Analyzing Energy Behavior of Spatial Access Methods for Memory-Resident Data

Ning An Anand Sivasubramaniam Narayanan Vijaykrishnan Mahmut Kandemir
Mary Jane Irwin Sudhanva Gurumurthi

Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
USA
{an,anand,vijay,kandemir,mji,gurumurt}@cse.psu.edu

Abstract

The proliferation of mobile and pervasive computing devices has brought energy constraints into the limelight, together with performance considerations. Energy-conscious design is important at all levels of the system architecture, and the software has a key role to play in conserving the battery energy on these devices. With the increasing popularity of spatial database applications, and their anticipated deployment on mobile devices (such as road atlases and GPS based applications), it is critical to examine the energy implications of spatial data storage and access methods for memory resident datasets. While there has been extensive prior research on spatial access methods on resource-rich environments, this is, perhaps, the first study to examine their suitability for resource-constrained environments. Using a detailed cycle-accurate energy estimation framework and four different datasets, this paper examines the pros and cons of three previously proposed spatial indexing alternatives from both the energy and performance angles. The results from this study can be beneficial to the design and implementation of embedded spatial databases, accelerating their deployment on numerous mobile devices.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

1 Introduction

Computing is becoming a pervasive and ubiquitous part of everyday life. The traditional modus-operandi of sitting at a desk to interact with a computer system is gradually going out of style, with users demanding access to computational resources and information whenever and wherever (even when they are on the move) they choose. These needs have opened the door to several interesting and crucial topics for research in the broad domain of mobile and resource-constrained computing. Focusing specifically on spatial databases (an important and useful class of mobile applications), this paper explores the energy (a scarce and valuable resource in mobile devices) consumption and performance trade-offs of different storage organizations for spatial data on resource-constrained mobile devices.

Programs running on mobile devices (PDAs, laptops, etc.) can be subject to very different operating conditions compared to their desktop/server counterparts. This includes limited computational resources, storage capacity, battery energy, and connectivity, that are a consequence of design considerations such as small form factor, weight, cost and diverse operating conditions. It is widely recognized that battery energy is, perhaps, one of the the most challenging limitations, with many other factors (such as computational speed) directly or indirectly related to energy availability. Mobility precludes the use of a wall socket to power the device, and at the same time one does not wish to carry a heavy battery along for its operation. The growing mismatch between energy capacity of batteries and the energy consumption of mobile devices makes it all that much more critical to employ algorithmic, software and architectural techniques for energy savings. It is hypothesized [13] that high level optimizations in algorithms and data structures can give much more energy savings than micro-managing the energy consuming resources at a very low level. Such optimizations can even amplify the savings obtained from well-known low level energy saving techniques [27], and are thus the motivation for this work.

Database applications are expected to be the dominant workloads running on the mobile devices [2]. This paper specifically focuses on spatial databases, an important class of applications for the mobile devices. In general, Spatial Database Management Systems (SDBMS) [26] have found widespread adoption in numerous areas including Geographical Information Systems (GIS), Image Processing, Computer Aided Design (CAD), Multimedia Systems, and Medical Database Systems. SDBMS are important for mobile computing, with several possible applications in this domain. Already, mobile applications for spatial navigation and querying using a street atlas are available for many PDAs [16, 6]. In addition, traditional data input and querying for conventional SDBMS can be supplanted by mobile operations for better productivity and convenience.

Even in a resource-rich environment, SDBMS design and implementation is a difficult problem [26], because the system has to deal with multidimensional data. Moving the target to a mobile device makes the design and implementation of a SDBMS even more challenging. Resource constraints such as limited energy, computational power and memory add to the complexity of the problem. Performance is not necessarily the only goal for optimization. Sometimes the user may be willing to sacrifice some amount of performance if that will enable the device to run longer on battery. Further, power dissipation of different system components may also be an important issue for thermal considerations.

There are several important and interesting issues in designing a SDBMS for a resource-constrained mobile device, and a few of them include *connectivity* (communication), *data storage and access methods*, *query processing*, and *dynamic adaptation*.

With limited resources, there is the important question of where should the operations (queries) be performed. Does it make sense to ship the operation to a resource-rich server (which may, perhaps, have access to the data) and simply ask the mobile device to act as an intermediary to display the end results to the user, or should the device itself perform the operation? While the former choice is attractive for saving energy (and maybe even speed), there are several practical considerations that may force the latter choice including limited connectivity, energy consuming communication devices [11], and even privacy. Such factors may warrant the storage of the spatial data on the mobile device itself, with the queries directly performed on it, and this paper specifically focuses on such scenarios.

If the dataset needs to be stored in the mobile device, how should it be organized for good performance? Earlier work has focussed mainly on optimizing the retrieval and processing of large disk-resident spatial datasets on server environments. It is imperative to revisit this issue for resource-constrained devices with limited memory and without the presence of a disk (while laptops are equipped with small disks, few other mobile devices enjoy this luxury) not only from the performance viewpoint, but from the energy consumption angle as well.

Query processing and optimization is always a key determinant to performance. Decomposing the high level user request into the fundamental database operations, and deriving a query execution path should be based on both performance and energy consumption. Dynamic adaptation based on changing resource constraints (such as energy, connectivity, etc.) is another important consideration. Modulation of the storage structures, query execution and optimizations is needed when the operating conditions are changing.

