memory transaction, it first goes to the active mode, and then performs the requested transaction. While one could employ all possible transitions given in Figure 4 (and maybe more), our query-directed approach only utilizes the transitions shown by solid arrows. The runtime (hardware-based) approaches, on the other hand, can exploit two additional transitions: from standby to nap, and from nap to power-down.

[1]Current DRAMs [30] support up to six energy modes of operation with a few of them supporting only two modes. One may choose to vary the number of modes based on the target memory.

[2]We exclusively concentrate on dynamic power consumption that arises due to bit switching, and do not consider the static (leakage) power consumption [28] in this paper.

3.4 System Support for Power Mode Setting

Typically, several of the memory modules (that are shown in Figure 3) are controlled by a memory controller which interfaces with the memory bus. For example, the operating mode setting could be done by programming a specific control register in each memory module (as in RDRAM [30]). Next is the issue of how the memory controller can be told to transition the operating modes of the individual modules. This is explored in two ways in this paper: *hardware-directed approach* and *software-directed* (*query-directed*) *approach.*

In the hardware-directed approach, there is a `Self-Monitoring and Prediction Hardware` block (as shown in Figure 3), which monitors all ongoing memory transactions. It contains some prediction hardware (based on the hardware scheme) to estimate the time until the next access to a memory bank and circuitry to ask the memory controller to initiate mode transitions (limited amount of such self-monitored power down is already present in current memory controllers, for example: Intel 82443BX and Intel 820 Chip Sets).

In the query-directed approach, the DBMS explicitly requests the memory controller to issue the control signals for a specific module's mode transitions. We assume the availability of a set of `configuration registers` in the memory controller (see Figure 3) that are mapped into the address space of the CPU (similar to the registers in the memory controller in [20]). These registers are then made available to the user space (so that the DBMS application can have a control) through operating system calls.

Regardless of which strategy is used, the main objective of employing such strategies is to reduce the energy consumption of a query when some memory banks are idle during the query's execution. That is, a typical query only accesses a small set of tables, which corresponds to a small number of banks. The remaining memory banks can be placed into a low-power operating mode to save memory energy. However, it is also important to select the low-power mode to use carefully (when a bank idleness is detected), as switching to a wrong mode either incurs significant performance penalties (due to large resynchronization costs) or prevents us from obtaining maximum potential energy benefits.

Note that energy optimization is our context can be performed from two angles. First, suitable use of low-power operating modes can reduce energy consumption of a given query execution. Second, the query plan can be changed (if it is possible to do so) to further increase energy benefits. In this work, we explore both these angles.

4. POWER MANAGEMENT SCHEMES

In a banked architecture, the memory can be managed through either of the following two approaches: (1) a runtime approach wherein the hardware is in full control of operating mode transitions; and (2) a query-directed scheme wherein explicit bank turn-on/off instructions are inserted in the query execution plan to invoke mode transitions. One also has the option of using both the approaches simultaneously (which we illustrate in later sections).

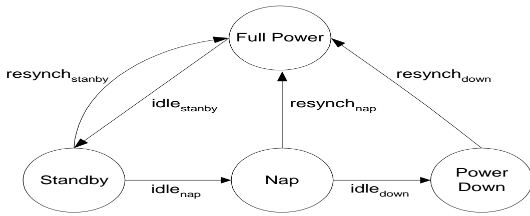

Figure 5: Dynamic threshold scheme.

4.1 Hardware-Directed Schemes

We explore two hardware-directed approaches that allow the memory system to automatically transition the idle banks to an energy conserving state. The problem then is to detect/predict bank idleness and transition idle banks into appropriate low-power modes.

4.1.1 Static Standby Scheme

The first approach is a per-access optimization. Most of the recent DRAMs allow the chips to be put to standby mode immediately after each reference [30]. After a read/write access, the memory module that gets accessed can be placed into the standby mode in the following cycle. We refer to this scheme as the static standby mode in the rest of our discussion. Note that, while this scheme is not very difficult to implement, it may lead to frequent resynchronizations, which can be very harmful as far as execution cycles are concerned.

4.1.2 Dynamic Threshold Scheme

Our second hardware-guided approach is based on runtime dynamics of the memory subsystem. The rationale behind this approach is that if a memory module has not been accessed in a while, then it is not likely to be needed in the near future (that is, inter-access times are predicted to be long). A threshold is used to determine the idleness of a module after which it is transitioned to a low-power mode. More specifically, we propose a scheme where each memory module is put into a low-power state with its idle cycles as the threshold for transition.

The schematic of our dynamic threshold scheme is depicted in Figure 5. After $idle_{stndby}$ cycles of idleness, the corresponding module is put in the standby mode. Subsequently, if the module is not referenced for another $idle_{nap}$ cycles, it is transitioned to the nap mode. Finally, if the module is not referenced for a further $idle_{down}$ cycles, it is placed into the power-down mode. Whenever the module is referenced, it is brought back into the active mode incurring the corresponding resynchronization costs (based on what low-power mode it was in). It should be noted that even if a single bank experiences a resynchronization cost, the other banks will also incur the corresponding delay (to ensure correct execution). Implementing the dynamic mechanism requires a set of counters (one for each bank) that are decremented at each cycle, and set to a threshold value whenever they expire or the module is accessed. A zero detector for a counter initiates the memory controller to transmit the instructions for mode transition to the memory modules.

4.2 Software-Directed Scheme

It is to be noted that a hardware-directed scheme works well independent of the DBMS and the query optimizer used. This is because the idleness predictors are attached to the memory banks and monitor idleness from the perspective of banks. In contrast, a query-directed scheme gives the task of enforcing mode transitions to the query. This is possible because the query optimizer, once it generates the execution plan, has a complete information about the query access patterns (i.e., which tables will be accessed and in what order, etc). Consequently, if the optimizer also knows the *table-to-bank mappings,* it can have a very good idea about the bank access patterns. Then, using this information, it can proactively transition memory banks to different modes. In this section, we elaborate on each step in the particular query-directed approach that we implemented, which includes customized bank allocation, query analysis, and insertion of bank turn-on/off (for explicit power mode control) instructions.

