Reducing Energy Consumption of Queries in 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

The tremendous growth of system memories has increased the capacities and capabilities of memory-resident embedded databases, yet current embedded databases need to be tuned in order to take advantage of new memory technologies. In this paper, we study the implications of hosting memory resident databases, and propose hardware and software (query-driven) techniques to improve their performance and energy consumption. We exploit the structured organization of memories, which enables a selective mode of operation in which banks are accessed selectively. Unused banks are placed in a lower power mode based on access pattern information. We propose hardware techniques that dynamically control the memory by making the system adapt to the access patterns that arise from queries. We also propose a software (query-directed) scheme that directly modifies the queries to reduce the energy consumption by ensuring uniform bank accesses. Our results show that these optimizations could lead to at the least 40% reduction in memory energy. We also show that query-directed schemes better utilize the low-power modes, achieving up to 68% improvement.

Categories and Subject Descriptors

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

General Terms

Design, Management

Keywords

database, DRAM, energy, hardware schemes, layouts, mapping,power consumption, query-directed energy management, query optimization

CASES'04, September 22–25, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-890-3/04/0009 ...$5.00.

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.[1]

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* [28]. 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

[1] Designers often use the term “power” almost interchangeably with ”battery life”. However, in embedded devices, one needs to focus on energy consumption rather than power, since there is a limited supply of energy in a battery, even though the power a battery is required to supply can vary over a substantial range during the course of its life. In this study, our focus is on energy consumption.

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 main 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. To see whether query execution can make use of available low-power modes, we study both hardware and software techniques. The hardware techniques predict the idleness of memory banks and switch the inactive (idle) banks (during 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 their energy consumptions using an energy simulator. Our experiments using a set of queries suitable for handheld devices clearly indicate that both hardware-based and query-directed strategies save significant memory energy.

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. Finally, Section 6 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 [9], a modified DRAM model for safely supporting memory-resident databases alike disk-based systems, and for achieving good performance. In PicoDBMS [25], 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. In [32].

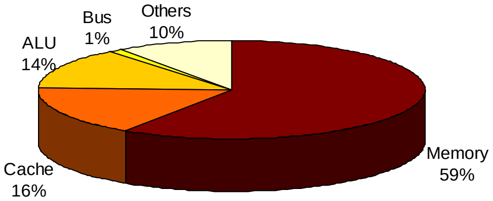


Figure 1: Breakup of the energy consumption for various system components after executing typical queries on a memoryresident DBMS system. The results were obtained by executing the queries on a system that simulates an embedded device with StrongARM SA-1100 processor and RDRAM memory (detailed discussion of our simulation environment and experimental results can be found in Section 5.1).

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 [15]. They consider an environment of mobile hosts. In [20], 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 [17]. 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 [14]. This paper provides insight into design of query cost plans and optimization using various approaches. In [21], 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 [11, 13]. We chose it due to its simple cost models and open source availability.

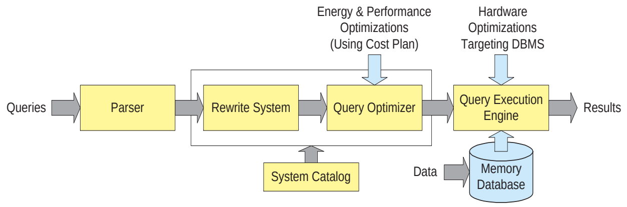


Figure 2: DBMS architecture.

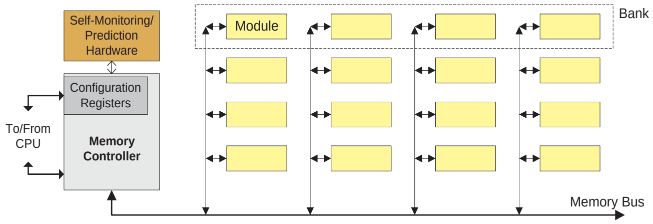


Figure 3: Banked memory architecture.