Examining all these issues is overly ambitious, and is well beyond the scope of this paper. Instead, we specifically focus on the following problem: *what are the performance and energy implications of storing and processing memory-resident spatial data on a resource-constrained device?* In particular, we address the issue of storing spatial data in main memory and performing certain basic spatial operations on this data including point queries, range queries and nearest-neighbor queries. We assume that all of the dataset is resident in the memory of the mobile device, there is no necessity for communication with a server (no dynamic updates), and complex queries (and their optimizations) are not considered. This is a largely unexplored area, with most previous work on spatial databases examining storage organizations on disks of resource-rich environments. Memory resident spatial data organization [15] has not been extensively studied from the performance angle, let alone the energy viewpoint.

The first step to the development of energy and performance efficient storage organizations for memory-resident spatial data is a rigorous examination of the pros and cons of the already existing solutions [5] that have been proposed for resource-rich environments. Such a study can not only identify energy-performance trade-offs between the existing solutions, but can suggest enhancements, or can even suggest entirely new storage organizations. At the same time, performance and energy profiles can suggest architecture/hardware enhancements to improve the performance and energy savings of resource-constrained systems. This is similar to the motivation behind a recent study [1] that has examined the execution profile of commercial relational DBMSs, except that our focus here is on SDBMS and energy profiling (together with performance-energy trade-offs) that has not been explored before. This paper takes the first step to the development of energy-efficient SDBMS by attempting to answer the following important questions:

- How do the previously proposed alternatives for spatial data organization such as Quadtrees [9, 10], R-trees [8, 14] and Buddy-Trees [25, 24], compare for memory resident datasets in terms of performance? What are the energy consumptions of these different structures when answering queries?

- During the processing of a query, how much energy and time are expended in traversing the index structures to identify candidates that are potential solutions for the query (filtering step)? Subsequently, how

much energy and time are expended in performing the geometric operations on the actual candidate data items to find the exact solutions (refinement step)? Such software profiles are very useful to find hotspots for potential optimization (code restructuring), and to study the pros and cons of the structures in detail.

- For each phase of query processing, how much energy is consumed by the different hardware components of the device - processor core, processor clock, cache, memory and buses? Such a hardware profile can also help us structure the code and suggest architectural enhancements to fix hardware hotspots, potentially without extending the execution time.

- How does the nature of the queries affect the performance and energy profiles? Spatial proximity can translate to improved locality in the data access patterns of the processor, thus reducing the cache and memory energy consumption. At the same time, queries resulting in the selection of several data items can cause capacity and conflict misses in the cache, thereby increasing the energy consumption of the memory hierarchy.

- Traditionally node sizes of the hierarchical index structure are governed by performance related issues such as disk access costs, tree spans, etc. With memory-resident structures, how important a role does node size play in performance for spatial data? Are there any additional insights that an energy perspective can give to the choice of a good node size?

To explore these issues, this study uses a detailed energy and performance estimation execution-driven simulator, called SimplePower [27], that is available in the public domain. Three different storage organizations have been implemented on this simulator, and they have been used to evaluate three kinds of spatial queries on four different datasets. Detailed hardware and software profiles are used to answer the questions listed above.

The rest of this paper is organized as follows. Section 2 gives a quick overview of previously proposed index structures that are used in this evaluation. Section 3 explains the experimental setup and workloads. The results are presented in Section 4 and their implications are given in Section 5. Section 6 summarizes the contributions of this work.

2 Spatial Structures Under Consideration

Numerous spatial data organizations have been proposed [23, 5] and exploring the energy behavior of all these structures is well beyond the scope of this paper. Rather, we select three previously proposed structures - PMR Quadtrees [9, 10], Packed R-Trees [14], and Buddy Trees [25] - that have been argued to perform relatively well for a range of datasets [5]. These structures are also representative examples from the design space of storage structures for spatial data. In Quadtrees, the index nodes at the same level have non-overlapping spatial extents, while R-trees and Buddy-Trees allow overlaps. Quadtrees are improvements over spatial partitioning techniques such as Grid Files, while R-trees are extensions of B-trees for spatial data. Buddy trees are representative of hashing based schemes using a tree structured directory. R-trees give more balanced structures than Quadtrees or Buddy trees.

As for the datasets, we consider line segments in a two dimensional space in this study. We believe that this does not significantly impact the main results and contributions of this work. Line segments represent an important class of datasets, especially in the road atlas applications for the mobile devices. Line segments (or polylines) can be used to represent streets, rivers, etc. Other related studies have also used line segment datasets [9, 10]. In all the structures, the line segments are sorted based on the Hilbert-order [7] of their centroids and kept in an array. The leaf nodes of the structures have pointers (index into the array) to the actual data items. As was mentioned in Section 1, we do not consider dynamic structures in this study, and assume that all the data items are pre-loaded into the memory-resident database (and do not change).

We consider three kinds of queries that have been identified [9, 10] as important operations for line segment databases:

- **Point Queries:** In these queries, the user is interested in finding out all line segments that intersect a given point. For instance, such an operation could be used to find out which streets meet at a given intersection.

- **Range Queries:** These are used to select all line segments intersecting with a specified rectangular window. Very often, the user wants to magnify a portion of the atlas for a closer examination, and this query can serve such a request.

- **Nearest Neighbor Queries:** These are proximity queries where the user is interested in finding the nearest line segment (street) from a given point (e.g. what is the closest street to a given landmark, subway station, etc.). This is the perpendicular distance to the line segment if the perpendicular intersects the segment, and is the distance to one of the end points (closest one) otherwise.