4.2.1 Bank Allocation

In the case of software-directed scheme, the table allocation is handled by the DBMS. Specifically, the DBMS allocates the newly-created tables to the banks, and keeps track of the table-to-bank mappings. When a "create table" operation is issued, the DBMS first checks for free space. If there is sufficient free space available in a single bank, the table is allocated from that bank. If a bank is not able to accommodate the entire table, the table is split across multiple banks. Also, while creating a new table, the DBMS tries to reuse the already occupied banks to the highest extent possible; that is, it does not activate a new bank unless it is necessary. Note that the unactivated (unused) banks – i.e., the banks that do not hold any data – can remain in the disabled mode throughout the execution. However, it also tries not to split tables excessively. In more detail, when it considers an already occupied bank for a new table allocation, the table boundaries are checked first using the available space in that bank. If a bank is more than two-thirds full with the table data, the rest of the bank is padded with empty bits and the new table is created using pages from a new bank. Otherwise, the table is created beginning in the same bank. Irrespective of whether the table is created on a new bank or not, the DBMS creates a new table-to-bank mapping entry after each table creation.

In hardware-directed schemes, we avoid these complexities involved in bank allocation as we assume that there is absolutely no software control. Consequently, in the hardware-directed schemes, we use the *sequential first touch placement policy*. This policy allocates new pages sequentially in a single bank until it gets completely filled, before moving on to the next bank. Also, the table-to-bank mapping is not stored within the DBMS since the mode control mechanism is handled by the hardware.

4.2.2 Estimating Idleness and Selecting the Appropriate Low-Power Mode

It should be emphasized that the main objective of our query-directed scheme is to identify bank idleness. As explained above, in order to achieve this, it needs table-to-bank mapping. However, this is not sufficient as it also needs to know when each table will be accessed and how long an access will take (i.e., the query access pattern). To estimate this, we need to estimate the duration of accesses to each table, which means estimating the time taken by the database operations. Fortunately, the current DBMSs already maintain such estimates for query optimization purposes [12, 15, 29, 33, 34]. More specifically, given a query, the optimizer looks at the query access pattern using the generated query plan. The inter-access times are calculated using the query plan. A query plan elucidates the operations within a query and also the order in which these operations access the various tables in the database. Even in current databases, the query plan generator estimates access costs using query plans [12]. We use the same access cost estimation methodology. These access costs are measured in terms of page (block) fetches. In our memory-resident database case, a page is basically the block that is brought from memory to the cache. For instance, the cost of sequential scan is defined as follows (taken from [12]):

$$Cost_{seq_scan} = N_{blocks} + CPU * N_{tuples}$$

Here, N_{blocks} is the number of data blocks retrieved, N_{tuples} is the number of output tuples, and CPU is the fudge factor that adjusts the system tuple-read speed with the actual memory hierarchy data-retrieval speed. Usually, optimizers use the above cost metric to choose between multiple query plan options before issuing a query. We attach a cost to each page (block) read/write operation to obtain an estimate of the access cost (time) in terms of execution cycles. For instance, the above scan operation is modified as follows:

$$Cost_{block_fetch} = T \ \ cycles$$
$$Cost_{seq_scan} = N_{blocks} * T + CPU * N_{tuples} * \frac{block}{tuples} * T$$

In these expressions, T is the delay in cycles to fetch a block from the memory. Thus, our cost plan is projected in terms of access cycles. We extend this to other database operations like JOIN and AGGREGATE based on the cost models defined in [14, 12].

Given a query, we break down each operation within the plan (including sub-plans) and estimate the access cost (in cycles) for each

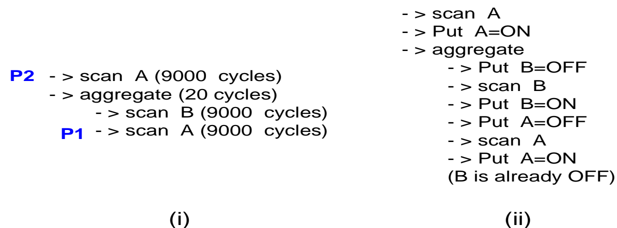

Figure 6: Example application of the query-directed scheme. (i) The original execution plan. (b) The augmented execution plan.

primitive operation. Our objective in estimating the per-operation time in cycles is to eventually identify the inter-access times of operations in the query (and hence, to put the banks that hold unused tables to low-power modes). There are table accesses associated with each operation, and bank inter-access times depend on the table inter-access times. A query has information of the tables that it accesses. Thus, knowing the inter-access time for each operation leads to the inter-access times for each table as well. A table is mapped to certain banks, and the table-to-bank mapping is available in the query optimizer.

Consequently, if the table inter-access time is T, and the resynchronization time is T_p (assuming less than T), then the optimizer can transition the associated modules into a low-power mode (with a unit time energy of E_p) for the initial $T - T_p$ period (which would consume a total $[T-T_p]E_p$ energy), activate the module to bring it back to the active mode at the end of this period following which the module will resynchronize before it is accessed again (consuming T_pE_a energy during the transition assuming that E_a is the unit time energy for active mode as well as during the transition period). As a result, the total energy consumption with this transitioning is $[T-T_p]E_p + T_pE_a$ *without any resynchronization overheads*, while the consumption would have been TE_a if there had been no transitioning (note that this calculation considers only the idle period). The DBMS optimizer evaluates all possible choices (low-power modes) based on corresponding per cycle energy costs and resynchronization times, and table inter-access time to pick up the best choice. Note that the DBMS can select different low power modes for different idle periods of the same module depending on the duration of each idle period. Specifically, we use the most energy saving low-power mode without increasing the original query execution time (i.e., when the original idleness is over, the module should be up in the active mode ready for the operation).

4.2.3 Inserting Bank-On/Off Instructions

The last part of the software-directed scheme is to insert explicit (operating) mode transitioning instructions in the final query execution plan. For this, we introduce place-markers (mapped to system calls) which are interpreted at the low-level (interpreted later by our memory controller, which actually sets the corresponding low-power modes). This is done so that the query execution engine can issue the query without much performance overhead, and with the same transparency.