Our work is unique in spite of the similarities that it has with a few previous work that addresses the control of memory operating modes [12, 19]. Even though the hardware techniques are similar at the basic operational level, it should be noted that our aim is to support a database environment, which is different from typical compiler and operating system tasks, and hence, such basic hardware techniques are not directly extendable. Also, to our knowledge, there is no prior work that address the high-level control of memory operating modes from a database. To be specific, we reorder queries for reducing energy consumption. The energy-aware table-to-bank mapping and the energy-aware optimization that we propose are unique contributions as well. Moreover, our database is completely memory-resident, with the presence of a banked memory environment that gives more freedom for optimizations that are not seen in typical database environments.

3. SYSTEM ARCHITECTURE

3.1 DBMS

For our work, we modified the PostgreSQL DMBS to model a main-memory resident database system. 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 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 ([23] 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 [28] as well as low-end embedded systems [29]. However, the system that we consider is generic, and the proposed optimizations will apply to most bank-organized memory systems. For instance, systems specified in [10, 24] provide multiple ways of organizing banks and also the DRAM devices itself. Our proposed optimizations can be extended to such systems as well.

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*[2]. 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 [28]. These modes are characterized by varying degrees of the module components being active. The details of the power modes are discussed below:

- *Active:* In this mode, the module is always ready to perform a read or write operation. As the memory unit is ready to service any read or write request, the resynchronization time for this mode is the least (zero units), and the energy consumption is the highest.
- *Standby:* In this mode, a few DRAM components are disabled resulting in significant reduction in energy consumption compared to the active mode. The resynchronization time for this mode is typically one or two memory cycles. Some state-of-the-art RDRAM memories already exploit this mode by automatically transitioning into the standby mode at the end of a memory transaction [28].
- *Nap:* This mode can typically consume two orders of magnitude less energy than the active mode, with the resynchronization time being higher by an order of magnitude than the standby mode.
- *Power-Down:* This mode provides another order of magnitude saving in energy. However, the resynchronization time is also significantly higher (typically thousands of cycles).
- *Disabled:* If the content of a module is no longer needed, it is possible to completely disable it (saving even refresh energy). There is no energy consumption in this mode, but the data is lost.

When a module in standby, nap, or power-down mode is re-

[2] Current DRAMs [28] 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.

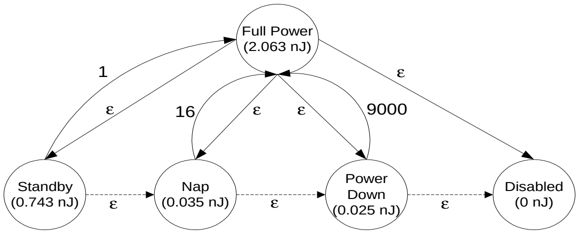


Figure 4: Available operating modes and their resynchronization costs.

quested to perform a memory transaction, it first goes to the active mode, and then performs the requested transaction. Figure 4 shows possible transitions between these modes (the dynamic energy [3] 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). Note that this model is flexible enough to take in different values for energy consumption and resynchronization costs, and the default values used in our experiments are the ones given in Figure 4.

While one could employ all possible transitions given in this figure (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. The energy values shown in this figure have been obtained from the measured current values associated with memory modules documented in memory data sheets (for a 2.5V, 3.3ns cycle time, 8MB modules) [28]. The resynchronization times have also been obtained from the same data sheets.

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. The interface is not only for latching the data and addresses, but is also used to control the configuration and operation of the individual modules as well as their operating modes. For example, the operating mode setting could be done by programming a specific control register in each memory module (as in RDRAM [28]). 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 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). The specific hardware depends on the prediction mechanism that is employed, and will be discussed later in the paper.

