CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

The Beach System: Building a PC from Many Tiny Computers - A First Step at Virtualization -

Makes life easy at hand, Even with a million motes of sand,
Rather than deal with each, Leave it to the Beach.

Boris Capitanu, Ellick Chan, Indranil Gupta
University of Illinois at Urbana-Champaign
Department of Computer Science
<capitanu, emchan, indy>@uiuc.edu

***Abstract* –The emergence of tiny computers, such as smart dust, Berkeley motes and Intel motes, makes it feasible to envision the conversion of a network of tiny computers into a regular computing device (i.e., a "PC" or personal computer). While the falling cost and increasing (yet tiny) computation power of these miniature computers portend well for this vision, there are significant technical hurdles. In this paper, we take a first step at building "PCs" out of such tiny computer networks, in order to run regular PC applications. Our system, called *Beach*, virtualizes the memory accessed by an application at a single sensor mote (a type of tiny computer), thus enabling this memory to be distributed out over multiple such motes. By using distributed page tables and caching, we transform the puny memory at each mote (few KBs) into several KBs of memory. We present trace-driven experimental results from running regular PC applications (e.g., sorting) on top of the Beach system. Due to the exploratory nature of this research, we ignore scalability and fault-tolerance issues for now. Our work provides initial insight into the pros and cons of the vision.**

1. INTRODUCTION

Tiny computers such as smart dust, Berkeley motes and Intel motes (I-motes) have small capabilities for computation (few MHz CPU), memory (few KBs), communication (few 10s of KBps) and energy (few days on full batteries at 100% operating time). Currently, such hardware is being used for mostly sensor-type applications, e.g., environmental monitoring, battlefield tracking, building infrastructures, etc.

However, with the decreasing costs (and sizes) of such tiny computers (e.g., motes), we envision that it will soon be feasible to use them for a different purpose. By stripping away the sensor hardware from such motes, and instead stringing together a network of tiny computers, the collection of motes can be made to function like a PC. Regular computation-intensive PC applications can then be run on such a "bottle of motes".

Our design decisions are motivated by systems-level goals. Any network-wide operating system consists of several components – virtual memory, processes, file systems, resource management software. For the tiny computer networks envisioned, we believe each of these components poses enough challenges that they should be dealt with separately and individually.

In light of the scarcity of work in this area so far, this paper takes a first step at implementing *virtualization* of memory across a collection of motes. We present the design of a new system, called the *Beach system*, which through virtualization and caching, enables a computation-intensive application to run on a single mote, and yet spread its virtual memory over other motes accessible over the network.

Through the Beach system, the master mote (which is running the application) will be able to malloc, read and write remote pieces of memory on other motes. The Beach system is customizable. Since we are given little optimization information about application-specific requirements, we will allow the user to tune operational parameters at compile time, thus avoiding the performance penalty of runtime flexibility.

We are currently in the process of deploying a prototype of Beach on the Intel motes. We present simulation results of our TinyOS prototype, driven by traces from traditional PC applications. This helps us identify the pros and the cons of this direction.

2. Related Work

Existing research for sensor networks primarily focuses on routing, power management, storage, and reliability. As far as we are aware of, there is no research done in the area of distributed memory in sensor networks. Other pertinent research areas include:

- *Distributed Memory:*

Current approaches use a memory management unit to divide the address space into pages. Pages may be present in memory, or swapped out to secondary storage such as network-mounted disks. These approaches are optimized for utilizing idle memory resources of peers on high-speed LANs [1, 2]. In the mobile realm, [3] proposes a model where PDAs and mobile devices request memory from nearby desktop systems. Furthermore, these designs rely on the presence of hardware support for virtualized memory. Such facilities are unavailable on most mote platforms.

- *Storage Management:*

The two notable storage management solutions are Freenet [10] and OceanStore [11], which use introspection techniques that guarantee optimality of data placement over time. Existing file sharing applications offer similar fundamental principles to network page management requirements such as page replication and eviction. Judicious selection of parameters can result in a custom-tailored protocol optimized for distributed memory in sensor networks. The combination of these techniques offers better energy efficiency and increased performance.

- *In-network processing*:

Approaches such as TAG [12] rely on manual decomposition of a task into local decisions made by sensors; data is aggregated at each node and processed at the source. This approach is not applicable to a general computational model because it is not necessarily Turing-complete (SQL). Our approach differs by allowing a mote to use virtualized memory resources, moving the tunable parameters closer to the system level.