Range and Point queries are typically implemented using a *filtering step* where the possible candidates are first identified using their minimum bounding rectangles (MBRs). Each index node of the hierarchical spatial structures represents a rectangular region of the spatial extent that it covers, and is represented by the MBR of this region. The filtering step, that traverses the index structure, uses these MBRs to identify possible candidates. Subsequently, a *refinement step* is needed to perform the actual geometric operations on each short-listed data item to find the exact answers to the query. In structures (Quadtrees) that do not allow overlapping ranges between the index nodes at the same level, a line segment that spans more than one range

needs to be replicated in all those ranges (we do not consider clipping based approaches that break a segment into multiple parts for each region that it falls in, and recombine/reconcile them in the refinement step). This does not need to be done for structures that allow overlapping ranges such as R-trees. As a result, part of the refinement step for Quadtrees involves duplicate elimination as well.

The Nearest Neighbor query is a little more complicated to implement for index structures, with different previous suggestions [9, 21, 22, 20]. For instance, [21] uses a progressively expanding (in size) range query centered around the query point till the first data item is found. Another possibility [9] is to actually go to that region of the index structure, and examine around this region in the structure instead of composing the searches as separate range queries. A more interesting, and perhaps more efficient, approach is studied in [22] that is the strategy used in this paper. The search starts at the root node and examines the MBRs of its children. It orders these MBRs in terms of distances from the query point, and uses these distances to determine the recursive search order. In addition, it also uses these distances to prune the search when noticing that certain MBRs will definitely contain data items that are closer than those for the other children. The process is then recursively carried out for the candidate child nodes. This is a general technique that can be used for any of the considered hierarchical spatial access methods. The nearest neighbor query does not have separate filtering and refinement steps in our implementation.

Due to space limitations, the reader is referred to [3] for further detailes on the three index structures and the implementation of the three queries using these structures.

3 Experimental Setup

3.1 Energy Estimation Framework

Energy consumption is the integral of the power consumed over operating time. Our energy estimation framework uses *SimplePower*, an architectural-level, cycle-accurate execution-driven energy simulator that is available in the public domain [27]. The architecture of the simulated system includes a single-issue five-stage pipelined integer datapath (instruction fetch (IF), instruction decode/operand fetch (ID), execution (EXE), memory access (MEM), and write-back (WB) stages), on-chip instruction (I) and data (D) caches, that is connected to an external (off-chip) memory. This architecture is representative of some of the current commercial offerings in the PDA domain [27]. The instruction set architecture is a subset of the instruction set (the integer part) of *SimpleScalar*, which is a suite of publicly available tools to simulate modern microprocessors [4]. All results reported in this paper are obtained using the parameters given in Table 1.

Parameter	Value
Supply Voltage	3.3 V
Cache Sizes (each of I and D) (32 bytes line size)	8KB 16KB 32KB
Cache Associativity	Direct-Mapped (DM) 2-way, 4-way
Data Cache Hit Latency	1 cycle
Memory Size	8 MB
Memory Access Latency	100 cycles
Per Access Energy for DM-Caches (nJ)	0.048 (8K) 0.082 (16K) 0.094 (32K)
Per Access Energy for Memory	3.57 nJ
On-Chip Bus Transaction Energy	0.069 nJ
Off-Chip Bus Transaction Energy	6.9 nJ
Per Cycle Clock Energy	0.18 nJ
Technology Parameter	0.35 micron

Table 1: Base configuration parameters used in the experiments.

3.2 Workloads

In our experiments, we use four line segment datasets: (a) **NYCS** contains 12355 streets of New York City, taking about 1.14 MB; (b) **PAFS** contains 16431 streets in Pennsylvania Fulton county, taking about 1MB; (c) **SVR** contains 5848 rivers from the Shenandoah valley, taking 106 KB; and (d) **IRR** contains 12338 railway tracks of Italy, taking 468KB. The first three datasets are taken from the Tiger Dataset [17] while the last is taken from the Digital Chart of the World [12].

These datasets are also representative of some of the SDBMS applications on mobile devices. NYCS and PAFS are typical of road atlas applications for navigation and locational information, the former is for a city and the latter for a rural county. SVR is a dataset that could be useful for hikers/environmentalists on the trails. Finally, IRR is from a different database and would be useful to find the nearest railway track, finding the identity of a station on a track, etc., with the queries that we are considering.

On these datasets, we use the results from 100 runs for each of the three kinds of queries (Point, Range and Nearest Neighbor). Each run uses a different set of query parameters. For the Point queries, we randomly pick one of the end points of line segments in the dataset to compose the query. For the Nearest Neighbor queries, we randomly place the point in the spatial extent in each of the runs. For the Range query, the size (between 0.01% and 1% of the spatial extent), aspect ratio (0.25 to 4) and location of the query windows is chosen randomly from the distribution of the dataset itself (i.e. a denser region is likely to have more query windows). The results presented are the sum total over all 100 runs.

The code sizes for the implementation of the index structures and the storage sizes of the index structures (not including the space taken by the dataset) are given in Table 2 for the chosen fan-outs (see Section 4.1). As far as the code size is concerned, the Quadtree code is a little larger because of the duplicate elimination code that is absent in the other two (the code for building the structures is not included in these sizes). Despite these minor differences, the code size is not very different across these struc-

Index	Code Size	NYCS	PAFS	SVR	IRR
Quadtree	39KB	150KB	183KB	59KB	133KB
R-tree	35KB	285KB	378KB	135KB	285KB
Buddy-Tree	38KB	670KB	989KB	344KB	732KB

Table 2: Code Size and Storage Overheads for the Index Structures

tures. R-tree incurs more storage overheads than Quadtree because of its more balanced nature. Despite the packed R-tree algorithm that is used, some nodes could still be under-utilized. The property of the Buddy-Tree which keeps index nodes that are not entirely packed to capacity (could be much sparser than R-tree nodes), results in a much poorer space utilization compared to the other two structures.