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 [18]). Programming these registers using one or more CPU instructions (stores) would result in the desired power mode setting. This brings up the issue of which CPU activity needs to issue such instructions. The memory control registers could potentially be mapped into the user address space directly, making it possible for the user application/DBMS to directly initiate the transitions. However, there are a couple of drawbacks with this approach. The first one is that powering down modules that are shared with other applications brings up the data protection issue. The other problem could be that a given program does not have much knowledge of the memory activity of other programs, and will thus not be able to accommodate more global optimizations. With two or more applications sharing a memory module, the operating system may be a better judge of determining the operating (power) modes. So, the other option is to make the issuance of these instructions a privilege of the operating system, with the DBMS availing of this service via a system call. However, since the focus of this paper is to explore the potential benefits of memory module energy optimizations from the perspective of queries, we focus on a single program environment, and assume that the registers are directly mapped into user space (thus, they can be controlled by the DBMS).

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. Suitable use of low-power operating modes can reduce energy consumption of a given query execution. Also, 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 aspects.

4. POWER MANAGEMENT SCHEMES

In such 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.

4.1 Hardware-Directed Schemes

We explore two hardware-directed approaches that allow the memory system to automatically transition the idle modules 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 [28]. After a read/write access, the memory module that gets accessed can be placed into the standby mode in the following cycle. Such schemes are already available in most

[3] We exclusively concentrate on dynamic power consumption that arises due to bit switching, and do not consider the static (leakage) power consumption [26] in this paper.

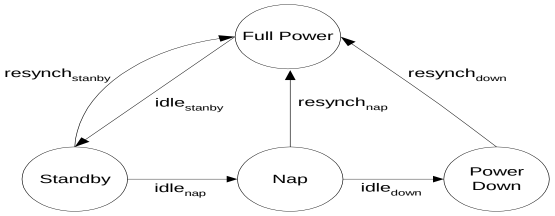


Figure 5: Dynamic threshold scheme.

DRAM systems, and are usually exploited by applications during page placement. 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. The rest of the schemes that we propose aim to minimize the expensive resynchronization costs that are seen in such static schemes.

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. Another alternative to this dynamic scheme would be one based on the adaptive threshold wherein each module adaptively learns the threshold for transition. Our analysis revealed that the hardware implementation costs of such a scheme would be extremely high. Consequently, we do not consider such complex implementations in this paper.

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 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 [11, 14, 27, 30, 31]. 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 [11]. 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 [11]):

$Cost_{seq\_scan} = N_{blocks} + CPU * N_{tuples}$

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. Thus, when a cost is attached to each page (block) read/write operation, an estimate of the access time is obtained 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 [11,13].

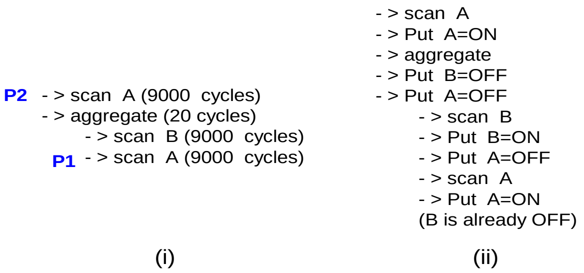
Given a query, we break down each operation within the plan (including sub-plans) and estimate the access cost (in cycles) for each primitive operation. Our objective in estimating the per-operation time in cycles is to eventually identify the inter-access times of operations in the query (and hence, 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 bank should be up in the active mode).

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 query execution plan. For this, we introduce markers (place holders) 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).



```
P2 - > scan  A (9000  cycles)
   - > aggregate (20 cycles)
        - > scan  B (9000  cycles)
     P1 - > scan  A (9000  cycles)
```

(i)

```
- > scan  A
- > Put  A=ON
- > aggregate
- > Put  B=OFF
- > Put  A=OFF
     - > scan  B
     - > Put  B=ON
     - > Put  A=OFF
     - > scan  A
     - > Put  A=ON
    (B is already OFF)
```

(ii)

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

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 [12,19]. We use values

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

Source	Query	Description	Tables	Fields (# characters)

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	a_name = 25
				a_address = 40
				a_city = 25
				a_office phone = 15
				a_home phone = 15
				a_mobile phone = 15
				a_email = 45
				a_web = 150
				a_specialnotes = 150

	P2	Lookup in directory of friends		
	P3	Lookup in directory of colleagues and friends		

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

Memory-resident databases run queries that are different from the typical database queries as seen in typical DBMS. The 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 in Figure 7. Recent organizers [16, 22] 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. A person interested in visiting a city can run this query before he/she leaves for that place. The “friends” view and hence the query P2 is defined in Figure 8. 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. The “friends” view is similar to P2. Figure 9 presents the “colleagues” view and the query P3. The intermediate tables and results during query execution are also stored in the memory.