As an example, consider the following. Let tables A and B each have 1000 records, each record being 64 bytes. Consider the query plan depicted in Figure 6(i), taken from PostgreSQL. The query plan reads from bottom to top (P2 follows P1). A scan of table A is done first, followed by a scan of table B. The result of these operations are then used by an aggregate operation. Another (independent) scan operation on table A follows the aggregate operation. The per step access costs are also shown. From the generated query plan, it is evident that table A is not accessed between point P1 and point P2. Once the results are extracted after the scan at point P1, the banks that hold table A can be put to a low-power mode, and the banks that hold table B can be activated for data extraction. This is illustrated in Figure 6(ii) using place-markers for tables A and B. Banks holding Table A are reactivated at point P2 (banks of Table B remain off).

5. EXPERIMENTAL EVALUATION OF HARDWARE-DIRECTED AND QUERY-DIRECTED SCHEMES

In this section, we study the potential energy benefits of our hardware and software-directed schemes. We first explain the experimental setup that we used in our simulations. Then, the set of queries that we used to study our schemes is introduced. After that, we present energy consumption results. While we discuss the energy benefits of using our schemes, we also elaborate the overheads associated with supporting each of our schemes.

5.1 Setup

5.1.1 Simulation Environment

As mentioned before, the query-directed schemes are implemented in the query optimizer of the memory database model elaborated in Section 3.1. We interface this DBMS to an enhanced version of the SimpleScalar/Arm simulator [4] to form a complete database system. The intermediate interface (invoked by DBMS) provides a set of operating system calls (on Linux kernel 2.4.25), which in turn invokes the SimpleScalar simulator. The SimpleScalar simulator models a modern microprocessor with a five-stage pipeline: fetch, decode, issue, write-back, and commit. We implemented our hardware techniques within the framework of the sim-outorder tool from the SimpleScalar suite, extended with the ARM-ISA support [4]. Specifically, we modeled a processor architecture similar to that of Intel StrongARM SA-1100. The modeled architecture has a 16KB direct-mapped instruction cache and a 8KB direct-mapped data cache (each of 32 byte-length). We also model a 32-entry full associative TLB with a 30-cycle miss latency. The off-chip bus is 32 bit-wide. For estimating the power consumption (and hence, the energy consumption), we use the Wattch simulator from Princeton University [8].

Our banked memory model is based on [13,21], as shown in Figure 4. We use values from Figure 4 for modeling the delay (transition cycles) in activation and resynchronization of various power-states. Our simulations account for all performance and energy overheads incurred by our schemes. In particular, the energy numbers we present include the energy spent in maintaining the idleness predictors (in the hardware-directed scheme) and the energy spent in maintaining the table-to-bank mappings (in the query-directed scheme), and in fetching and executing the bank turn-on/off instructions (in the query-directed scheme). The predictors were implemented using decrementing counters (equal to the number of banks) and zero detector based on the discussion in Section 4.1. The predictors are synchronized with the system cycles to maintain consistency of operation, and to minimize the overheads. The query optimizer maintains the table-bank mappings, which is modeled as an array list for instant access. The bank turn-on/off instructions are executed by setting hardware registers, and hence, these instructions are modeled as register operations using the existing instruction set architecture. We present two important statistics in our experimental results. *Energy consumption* corresponds to the energy consumed in the memory system (including the above mentioned overheads). We also present statistics about the *performance overhead* (i.e., *increase in execution cycles*) for each of our schemes. This overhead includes the cycles spent in resynchronization (penalty cycles are modeled based on values in Figure 4) as well as the cycles spent (in the CPU datapath) in fetching and executing the turn-on/off instructions (in the query-directed scheme).

5.1.2 Queries

To evaluate our scheme for memory-resident databases, we considered two classes of queries. The first class is a subset of queries from the Transaction Processing Council (TPC-H) benchmark [35]. TPC-H involves complex queries with a large amount of data accesses. Operations in decision support benchmarks (TPC-D evolved to TPC-H) have good spatial locality with abundant data intensive operations [9]. This assists us to perform a rigorous test of our schemes. The top part of Table 1 gives details of the TPC-H queries we used and the corresponding database parameters. The selected operations represent a good mix and could be used to build a variety of complicated queries.

Table 1: **The two classes of queries considered for our experiments.**

Source	Query	Description	Tables
TPC-H	Q6	Simple query	PART, CUSTOMER, ORDERS, and LINEITEM tables generated using DBGEN with scale 1.0
	Q3	Complex query involving JOIN	
	Q4	Complex query involving NEST	
	Q17	Complex query involving JOIN and NEST	
Queries targeting a simple organizer	P1	Simple name and address lookup	ADDRESSBOOK populated with 1.3 million entries, 50% subset of FRIENDS and 25% subset of COLLEAGUES
	P2	Lookup in directory of friends	
	P3	Lookup in directory of colleagues and friends	

Memory-resident databases run queries that are different from the typical database queries as seen in TPC-H. The second set of queries that we consider are representative of applications that execute on handheld devices. The typical operations that are performed on an organizer were imitated on our setup (we name the queries P1, P2, P3). The first query involves a simple address lookup using a 'NAME' as input. The SQL for query P1 is shown on the left section of Table 2. Recent organizers [17,25] provide an ordered view of the underlying addressbook database. For instance, organizers provide the creation of folders. A "friends" folder can be a collection of personnel with a tag set as "friend" in the addressbook. We defined folder as a restrained/customized view of the same database (address book). Intuitively, query P2 strives to do a lookup of friends living in a particular city. The "friends" view and hence the query P2 is defined on the right section of Table 2. Query P3 combines views (folders). For this we defined a new folder called "colleagues". P3 aims to find friends and/or colleagues whose names start with an 'a', living in a particular 'CITY'. Since P3 is very similar to P2 with some extra fields, we do not present the SQL for P3. The intermediate tables and results during query execution are also stored in the memory.