As was mentioned earlier, we do not study the building costs for the structure since we are examining a static situation without dynamic insertions (and the storage structure is downloaded from a server on to the mobile device similar to how it is done in [16]). The chosen dataset sizes and their index overheads are also similar to some of the pocket atlas datasets (e.g., the New York City map that is available in the public domain for PocketStreets [16] running on Windows CE takes 865KB).

3.3 Metrics

We examine both the energy behavior as well as the performance profile for each execution. This helps us understand the trade-offs between the two if any.

For the energy behavior, we profile the consumption (in joules) by each of the hardware components - processor datapath, I-cache, D-cache, Memory, Buses (between cache and memory), and clock network. For the performance profile, we give the breakdown of the cycles spent by the processor performing useful work, and also when stalling on I-cache and D-cache misses.

These profiles are given for each of the query executions on each dataset using the different index structures. Though explicitly not shown here, we have also separated the profiles for the Filtering and Refinement steps [3]. From the hardware perspective, the impact of different cache organizations on energy and performance behavior is also studied.

Energy consumption, execution cycles and the product of these two (denoted as Energy*Delay) are the key metrics that are used for comparison. Energy*Delay helps us capture the relative trade-offs of how much energy savings can be obtained with one alternative over another without significantly degrading performance (or vice versa). We also study energy/cycles values capturing the average energy per unit time (power), which is important for packaging and thermal considerations in [3].

Many of the results and trends are common across the datasets. As a result, *the graphs show the behavior averaged over all the datasets.* Whenever there is a dataset influence, the effects are explicitly mentioned in the discussion.

4 Experimental Results

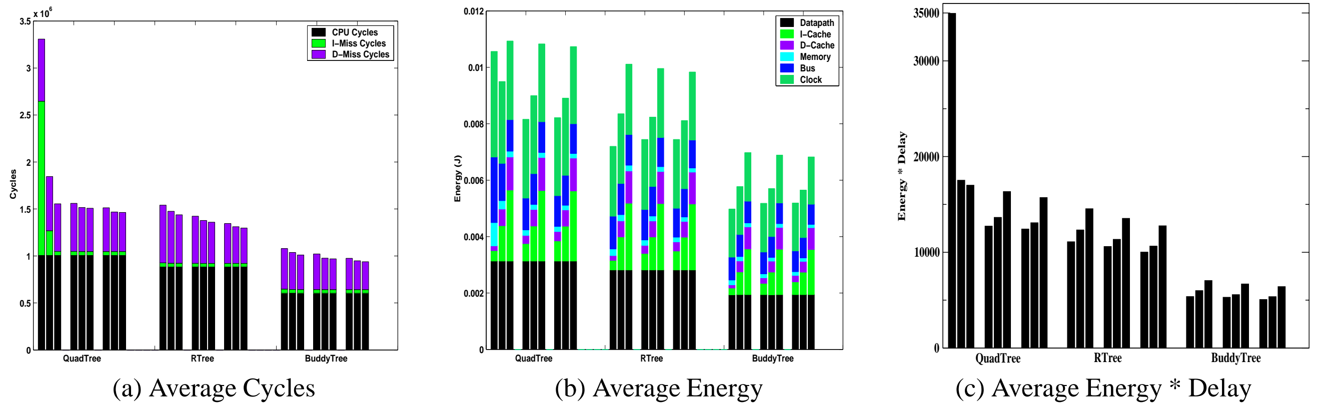
4.1 Impact of Fan-Out

One of the important considerations for each index structure is the fan-out issue. For R-tree and Buddy-Tree, this corresponds to the number of (MBR, ptr) pairs at all levels of the hierarchical structure, with each such entry taking 20 bytes. In the Quadtree, the fan-out of the internal nodes is fixed at 4 entries (taking 80 bytes totally) as per the definition of the structure, and the only choice is for the number of pointers to maintain at the leaf level for the lines falling within this bucket (as suggested in [9]). Apart from the nature of the dataset itself, several factors govern the choice of a fanout. In a disk-based storage structure, the disk access times have a large influence on the choice of the fan-out, and it will be interesting to see how memory resident datasets affect this issue. We have varied the fan-out of the different structures, and collected both performance and energy profiles for the different datasets. The graphs are explicitly given in [3] and we briefly summarize the results here. As fan-out increases, the depth of the tree decreases, thereby improving performance initially. On the other hand, the number of paths to be searched and the number of comparisons at each index node may increase, which worsens performance (in terms of CPU cycles). With these two contrasting factors, the best fan-out that we observe is at 16 for the R-trees, which yields a node size of 320 bytes. We also observed a similar behavior for the fan-out of the leaf nodes of the Quadtree, where the ideal leaf node size again turned out to be 320 bytes (80 pointers). A fan-out of 16 was observed to give the best performance for the Buddy-Tree as well. These observations hold across cache sizes and associativities.

Another interesting observation is that the fan-out has a similar effect on energy consumption as performance, suggesting that using one of these metrics to optimize the fan-out may suffice in practice for the overall energy*delay savings. It should be noted that each query can demand a different fan-out, and it is difficult to predetermine this value unless we have a good idea of the workload imposed on these structures. Since range queries are usually much more prevalent, we have chosen a fanout for each structure that is optimized for the range queries, and use this ideal fanout for all our experiments (regardless of the query).