5.1.3 Default Parameters

For our experiments, we populate the organizer database with 1.3 million records based on parameters specified in Table 1.

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 benchmark 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. The core benchmark characteristics pertaining to the database and memory are shown in Table 2. These characteristics were derived after running the benchmarks without incorporating any of our proposed optimizations (with the default optimizations of Postgres, default parameters of Simple Scalar, and default bank sizes). The instructions executed (in million) indicate the total number of instructions retired in the system. The number of memory reference

```
 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];
```

Figure 7: SQL for query P1

Table 2: Benchmark characteristics

Benchmark	Instructions Executed (million)	# of Loads and Stores (memory references)	Average Bank Idle Times (ms)	Total DB Size
P1	1.796	503403	1.7	637 MB
P2	1.845	614528	3.4	
P3	1.895	727017	3.74	

```
   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;
```

Figure 8: SQL for query P2

```
CREATE VIEW colleagues AS
    SELECT
        a_name,
        a_address,
        a_city,
        a_office_phone,
        a_mobile phone,
        a_email
    FROM
        addressbook
    WHERE
        a_tag = [COLLEAGUE]
    GROUP BY
        a_name;

P3:
   SELECT
        a_address,
        a_home_phone,
        a_office_phone,
        a_mobile phone,
        a_email
   FROM
        friends, colleagues
   WHERE
        a_city = [CITY]
   GROUP BY
        a_name = [a*];
```

Figure 9: SQL for query P3

are based on the total number of loads and stores issued to the system exclusively from the queries. The average bank idle times indicate the total time the memory bank remains unused throughout query execution (approximately 35%). This clearly indicates that there is ample amount of energy that is wasted in the system, and there is room for optimizations. In the following section, we incorporate our hardware and software techniques, and then study the energy implications of those schemes on this default setup. We also present the performance overheads and study the sensitivity of our schemes to various system parameters.

5.2 Query Energy Evaluation

Figure 10 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 10 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 a 37% reduction reduction on the average. These results also depend on the number of tables manipulated by queries. If multiple tables are scattered across various banks, 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 10 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 43% energy improvement in handheld queries.

Figure 10 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 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 44% of the unoptimized version (i.e., an additional 13% improvement over the hardware schemes). The last bar (marked as “History-Based”) in Figure 10 will be discussed later in the paper.

5.3 Bank Idleness Analysis

To better understand the energy behavior of memory banks, we identified the most widely-used low-power mode. That is, we found the mode in which a bank spends most of its time. For this, we profiled the per-cycle energy behavior of each benchmark. Considering the total execution cycles, we found that on an average a given application spends only 66% of its time in active mode in the memory. For the rest of the 44% of the total cycles, the memory re-

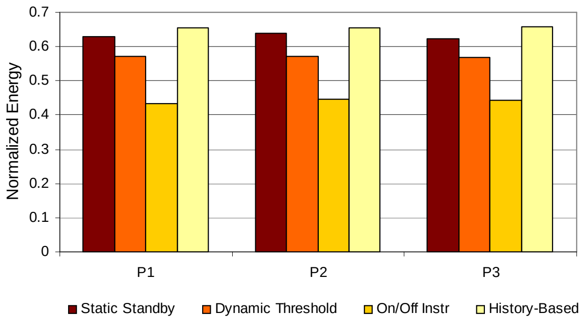


Figure 10: Energy consumption of hardware and software-directed modes. The values shown are normalized to the version with no energy optimizations.