Table 2: SQL for organizer queries

Query P1	Query P2	
SELECT a_name, a_address, a_city, a_office_phone, a_home_phone, a_mobile_phone, a_email, a_web, a_specialnotes FROM addressbook WHERE a_name = '[NAME]';	CREATE VIEW friends AS SELECT a_name, a_address, a_city, a_home_phone, a_mobile_phone FROM addressbook WHERE a_tag = '[FRIEND]' GROUP BY a_name;	**P2:** SELECT a_address, a_home_phone, a_mobile_phone FROM friends WHERE a_city = '[CITY]' GROUP BY a_name;

5.1.3 *Default Parameters*

For our experiments, we populate our database using the *DBGEN* software from TPC-H benchmark suite with a scale factor of 1.0. Our organizer database is populated with 1.3 million records.

For dynamic threshold scheme, we use 10, 100 and 10,000 cycles as $idle_{stndby}$, $idle_{nap}$, and $idle_{down}$, respectively. For all schemes, the banks are in power-down mode before their first access. On/Off instructions are inserted based on the inter-access times of table. We use the same cycles as in $idle_{stndby}$, $idle_{nap}$, and $idle_{down}$ for inserting instructions. As an example, consider the inter-access (T) of a table as 25 cycles, which lies between 10 ($idle_{stndby}$) and 100 ($idle_{nap}$) cycles. We insert an On/Off instruction at the beginning of T to put a table to standby mode for 24 cycles, taking into consideration the resynchronization period of 1 cycle as well. Similar technique is applied for inter-access times that fall in between other power modes.

A single page transfer time is needed for access cost calculation in software-directed scheme. We derive this by executing the TPC-H queries on the SimpleScalar simulator (with the SA-1100 model) and by studying the cycle times for transferring a data block from memory to the cache. For all experiments, the default configuration is the 512MB RDRAM memory with 8MB banks. In the following section, we study the energy implications of our hardware and software schemes using this setup. We then present the performance overheads.

5.2 Query Energy Evaluation

Figure 7 shows the *normalized* memory energy consumption for our hardware-directed schemes. While presenting our results, we normalize all values with respect to the base case, which is the version with *no* query optimizations. "Static Standby" in Figure 7 indicates the static standby scheme. We see that, by simply putting the modules to standby mode after each access, this scheme is able to achieve an average 55% reduction in memory energy consumption of TPC-H queries when compared to the unoptimized case. The energy improvements are less pronounced in the case of handheld queries (37% reduction on the average). This is mainly because of the different number of tables manipulated by these two types of queries. In the TPC-H case, multiple tables are scattered across various banks and hence, there is a potential of placing more memory banks into low-power modes. In the case of handheld queries, there is just one table scattered across multiple banks, which makes putting modules to a low-power mode more difficult as modules are tightly connected, as far as query access patterns are concerned. We also observe from Figure 7 that the dynamic threshold scheme further extends these improvements through its ability to put a bank into any of the possible low-power modes. On an average, there is a 60% (43%) energy improvement in TPC-H (handheld) queries.

Figure 7 also shows the normalized energy behavior of our query-directed scheme (denoted On/Off Instr). It is evident that this scheme outperforms the best hardware-directed scheme (by an average of 10%) in saving the memory energy consumption. This is because of two main reasons. First, when a bank idleness is estimated, the query-directed scheme has a very good idea about its length (duration). Therefore, it has a potential of choosing the most appropriate low-power mode for a given idleness. Second, based on its idleness estimate, it can also preactivate the bank. This

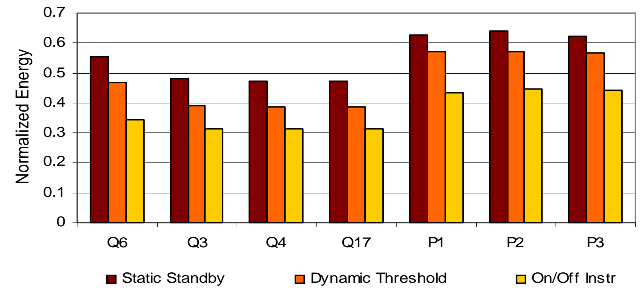

Figure 7: Energy consumption of hardware and software-directed modes. The values shown are normalized to the version with no energy optimizations.

eliminates the time and energy that would otherwise have spent in resynchronization. Consequently, the average memory energy consumption of the query-directed scheme is just 32% of the unoptimized version for TPC-H queries, and 44% in case of organizer (handheld) queries [i.e., an additional 8% (13%) improvement over the hardware schemes for TPC-H (handheld) queries].

5.3 Performance Overhead Analysis

Our techniques are very effective in reducing the memory energy consumption. As mentioned earlier, transitions from the low-power modes to the active mode come with an overhead of resynchronization (in terms of both performance and energy). The energy values reported in previous section take into consideration the extra energy needed to activate the modules as well. In this part, we quantify the basic performance overheads that are faced in supporting our schemes.

Figure 8 shows the performance overheads for both the hardware and software-directed schemes. The static standby scheme has the maximum overhead, which is expected. This is especially the case when queries generate frequent memory accesses. The memory is brought down to the standby mode after each access, and is resynchronized in another access that follows immediately. As a result, the performance worsens as bad as 28% for the static standby case. On the other hand, for the dynamic threshold scheme, the performance overhead is slightly better since the banks are not blindly put to a low-power mode after each access. This verifies our prediction that when a module goes to low-power mode, it would either remain for a while in that mode or may even be transitioned into a lower power mode. The query-directed scheme has the least overhead ($<2\%$). The main reason for this is the ability of pre-activating a bank before it is actually accessed. Therefore, considering both performance and energy results, one may conclude that the query-directed scheme is better than the hardware-directed schemes. However, it is also to be noted that the query-directed scheme requires access to the query optimizer. In comparison, the hardware-based schemes can work with any query optimizer. Therefore, they might be better candidates when it is not possible/profitable to modify the query plan.