4.2 Results for Point Queries

Figure 1 shows the performance, energy and energy*delay profiles for the point queries with the different schemes averaged over the four datasets. Examining the execution cycles graph, we see that Quadtree has a higher processor cycle count compared to R-tree. Since the Quadtree does



(a) Average Cycles (b) Average Energy (c) Average Energy * Delay

Figure 1: Comparison of Index Structures for Point Queries. The nine bars from left to right for an index correspond to cache configurations (cache size, cache associativity) of (8K,DM),(8K,2way),(8K,4way),(16K,DM),(16K,2way),(16K,4way),(32K,DM),(32K,2way),(32K,4way).

not allow overlaps (and a point query does not have a high probability of falling in more than one bucket), the filtering step on the Quadtree to find the candidate leaves is relatively fast. However, the refinement step for Quadtree is much more time consuming, placing the sum of these two steps slightly in favor of the R-tree. It should be noted that the overhead for Quadtree in the refinement step is mainly due to the larger number data items (even though the leaf nodes in the two structures have the same size - 320 bytes, this corresponds to 80 data items in the Quadtree which stores only pointers and to 16 data items in the R-tree which stores pointers and MBRs) that the Quadtree has to deal with in the refinement step. The reason we chose the 80 data items fanout in our implementation was because it gives good performance for range queries (as mentioned earlier). Though the graphs are not explicitly shown, we would like to point out that a 16 data item fanout (same as the R-tree leaf node), does indeed give better performance for point queries with Quadtrees, cutting down the overhead of the refinement step, thus making the Quadtree performance similar (or even slightly better in some cases) to the R-tree. Between the R-tree and Buddy-Tree, we find the latter giving better performance. This is mainly due to the splitting criteria for a node, where the Buddy-Tree partitions based on spatial locations while the packed R-tree just uses Hilbert order groupings.

We find that Quadtree has better data locality than R-tree, which can be explained with the higher internal node fanout and overlapping buckets in the latter. With overlapping buckets, R-tree may entail searching more paths even with a point query (while the Quadtree is more focussed). Since the fanout of the internal nodes is higher, there is more scope for eviction of data items from the cache that may be needed again. The Buddy-Tree locality falls between these two.

It is difficult to comment on the I-cache locality behavior of the different codes without clearly understanding where the called procedures fall within the code segment and how they reference each other (for conflicts). In general, we find that 8K caches direct-mapped caches are not a good idea for the I-cache (which is true in later queries as well). Most of the penalties are reduced with a 16K 2-way I-cache.

From the energy perspective, we find that the datapath (and the resulting clock) contribution to the overall energy consumption is quite a significant portion reflecting the importance of the CPU cycles spent in instruction execution in the performance graphs. At the same time, there is a significant portion that is expended in the other system components (caches, memory and buses) as well. Overall, the differences between the three indexes in terms of energy consumption reflect the same observations that were made between them from the performance perspective. As a result, for this set of experiments the energy and performance results go hand-in-hand to a large extent. The only exception to note is the I-cache and D-cache energy consumption changes as we change the cache configuration. With improved (larger size or better associativity), the miss rate is expected to go down, but at the same time energy cost incurred per access goes up. These two factors can help us decide on a good energy-delay conscious cache configuration (captured by the energy*delay values in Figure 1(c)). With 16K and 32K (I and D) caches, most of the locality in instruction and data references is captured well by these configurations, and the energy increase with associativity is more significant. As a result, with these cache sizes, it would be better to have a direct-map structure from the energy*delay perspective (see Figure 1(c)). With 8K I and D caches, we find that the performance penalties due to conflict misses are quite severe, preferring a higher associativity from the energy*delay perspective. For the Quadtree (especially due to its high I-cache misses), an associativity of 4 is needed, while R-tree and Buddy-Tree give the best energy*delay for a direct-mapped cache.

Despite the depiction of lower cycles, energy and energy*delay for R-tree over Quadtree in Figure 1, we would like to reiterate that these differences are mainly due to the differences in the chosen fanouts. We believe that these two structures are more or less comparable in terms of these metrics if we fine-tune the fan-out values at the leaf level for the Quadtree to suit this query. In terms of all these

metrics, we find that Buddy-Tree delivers the best results for point queries.

4.3 Results for Range Queries

Figure 2 shows the performance, energy and energy*delay profiles for the range queries with different schemes averaged over the four datasets. Rather than repeat all the observations that are similar to those for the point query, we would like to point out the differences. The first noticeable difference is that the Quadtree performs much worse than the R-tree and Buddy-Tree in terms of both performance and energy (despite having chosen a fan-out that gives the best performance for the Quadtree). Compared to the point query, range queries have higher likelihood of covering spatial extents of more than one leaf node. As a result, the searches are not that focussed any more on a Quadtree, and more than one path may need to be searched. Second, since the region boundaries of a Quadtree's index nodes are pre-determined and are not adapted to a dataset's vagaries, there is the scope for traversing more paths in a Quadtree compared to the R-tree. Finally, non-overlapping boundaries of index nodes can result in a data item being replicated, and duplication elimination is time-consuming for the Quad-tree. In the filtering step, all candidates are inserted into a list. The refinement step for the Quadtree first sorts this list to remove duplicates, and then performs an item-by-item comparison.

In general, we find that range queries are more processor intensive than point queries, with a smaller fraction of the time spent stalling on cache misses for both index structures. The significance of the refinement step which has good locality (due to sequentially searching a list for exact matches) in the overall performance picture is the main reason for this behavior (reader is referred to [3] for detailed profiles on filtering and refinement steps).

We find that performance is the main factor governing these schemes when we examine them from the energy and energy-delay perspectives. Higher number of instruction executions imply a larger datapath energy and clock energy. At the same time, each instruction fetch references the I-cache, incurring an energy cost (even when it is a hit).