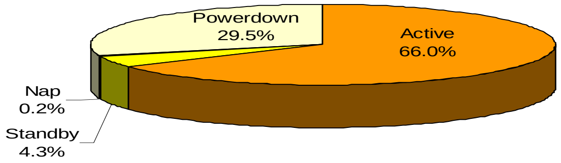


Figure 11: Utilization of various operating modes during query execution. Modules are active most of the time. Power-down is the most frequently-used mode. The threshold of transition is 10, 100, 10000 cycles for standby, nap and power-down respectively.

mains idle. This is illustrated in Figure 11 for the hardware-directed scheme with 10, 100, 1000 cycles as thresholds for transition to the standby, nap and power-down modes, respectively. An interesting note is that the power-down mode is the most widely-used low-power mode, and nap is the least frequently used. This proves that when an application goes to the nap mode, it is more likely to continue on to the power-down mode. This also explains why the static standby mode fails to exhibit good energy behavior as compared to the dynamic threshold scheme (which is able to utilize the most preferred low-power mode). In this case, it is seen that the nap mode is insignificant in contributing towards energy savings. Thus, power-down contributes the maximum benefits, and is also the most sought low-power mode of applications.

5.4 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 12 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 13% for the static standby case. Note that a 13% overhead does not mean that there is 13% memory accesses in the program. This overhead includes the total resynchronization cycles incurred while supporting the static standby scheme. It is possible that a bank is accessed in successive cycles, in which case the banks are not turned off to low-power modes (which implies there could be more memory references than the percentage of overheads). 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 (<3%). The main reason for this is the ability of preactivating a bank before it is actually accessed. In addition, the number of bank turn on/off instructions inserted are less (average of 2%). 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.

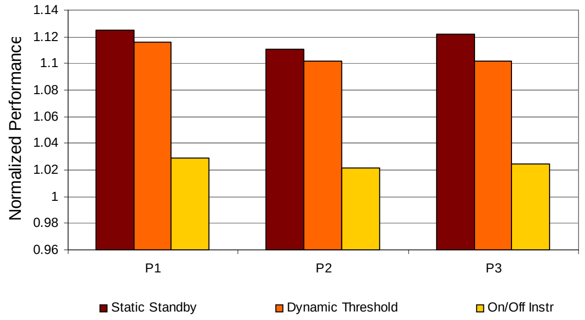


Figure 12: The performance overhead involved in supporting our schemes. There is an average overhead of 11.9%, 10.6%, and 2.5% for standby, dynamic and on/off schemes, respectively, over the unoptimized version.

5.5 Sensitivity Analysis

We now study the sensitivity of our schemes to various key parameters in the simulations.

5.5.1 Number of Banks

We varied the bank size of the memory, keeping the total memory capacity the same. When the size of a bank is increased, the number of banks decreases (for a fixed total memory size). This implies that more data fits into a bank, that is, a table fits into lesser number of banks. This reduces the opportunity to put more banks into a low-power mode. Figure 13 illustrates this by showing the average energy savings for our benchmarks. When the bank size is increased from 4MB to 32MB, the savings in energy starts to drop. Also, too many smaller banks lead to increased resynchronization times. So, care should be taken to choose a fitting bank size for a given system; but, this architectural design issue is beyond the scope of this paper.

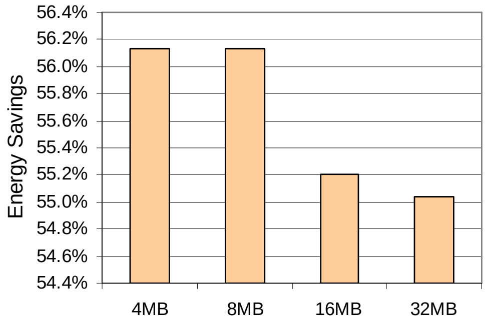


Figure 13: Impact of bank size on the energy consumption. As bank sizes increase (number of banks reduce), there is less savings in energy.