6. QUERY RESTRUCTURING

The approaches presented above mainly try to optimize energy consumption without modifying the queries themselves (except maybe for the query-directed scheme where we insert turn on/off instructions in the query plan). In this section, we go one step further, and demonstrate that even larger energy savings are possible if one has the flexibility of reorganizing query operations. We show how this can be achieved in the context of both individual queries and multiple queries (optimized simultaneously). Our main objective in restructuring queries is to *increase* memory bank inter-access times. Note that when bank inter-access time is increased, we can either remain in a given low-power operating mode longer, thereby feeling the potential impact of resynchronization less (i.e., amortizing the cost of resynchronization); or we can switch to a more energy saving mode (as we now have a longer idleness), which means more energy savings. We present different query restructuring strategies for achieving this.

When considering a single query, the bank inter-access times can be increased by re-ordering query operations. On the other hand, the primary goal of the heuristic that targets at multiple-queries is to cluster the usage of tables from multiple queries together, so that the overall table accesses are localized. That is, assuming that we have multiple queries to optimize, our objective is to interleave these query executions in such a way that the reuse of individual tables (or of table portions) is maximized. In other words, when a table is accessed, we want to execute all other query operations (potentially coming from different queries) to that table (one after another), before we move to the next table. This also tends to cluster accesses to the same bank, and tends to increase the bank inter-access times (which is very important from an energy perspective as explained above). In the following, we first study intra-query restructuring and then inter-query restructuring. After these two steps, bank turn-on/off instructions are inserted at the relevant points, depending on the bank access patterns.

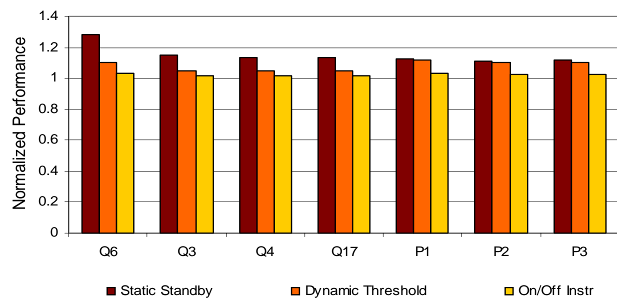

Figure 8: The performance overhead involved in supporting our schemes. There is an average overhead of 15%, 8%, and 2% for standby, dynamic and on/off schemes, respectively, over the unoptimized version.

Step 1 (intra-query optimization): *A query is first examined to see if there are any potential reuse regions. If there are any reusable regions, their accesses are grouped together.*

We achieve this by examining the query execution plan. The query plan is studied to see if there are any advantages in rearranging the operations (primitives) in a query based on its table usage. Operations that require the same (set of) table(s) are then grouped together (i.e., they are scheduled to be executed one after another). The detailed procedure is shown in Figure 9. Each operation in the query plan is first scanned and placed into a *table group* based on the table(s) that it accesses. Then, the operations are rearranged in the query plan (taking into account the dependencies between them) based on their corresponding table groups. For this, we look at the query plan tree. The path from each leaf node to the root, called *stream*, is investigated. The ultimate goal is to schedule operations (*nodes* in the plan tree) based on their table groups. We try to schedule operations within one table group (which is currently active) before scheduling the operations from another table group (which is not active) in an attempt to increase the bank inter-access times. That is, a stream is traversed from bottom to top, and each node within the stream is put to the schedule queue (as they are encountered) based on its table group. It should be emphasized that we preserve the original semantics of the operations (*constraints*) in the algorithm. This procedure is repeated for each stream in the tree, and until all streams have the most energy-efficient schedule based on their table accesses. At the end of this step, an energy-aware schedule queue gets generated for the considered query (saved in *schedule_list*).

Step 2 (inter-query optimization): *Tables are examined to optimize multiple queries simultaneously. For each table that is accessed, all accesses arising from multiple queries to the particular table are grouped together.*

In this step, the *schedule_list* from multiple queries are grouped together. Each list is scanned to identify nodes that access a given table. The nodes that access the same table are then scheduled to execute together (without disturbing the dependency constraints). In fact, the nodes from multiple queries are just grouped (combined) not reordered. Thus, in this step, the constraint flow for each *schedule_list* (taken care of in Step 1) is automatically maintained. Additional conditional flow checks could be reinforced at this stage if desired. Figure 10 shows the regrouping procedure. *final_schedule_list* stores the final consolidated schedule of operations from all the queries.

Step 3 (energy optimizations): *Include energy optimizations by inserting On/Off instructions into the final schedule list.*

In this step, the access costs are calculated for each operation in the *final_schedule_list* as shown in Section 4.2.2. Each operation is attached with an access cost, and the turn-on/off instructions are inserted based on the table inter-access times. The methodology used for adding these instructions to the *final_schedule_list* is the same as in Section 4.2.2, and the on/off markers are placed as elaborated in Section 4.2.3.

As an example, consider two queries Q1 and Q2. Their original

```
table_group is a table-to-operations mapping list.
schedule_list stores the final schedule of operations.

/* identify the group to which an
 * operation belongs */
operation_rearragement (){
 for (each operation in query i) {
   identify the table(s) in i;
   for (each table j in i) {
     add operation to table_group[j];
   }
 }
 schedule_operations();
}

/* schedule operations */
schedule_operations() {
 schedule_list = empty;
 do {
 for (each stream in query plan tree) {
   start from leaf node;
   for (each node in stream) {
     identify its constraint nodes that follow;
     /* the rest are independent nodes */
     group(constraint nodes);
     group(independent nodes);
     check for new violations;
     add new constraints if necessary;
     save the schedule_list;
     move up a node in the stream;
   }
   move to the next stream;
 }
 } until no more changes
}

/* group nodes */
group(node_list) {
 if(node_list is constraint node list)
 {
   for (each node in node_list) {
     lookup table_group of node;
     add node to schedule_list based on table_group;
     /* preserve the dependency order */
     preserve flow of node_list in schedule_list;
   }
 }
 else
 { /* set of independent nodes */
   add node to schedule_list based on table_group;
   /* no need to preserve constraint flow */
   regroup to put all table_group nodes together;
 }
```