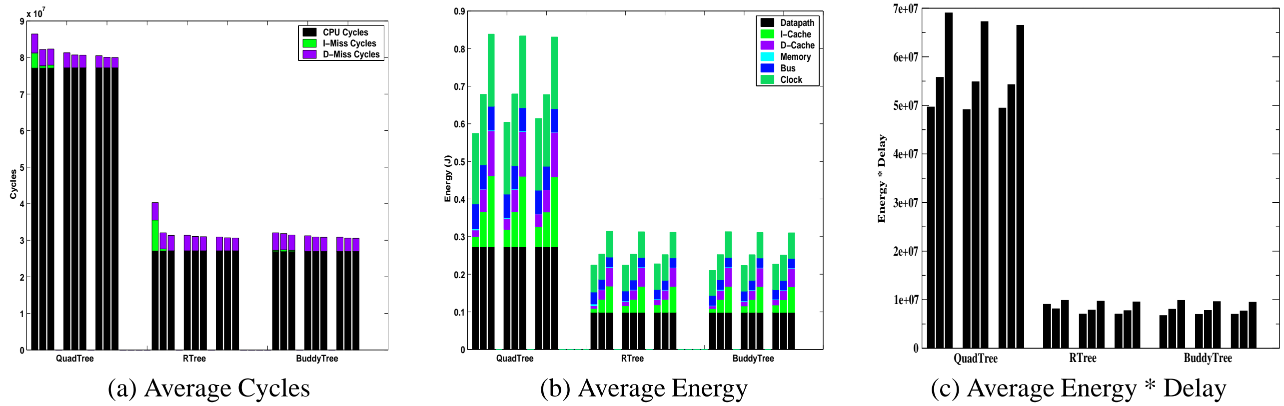
In the point queries, we could see both performance and energy impact of cache configurations playing significant roles when determining a good operating point in both R-trees and Quadtrees. In the range queries, we find that the energy*delay metric obeys the performance (cycles) trend in nearly all cases (except for the Buddy-Tree with 8K 4-way caches). Both R-trees and Buddy-Trees do a good job for this query along all three perspectives - performance, energy, and energy-delay, with Buddy-Trees having a slight edge.

4.4 Results for Nearest Neighbor Queries

Figure 3 shows the performance, energy, and energy*delay profiles for the nearest neighbor queries with the different schemes averaged over the four datasets. This query presents an entirely different picture from what we have observed in the previous two queries. Compared to the previous two, the results show that cache misses dominate the execution time, and processor cycles are a much smaller fraction in many of the datapoints. In the first place, there is no separate refinement step for this query, with data items examined closely when they are first encountered. Even with the earlier queries we pointed out that misses are more significant in the filtering step (tree traversals) than in the refinement step. Further, the working set sizes for implementing the nearest neighbor algorithm are higher than for point/range queries. Specifically, when traversing a subtree, closest distances need to be calculated for all children and they need to be sorted and pruned, before recursively traversing them. Point/Range queries can examine children one at a time, moving to the next after traversing the subtree under the previous child. These operations make the nearest neighbor query much more dependent on miss penalties. The number of D-misses is closely related to the number of children (fanout of internal nodes), which also explains why R-tree has higher D-cache misses compared to Quadtree and Buddy-Tree. The I-misses show a reverse behavior with R-trees having better code locality (except for the 8K DM case) than Quad-trees. Since R-tree has a larger fanout and lower depth, the sorting/pruning operations and overheads are amortized over a larger number of children at a time, while the Quadtree and Buddy-Tree may keep switching between traversal and pruning more often. Overall, from the performance viewpoint, we find that R-tree does the best except for the 8K DM cache. Of the other two, the Quadtree outperforms the Buddy-Tree in many cases.

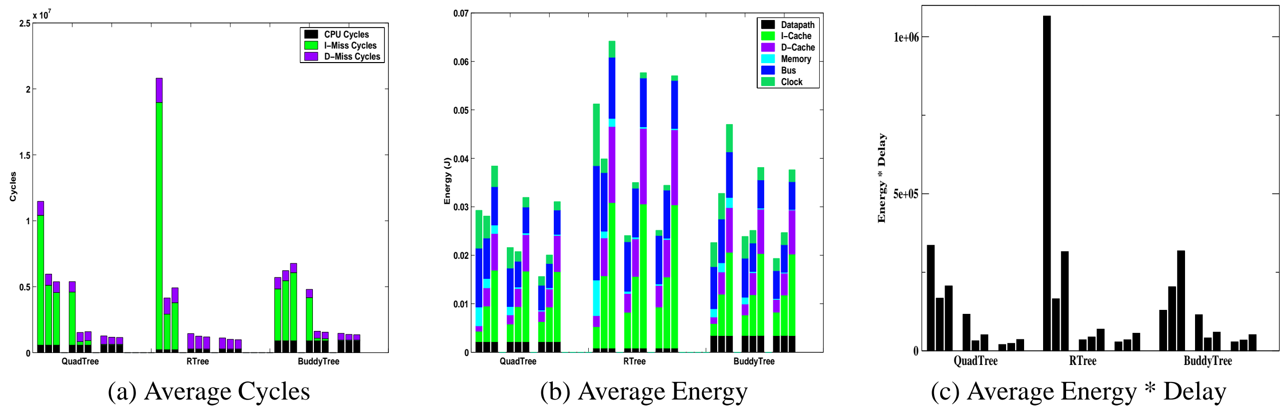
While there was not a noticeable difference in the relative performance of the schemes across the datasets for the previous two queries, we would like to mention that there is a difference between the datasets for this query with the Buddy-Tree structure (there was not a significant effect on the other two indexes). In datasets that are much more clustered (NYCS and IRR), the Buddy-Tree incurred more processor cycles than the others since it does not do as good a job as the R-tree (or even the Quadtree) in balancing the hierarchical structure to reduce the number of levels. For the other two datasets, its performance becomes comparable to the R-tree.