5.5.2 *Idleness Threshold*

In our next set of experiments, we tighten the threshold of Section 5.3 and study the alteration in the behavior of applications. Figure 14 shows the behavior of our queries when the threshold for standby, nap and power-down transition is tightened to 100, 250 and 500 cycles, respectively. When this is compared with Figure 11, it is evident that the usage of standby mode decreases drastically. Also, when a memory module enters the standby mode, it has a high probability of getting transitioned all the way to power-down mode. Thus, the behavior is dependent on the chosen threshold. Such techniques of tightening the thresholds can be deployed to reduce the energy consumption. For instance, if the power-down mode is the most frequently-used mode (and if there is a high probability that when a module enters standby, it will get transitioned all the way), modules could be transitioned directly to power-down mode using turn-on/off instructions instead of using hardware-directed mechanisms. However, it should also be noted that the resynchronization times could increase if the module is frequently transitioned back to active mode from power-down mode. It should be noted that changing the threshold affects the behavior of the entire system. If the standby threshold is too low, it leads to many resynchronizations. If is too high, nap and power-down modes are used more frequently, making the impact of the standby mode insignificant. This is the case for all thresholds. Consequently, care should be taken to ensure that all modes are utilized properly in dynamic schemes.

5.6 History-Based Adaptive Scheme

There are two main problems associated with the dynamic threshold scheme. First, we gradually decay from one mode to another (i.e., to get to power-down, we go through standby and nap), though one could have directly transitioned to the final mode if we had a good estimate. Second, we pay the cost of resynchronization on a memory access if the module has been transitioned. To tackle this problem, we also implemented and conducted experiments with a *history-based scheme.* In this scheme, we try to estimate the bank inter-access time, directly transition to the best energy mode, and activate (resynchronize) the module so that it becomes ready by the time of the next estimated access. While one could use sophisticated history information to estimate bank inter-access time, in this paper, we use a relatively simple mechanism (keeping hardware implementation energy costs in mind): the estimate for the next bank inter-access time is set to the previous bank inter-access time. History-based scheme requires a mode assignment table that contains the maximum and minimum values of the estimated inter-access time for which a particular mode is optimum. This table can easily be pre-constructed based on the energy values and resynchronization times for the different modes available, and needs to hold only as many entries as energy modes. Once the target power mode is determined, the corresponding resynchronization time is subtracted from the inter-access time estimate, to determine the amount of time to spend in that mode.

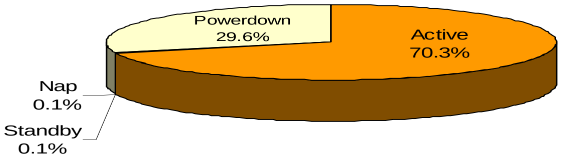


Figure 14: Mode utilization. When threshold for transition is tightened to 100, 250 and 500 cycles (for standby, nap and power-down), the utilization of power-modes also changes. Intermediate low-power modes are very less utilized.

We implemented this scheme using our experimental setup, and studied its energy and performance behavior. The last bar of Figure 10 shows the performance of history-based scheme. There is an average 35% reduction for organizer queries. However, the improvements obtained from our hardware and software-directed schemes of Section 4.1 are better than history-based scheme. This is due to the following reason. We found that it is very difficult to predict/reestimate the bank inter-access times accurately. This is partly due to our particular workloads. In particular, the decision support database workloads exhibit complex memory access behavior, and it is not easy to extract exploitable patterns. While one may argue that a more sophisticated predictor could do better, such a predictor would also have substantial energy and performance cost as well.

6. 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. Overall, we can conclude that a suitable combination of query restructuring and low-power mode management can bring large energy benefits without hurting performance.

7. 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] 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.

[10] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary dram architectures. In *Proc. of the International Symposium on Computer Architecture*, 1999.

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

[12] 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.

[13] 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/.

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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.

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

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

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

[24] B. Prince. *High Performance Memories: New Architecture DRAMs and SRAMs - Evolution and Function*. John Wiley & Sons, 1999.

[25] 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.

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

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

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

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

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

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

[32] G. Weikum, A.C. Konig, A. Kraiss, and M. Sinnwell. Towards self-tuning memory management for data servers. *Data Engineering Bulletin*, 22(2):3–11, 1999.