Figure 9: Reorganizing operations within a query to optimize for energy (Step 1). The query tree is investigated from the bottom to top for grouping operations based on their table accesses.

```
group_multiple_queries {
  for (each schedule_list) {
    do {
      pick an unscheduled node i in schedule_list;
      /* i.e. pick a node without a "complete" tag */
      for (other schedule_lists) {
       if (node j has same table_group as node i) {
         schedule node j after node i;
         mark node j as "complete";
         /* with respect to multi-query schedule */
       }
      }
    } until all node in schedule_list is "complete"
  }
}

Towards the end of the procedure,
final_schedule_list stores the
entire list of "complete" schedule.
```

Figure 10: Grouping schedule list from multiple queries (Step 2). Operations from multiple queries are grouped based on their table accesses using their corresponding schedule lists.

query plan is shown in Figure 11(i). Q1 is revised as the table accesses are optimizable. Figure 11(ii) shows the result after applying Step 1. Step 2 results in the output depicted in Figure 11(iii). Finally, in Step 3, we insert on/off instructions in appropriate places (see Figure 11(iv)).

7. EXPERIMENTAL EVALUATION OF QUERY RESTRUCTURING

In this section, we evaluate our query restructuring approach by extending our database and queries discussed in Section 5.1.2. As before, our focus is on memory energy consumption. We also study the impact of the technique on the overall performance. Towards the end, other alternative options are also elaborated.

7.1 Multi-Query Setup

Since simultaneous processing of multiple queries is needed to validate our approach, we considered a combination of queries, which we term as *scenarios* in the rest of this paper. Among the queries considered in Section 5.1.2, there can be multiple combinations of queries that arrive sequentially, and that (which) are optimizable using our technique. The various combination (scenarios) of organizer queries and their naming schemes are shown in Table 3. For instance, P12 indicates that P1 is sequentially processed along with P2. The combination scenarios for TPC-H queries are shown in Table 4. The combinations shown in these tables are the prominent ones and the behavior of other combinations are very similar to these, hence, they are not included in this paper.

Table 3: Scenarios for organizer queries.

Type	Legend	Combination
Two query combinations	P11	P1 + P1
	P12	P1 + P2
	P23	P2 + P3
Three query combination	P123	P1 + P2 + P3

Table 4: Scenarios for TPC-H queries.

Type	Legend	Combination
Two query combinations	S11	Q6 + Q6
	S12	Q6 + Q3
	S13	Q6 + Q4
	S14	Q6 + Q17
	S23	Q3 + Q4
	S24	Q3 + Q17
	S34	Q4 + Q17
Three query combinations	S222	Q3 + Q3 + Q3
	S123	Q6 + Q3 + Q4
Four query combinations	S1111	Q6 + Q6 + Q6 + Q6
	S1234	Q6 + Q3 + Q4 + Q17

7.2 Query Energy Evaluation

In this section, we evaluate the query energy of the various scenarios we presented in the previous section. We first study the improvements obtained from our query restructuring heuristic, and further extend our study to combine query restructuring with various hardware and software-directed schemes (of Section 4) meant to improve the energy consumption.

Figure 12 shows the sole contribution of query restructuring scheme in improving the energy consumption. The energy reduces by an average 55% from the unoptimized version when our query restructuring scheme is used. By just grouping similar accesses (to ensure data reuse), query restructuring can achieve significant reduction in the energy consumption of multiple queries.

In order to identify the benefits coming solely from Step 1 (intra-query optimization) in our query restructuring scheme, we also combined Step 1 and Step 3, and compared it with our query-directed scheme (studied in Section 4.2 — which is simply Step 3 of our query restructuring). Figure 13 shows the results. There is up to 19% improvement in energy when operations are shuffled with a query based on their table usage.

When the query restructuring scheme is combined with hardware-directed schemes, there is further improvement in energy savings (Figure 14). The static standby scheme works only for small queries that have a uniform access pattern, but when complex queries are encountered, the dynamic runtime scheme outperforms the static standby scheme due to its good prediction of

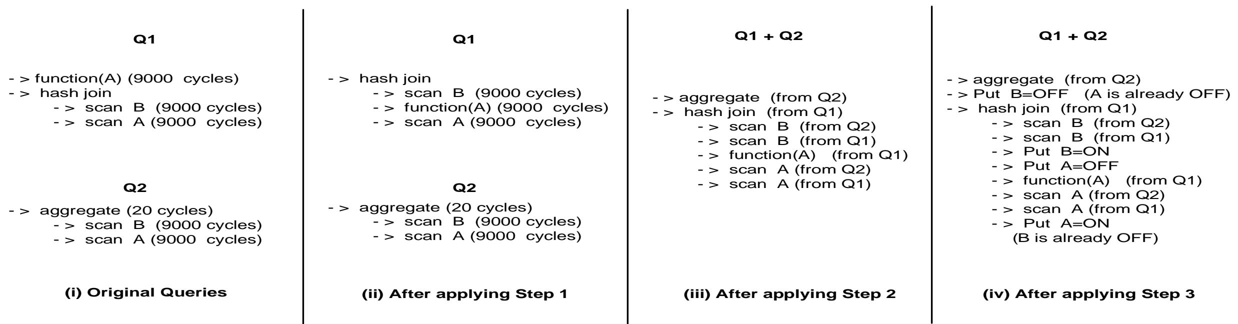

Figure 11: Example of query restructuring and regrouping based on energy behavior.

the application behavior. This can be seen in Figure 14, where the dynamic threshold scheme performs better in the TPC-H scenarios than for the handheld query scenarios. The savings obtained by putting a module into multiple low-power modes for longer periods are more than the savings obtained by periodically putting a module to just standby mode.

The software-directed schemes perform similar to dynamic the runtime threshold strategy when combined with the query restructuring algorithm. In Figure 14, the insertion of explicit turn-on/off instructions improves the energy by an average 78%, when compared to the unoptimized version. This result is comparable to the improvements obtained using the dynamic threshold scheme. In fact, the dynamic threshold scheme performs slightly better for some TPC-H queries (e.g., S12, S13, and S14). This situation occurs due to the following factor. When multiple queries are combined using query restructuring, it becomes difficult to predict the inter-access times since each query has a varying access pattern, and combining random access patterns complicates the job of the predictor (and requires a more sophisticated predictor). The runtime schemes work at the hardware instruction level without any knowledge of the DBMS application. But, this illustrates how a simple software technique implemented at the query optimizer (by just analyzing the high-level query structure) is able to achieve improvements as good as an equivalent but expensive hardware technique.