The most interesting observation with this query (compare Figures 3(a) and (b)) is that **better performance does not necessarily imply better energy** (except in 8K DM). R-tree takes fewer cycles to service the query, while Quadtree takes lower energy (with Buddy-Tree energy falling in between). The reason for this behavior can be explained as follows. R-tree incurs much lower CPU cycles than the Quadtree, but incurs higher cache misses. Miss penalties (which require crossing pin boundaries and bus to get to main memory) translate to much more overheads in terms of energy (*off-chip energy*) compared to performance. While the additional miss cycles are not significant enough to put R-tree overall cycles higher than Quadtree, the miss energy (in D-cache, Bus and memory) overhead



(a) Average Cycles (b) Average Energy (c) Average Energy * Delay

Figure 2: Comparison of Index Structures for Range Queries. The nine bars from left to right for an index correspond to cache configurations (cache size, cache associativity) of (8K,DM),(8K,2way),(8K,4way),(16K,DM),(16K,2way),(16K,4way),(32K,DM),(32K,2way),(32K,4way).



(a) Average Cycles (b) Average Energy (c) Average Energy * Delay

Figure 3: Comparison of Index Structures for Nearest Neighbor Queries. The nine bars from left to right for an index correspond to cache configurations (cache size, cache associativity) of (8K,DM),(8K,2way),(8K,4way),(16K,DM),(16K,2way),(16K,4way),(32K,DM),(32K,2way),(32K,4way).

compensates for any savings in the lower datapath energy (e.g. compare the datapath, D-cache, bus and memory energy components for the R-tree with that for the Quadtree for the 32K 2-way caches). The Buddy-Tree energy falls between that for R-tree and Quadtree in most cases.

A consequence of the differences between the energy and performance behavior for this query is the interesting observation in the resulting energy*delay metric shown in Figure 3(c). This captures the facets of whether the improvement in performance warrants the additional energy that is expended. The results show that *even though R-tree is better in terms of performance, Quadtree (which is better in terms of energy consumption) may be a better alternative from the energy*delay perspective* (i.e. the performance benefits for the R-tree come at a much higher energy cost that it may not be as attractive in energy-constrained environments) in most of the better cache configurations. The energy*delay of Buddy-Tree falls in between these two in many cases.

5 Discussion

Despite the relatively small size of the datasets (to fit in the main memory or resource-constrained devices) it is imperative to provide an index-based spatial access method to answer the three considered queries. Performance penalties of brute-force approaches, that do not use an index, are so significant (despite not incurring storage overheads needed to maintain index structures), and have a direct consequence on energy costs as well (the reader is referred to [3] for detailed results comparing brute force approaches to index-based access methods). If storage space overhead is a major concern for the resource-constrained environments, Quadtree is a better alternative than the other two structures. Of the other two, the packed R-tree makes better utilization of the space taken by its index nodes.

Between the three index structures, we find no clear winner across all queries and criteria that have been studied. Table 3 summarizes some of the observations that have been made in the earlier sections. It ranks the schemes (from 1 to 3) based on their relative merits for the performance, energy, and energy-delay criteria, and a list of

	Cycles	Energy	Energy*Delay
Point Query	1. Buddy-Tree 2. R-tree,Quadtree	1. Buddy-Tree 2. R-tree,Quadtree	1. Buddy-Tree 2. R-tree,QuadTree
Range Query	1. Buddy-Tree,R-tree 2. Quadtree	1. Buddy-Tree,R-tree 2. Quadtree	1. Buddy-Tree,R-tree 2. Quadtree
Nearest Neighbor	1. R-tree 2. Quadtree 3. Buddy-Tree	1. Quadtree 2. Buddy-Tree 3. R-tree	1. Quadtree 2. Buddy-Tree 3. R-tree

Table 3: Comparison of Index Structures for different queries and criteria using the results of 2-way 16K cache configuration. (1) denotes the best and (3) denotes the worst for each entry in this table

observations follow:

- For the point queries, we find the Buddy-Tree giving better performance while incurring a lower energy cost. Consequently, it has the lowest energy*delay values of the three. Between the other two, the differences are not very prominent, especially if we can tune their fan-outs for this query.
- With range queries, both R-trees and Buddy-Trees are giving good performance, energy savings and energy*delay values. Quadtree is worse than these with queries needing to process more data for refinement.
- While performance largely dictates energy costs for the point and range queries, this study has shown that these criteria do not always go hand-in-hand. There could be circumstances when a scheme giving the best performance can incur the highest energy cost. This was observed with the nearest neighbor query where R-tree was giving the best performance but incurs the highest energy. Quadtrees turn out to be better from the energy or energy*delay perspective for this query.
- The results show that index-based query executions on spatial databases exercise the memory system considerably. A similar result has been noted recently in [1] where misses have been found to constitute around 40% of the execution time for memory-resident relational databases. The energy perspective shows that it is not only important to optimize miss behavior (by lowering number of misses, or by reducing energy consumption during misses), it is crucial to optimize energy consumption of hits as well (or even reduce the number of memory references). Energy consumption of caches plays an even more dominant role than its performance impact. While improving the caches (in terms of size and associativity) can reduce the miss behavior, the access costs increase. Consequently, we find that 16K 2-way associative caches are a good compromise between performance and energy for these workloads. Architectural techniques to reduce cache energy consumption and their benefit on these workloads is studied in [3].

These observations can help a designer customize a SDBMS for a given target resource-constrained environment, fine-tune the implementation to dynamically adapt for changing energy and performance criteria, and to even provide guidelines on incorporating architectural enhancements that can help meet energy-performance criteria in a more effective manner.