As mentioned earlier in the paper, when queries are restructured and grouped, the memory access pattern changes. The bank turn-on/off instructions can be inserted only in prominent "hot" and "cold" access regions, respectively. There are a few modules, which is beyond the control of software. For instance, we insert turn-on/off instructions based on tables. A given table could be scattered across many modules. Our predictor estimates the inter-access time for which the entire table needs to be put to low-power mode. However, even during a table access, there are regions (modules) that are hardly used. Dynamic runtime scheme is extremely good in handling this situation by its ability to put individual modules to a low-power state based on just that module's access. This implies that the combination of hardware and software schemes form the best strategy when query restructuring is deployed.

Figure 14 also shows the case when both dynamic runtime scheme and the turn-on/off instructions are used in tandem after query restructuring. The benefits obtained from such a hardware-software interaction is prominent. There is an average 90% reduction in the memory energy consumption across the applications. In some cases, there is up to 95% improvement in the energy consumption. These results clearly show that query restructuring combined with the use of low-power operating modes can lead to significant energy savings.

7.3 Performance Overhead Analysis

Query restructuring combined with the use of low-power modes has an impact on the performance. In Figure 15, we present the normalized system-wide performance of our query restructuring scheme. It is evident that the performance improves by an average of 48% when multiple queries are restructured and grouped. The improvement in performance is mainly due to the improved locality utilization in the memory hierarchy. That is, the data brought to the cache by one query is reused by other queries (as a result of restructuring). We do not present here detailed cache behavior statistics due to lack of space.

Figure 16 shows the normalized performance for the combination schemes as well. When static standby scheme is used with query restructuring, the performance improvements obtained from query restructuring gets negated by the resynchronization overhead from the standby mode for each access. Thus, the performance

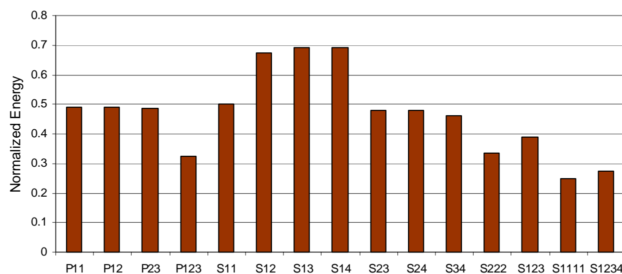

Figure 12: Contribution of query restructuring towards energy improvements. The energy values shown are normalized to the version with no optimizations.

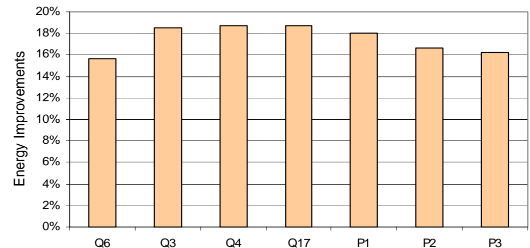

Figure 13: Benefits obtained by restructuring operations within a query (contribution of Step 1).

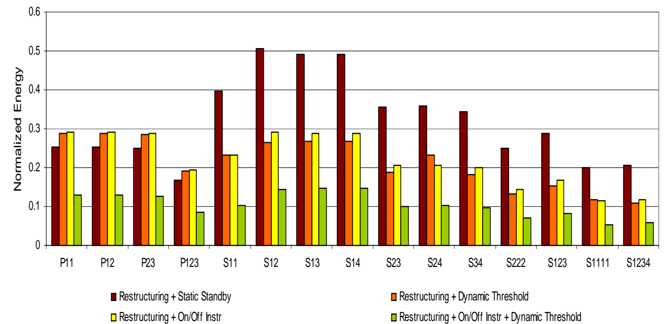

Figure 14: Energy consumption reduces significantly when low-power modes are utilized along with query restructuring scheme. Values shown are normalized to the unoptimized version. Best energy savings comes from a hybrid hardware-software scheme.

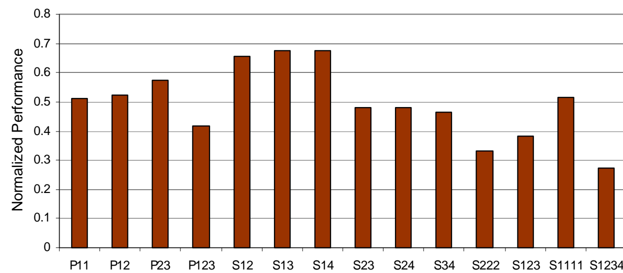

Figure 15: Performance improvement obtained from basic query restructuring over the unoptimized version.

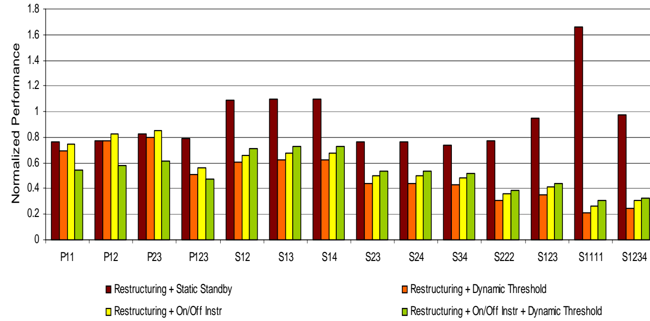

Figure 16: Overall performance after applying energy optimizations along with query restructuring. Values shown are normalized to the unoptimized version.

worsens in some cases by even 65% for complicated queries. However, overall, there is still a 10% performance improvement for all applications on the average. The turn-on/off instructions have the least performance overhead, and hence, preserve the performance improvements obtained from query restructuring. From Figure 16, this combination shows a 47% improvement in performance (negating the improvements obtained from basic query restructuring by a mere 1%). Dynamic runtime threshold on the other hand negates the performance improvements from query restructuring by average 6%. Combining turn-on/off instructions with dynamic runtime threshold shows an average performance improvement of 45% for applications, which implies a 3% overhead addition from the low-power schemes towards query restructuring. Thus, it is clear that query restructuring with both turn-on/off instructions and runtime threshold forms the best alternative from both energy consumption and performance perspectives.

8. CONCLUDING REMARKS

This paper is an attempt to study the potential of employing low-power operating modes to save memory energy during query execution. We propose hardware-directed and software-directed (query-directed) schemes that periodically transition the memory to low-power modes in order to reduce the energy consumption of memory-resident databases. Our experimental evaluations using two sets of queries clearly demonstrate that query-directed schemes perform better than hardware-directed schemes since the query optimizer knows the query access pattern prior to query execution, and can make use of this information in selecting the most suitable mode to use when idleness is detected. This scheme brings about 68% reduction in energy consumption. In addition, the query-directed scheme can also preactivate memory banks before they are actually needed to reduce potential performance penalty.

Our query restructuring scheme based on memory bank accesses provides another scope for optimization. One can re-order operations within a query to increase bank inter-access times. It is also possible to go beyond this, and consider the access patterns of multiple queries at the same time. Multiple queries are optimized based on their table accesses, i.e., all accesses to a table are clustered as much as possible. This scheme is able to put memory banks to low-power operating modes for longer periods of time due to fewer table activations. There is up to 90% improvement in energy and 45% improvement in performance when queries are restructured and regrouped based on their table accesses. Overall, we can conclude that a suitable combination of query restructuring and low-power mode management can bring large energy benefits without hurting performance.

9. REFERENCES

[1] R. Alonso and S. Ganguly. Query optimization for energy efficiency in mobile environments. In *Proc. of the Fifth Workshop on Foundations of Models and Languages for Data and Objects*, 1993.

[2] N. An, S. Gurumurthi, A. Sivasubramaniam, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin. Energy-performance trade-offs for spatial access methods on memory-resident data. *The VLDB Journal*, 11(3):179–197, 2002.

[3] N. Anciaux, L. Bouganim, and P. Pucheral. On finding a memory lower bound for query evaluation in lightweight devices. Technical report, PRiSM - Laboratoire de recherche en informatique, 2003.

[4] T. M. Austin. The simplescalar/arm toolset. SimpleScalar LLC. http://www.simplescalar.com/.

[5] Birdstep Technology. *Database Management In Real-time and Embedded Systems - Technical White Paper*, 2003. http://www.birdstep.com/collaterals/.

[6] Bloor Research Ltd. *Main Memory Databases*, November 1999.

[7] P.A. Boncz, S. Manegold, and M.L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *The VLDB Journal*, pages 54–65, 1999.

[8] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proc. International Symposium on Computer Architecture*, 2000.

[9] Q. Cao, P. Trancoso, J.-L Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. Detailed characterization of a quad pentium pro server running tpc-d. In *Proc. of the International Conference on Computer Design*, 1999.

[10] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, 1999.

[11] G.P. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe ram. In *Proc. of the Fifteenth International Conference on Very Large Data Bases*, pages 327–335, 1989.

[12] Database Management System, The PostgreSQL Global Development Group. *PostgreSQL 7.2*, 2001. http://www.postgresql.org/.

[13] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M.J. Irwin. Dram energy management using software and hardware directed power mode control. In *Proc. of the International Symposium on High-Performance Computer Architecture*, 2001.

[14] Z. Fong. The design and implementation of the postgres query optimizer. Technical report, University of California, Berkeley. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/.

[15] P. Gassner, G.M. Lohman, K.B. Schiefer, and Y. Wang. Query optimization in the ibm db2 family. *Data Engineering Bulletin*, 16(4):4–18, 1993.

[16] Le Gruenwald and S.M. Banik. Energy-efficient transaction management for real-time mobile databases in ad-hoc network environments. In *Proc. of the Second International Conference on Mobile Data Management*, 2001.

[17] Handspring. *Handspring Organizers*, 2004. http://www.handspring.com/products/.

[18] J.M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 23(2):113–157, 1998.

[19] T. Imielinski, S. Viswanathan, and B.R. Badrinath. Energy efficient indexing on air. In *Proc. of ACM SIGMOD Conference*, 1994.

[20] Intel Corporation. *Intel 440BX AGPset: 82443BX Host Bridge/Controller Data Sheet*, April 1998.

[21] A.R. Lebeck, X. Fan, H. Zeng, and C.S. Ellis. Power aware page allocation. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[22] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 491–502. ACM Press, 2003.

[23] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, December 2002.

[24] C.L. Monma and J.B. Sidney. Sequencing with series-parallel precedence constraints. *Mathematics of Operations Research*, 4:215–224, 1979.

[25] Palm Inc. *Palm Handhelds*, 2004. http://www.palm.com/products/.

[26] The PostgreSQL Global Development Group. *PostgreSQL 7.2 – Developers Guide*, 2002. http://www.postgresql.org/docs/.

[27] P. Pucheral, L. Bouganim, P. Valduriez, and C. Bobineau. Picodbms: Scaling down database techniques for the smartcard. *The VLDB Journal*, 12(1):120–132, 2001.

[28] J.M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits*. Prentice Hall, second edition, 2002.

[29] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill publishers, third edition, 2002.

[30] Rambus Inc. *Rambus RDRAM 512MB Datasheet*, 2003.

[31] Samsung Microelectronics. *Mobile 512MB DRAM Chip Series*. http://www.samsung.com/Products/Semiconductor/.

[32] S. Sarawagi and M. Stonebraker. Reordering query execution in tertiary memory databases. In *The VLDB Journal*, pages 156–167, 1996.

[33] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, fourth edition, 2001.

[34] Sleepycat Software. *Berkeley DB V4.2*, 2004. http://www.sleepycat.com/docs/index.html.

[35] Transaction Processing Performance Council. *TPC-H Benchmark Revision 2.0.0*, 2003.