In addition to energy, power dissipation is another important consideration to keep the packaging and cooling costs low. The reader is referred to [3] for some exploratory issues with respect to power dissipation which show that datasets can have as much as an impact on power dissipation as the index structure itself.

6 Concluding Remarks

The growth in mobile computing has made mobile databases one of the most prominent segments of embedded database market [18, 19]. The market for embedded databases is expected to grow about 12% annually to 705 million dollars in 2003. Many of these applications are targeted for automotive and handheld devices which are likely to hold, access and process spatial data. With the resource-constraints imposed on these embedded environments, energy and limited memory designs, take center-stage together with performance. This paper has presented the first in-depth examination of memory-resident spatial access methods for three index structure (Quadtrees, R-trees and Buddy-Trees) from the energy, performance and energy-delay perspectives. By doing so, this paper has identified the key issues affecting both energy and performance, at the algorithmic and architectural levels. It has taught us several important lessons including the fact that optimizing performance does not necessarily optimize energy and could in fact aggravate power dissipation. Since the target environments may have different (storage) capacities, processing power, and resource-constraints, the results from this work can help to select and tailor the spatial access methods for designing mobile applications operating in diverse conditions. This exploration can in turn provide insight on new problems for research on embedded and spatial databases, accelerating their deployment on numerous mobile devices.

Acknowledgements

This research has been supported in part by several NSF grants: CCR-9988164, CCR-9900701, CCR-0097998, DMI-0075572, Career Award MIP-9701475, NSF grants CCR-0093082, CCR-0093085, CCR-0073419, CCR-0082064, 9705128, 9617308, and equipment grant EIA-9818327.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs On a Modern Processor: Where Does Time Go? In *Proceedings of Very Large Databases Conference*, 1999.

[2] R. Alonso and H. F. Korth. Database System Issues in Nomadic Computing. In *Proceedings of the ACM-SIGMOD Conference*, pages 388–392, 1993.

[3] N. An, A. Sivasubramaniam, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and S. Gurumurthi. Analyzing Energy Behavior of Spatial Access Methods for Memory-Resident Data. Technical Report CSE-00-023, Dept. of Computer Science and Engineering, The Pennsylvania State University, November 2000.

[4] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical report, Computer Sciences Department, University of Wisconsin, June 1997.

[5] V. Gaede and O. Gunther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–230, June 1998.

[6] GEOPlace.Com. Mobile Technology Takes GIS to the Field. http://www.geoplace.com/gw/2000/0600/0600IND.ASP.

[7] J. G. Griffiths. An Algorithm for Displaying a Class of Space-filling Curves . *Software - Practice and Experience (SPE)*, 16(5):403–411, 1986.

[8] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM-SIGMOD Conference*, pages 47–57, 1984.

[9] E. G. Hoel and H. Samet. Efficient Processing of Spatial Queries in Line Segment Databases. In *Proceedings of the 2nd Symposium on Advances in Spatial Databases(SSD)*, pages 237–256, 1991.

[10] E. G. Hoel and H. Samet. A Qualitative Comparison Study of Data Structures for Large Line Segment Databases. In *Proceedings of the ACM SIGMOD*, pages 205–214, 1992.

[11] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Energy Efficient Indexing on Air. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 25–36, 1994.

[12] Environmental Systems Research Institute. Digital Chart of the World. http://www.maproom.psu.edu/dcw/.

[13] M.J. Irwin, M. Kandemir, N. Vijaykrishnan, and A. Sivasubramaniam. A holistic approach to system level energy optimization. In *Proceedings of the International Workshop on Power and Timing Modeling, Optimization, and Simulation*, September 2000.

[14] I. Kamel and C. Faloutsos. On Packing R-trees. In *Proceedings of the ACM CIKM*, pages 490–499, Washington, DC, 1993.

[15] T. Lehman and M. J. Carey. Query Processing in Main Memory Database Management Systems. In *Proceedings of the 1998 ACM-SIGMOD Conference*, pages 239–250, 1998.

[16] Microsoft. Microsoft Pocket Streets. http://www.microsoft.com/mobile/downloads/streets.asp.

[17] U. S. Bureau of the Census. TIGER/Line(R) 1995 Data. http://www.esri.com/data/online/tiger/index.html.

[18] M. A. Olson. Selecting and Implementing an Embedded Database System. *Computer*, 33(9):27–34, September 2000.

[19] S. Ortiz. Embedded Databases Come out of Hiding. *Computer*, 33(3):16–19, March 2000.

[20] A. Papadopoulos and Y. Manolopoulos. Performance of Nearest Neighbor Queries in R-Trees. In *Proceedings of Intl. Conference on Database Theory*, pages 394–408, 1997.

[21] J. M. Patel. *Efficient Database Support for Spatial Applications*. PhD thesis, University of Wisconsin-Madison, 1998.

[22] N. Roussopoulos et al. Nearest Neighbor Queries. In *Proceedings of the ACM SIGMOD*, pages 71–79, 1995.

[23] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[24] B. Seeger. Performance Comparison of Segment Access Methods Implemented on Top of the Buddy-Tree. In *Proceedings of the Second International Symposium on Advances in Spatial Databases*, pages 227–296, 1991.

[25] B. Seeger and H-P. Kriegel. The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems. In *Proceedings of the VLDB*, pages 590–601, 1990.

[26] S. Shekhar, S. Chawla, S. Ravada, et al. Spatial Databases - Accomplishments and Research Needs. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):45–55, 1999.

[27] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Kim, and W. Ye. An energy estimation framework with integrated hardware-software optimizations. In *Proceedings of the International Symposium on Computer Architecture*, 2000.