- *Programming Sensor Networks:*

TinyOS [5, 6, 7] is a set of modules built using the NesC language; it provides an interface to commonly used sensor network functionality. To build a TinyOS application, a programmer connects several components through well-defined interfaces. Events are generated from interrupts, and non-preemptive tasks are posted to a FIFO queue for execution.

Virtual machines, such as Maté [8, 9], provide a clean abstraction to machine resources. Maté allows application programmers to customize the instruction set, which is then translated into bytecodes. A memory virtualization model using this approach will be completely transparent to client applications.

- *Routing:*

Several techniques and optimizations may be applied from existing approaches such as Directed Diffusion [13], which establishes virtual circuits from sources to sinks of information. Geographical routing [14] relies on á priori knowledge of mote placement and topology to ensure optimal routing. LEACH [15] reduces hot-spot formation by ensuring uniform long-term availability of sensor motes. Routing facilities can be used in the formation of a static group of members which can be optimized for low latency communication.

- *Allocation:*

TinyAlloc provides a malloc-like interface to memory on a mote. It allows applications to dynamically allocate space within a specified heap area. TinyAlloc requests are split-phased in consistence with all TinyOS-based operations. In the first phase, a memory request is generated. The second phase signals completion of the request by sending an event to the application. TinyAlloc does not provide an interface for remote memory access. Therefore, an application requiring more memory than what

a single mote can offer does not benefit from this interface.

3. Design Considerations

In a typical distributed shared memory system, memory is divided into pages. A memory management unit (MMU) present on each client manages access to memory and caches. The MMU is responsible for translating virtual addresses used by applications to physical addresses. A page table provides a map of memory addresses and permission bits for each block of memory. In a desktop computer page sizes are typically 4 kilobytes. The effect of finer granularity on pages is a larger overhead for structures describing the allotment of pages. Larger pages require less overhead, however, they can be prone to waste due to internal fragmentation. One approach to help increase the utilization ratio without requiring large amounts of overhead is to split pages into subpages, or keep separate page pools of various sizes. In such a pooled architecture pages may be sized to powers of two, which makes it easy to compute tag addresses for caching.

Distributed shared memory schemes rely on messaging to pass pages between nodes. The judicious selection of parameters, such as appropriate sized quanta for pages, can have great impact on performance. Since pages must be transmitted as network packets, a page size much larger than the optimal packet size may lead to fragmentation and ultimately high loss rates as partially transmitted pages may need to be fully retransmitted to guarantee integrity. Larger packets also imply higher error rates and latency characteristics; however, they offer the lowest overhead and best performance. Smaller packets have the advantage of speed and low failure rate at the cost of high overhead. An application designer must carefully profile the results of changing parameters with requirement metrics to ensure optimality. We briefly analyze and discuss several application scenarios in the evaluation section to suggest some suitable values.

Caching

Caching is vital to performance in any system that relies on frequent access to a working set of data. Several cache classes and optimizations such as prefetching are available at the programmer's dispense. Cache size may affect the application's aptitude for data processing. A small cache has the advantage of allowing a programmer to better partition memory for application-specific needs. A large cache may decrease the miss rate, but occupy more application space and unnecessarily waste memory if the hit rate is low for the application data. Such a scenario may be possible if an application only accesses each memory address once, such as calculating the sum of a large array. Prefetching pages can allow for better performance where the access pattern is predictable by a simple predictor. This optimization has utility in the checksum example, but fails in an example that may involve sorting. Caches are better suited to applications that have high locality of accesses, such as bubble sort.

Cache replacement policies can have great effect on the performance of an application. For applications that repeatedly work on a small subset of data, such as matrix relaxation algorithms (temperature analysis), a least recently used (LRU) replacement policy may be best. An algorithm that repeatedly traverses an array, such as selection sort, would be best suited by a circular replacement policy. LRU, LFU (least frequently used) and circular replacement policies have worst-case scenarios that cause performance degradation. One possible solution may be a random cache eviction policy. This approach has the benefit of providing probabilistically good expected running times, but the random choice of an eviction may not be cheap to compute on a mote.

One clear benefit to caching data comes in the form of write caching. By delaying the commit of a frequently used cache line, we can avoid unnecessary use of the radio. One scenario where this might occur is in a simulation where highly used variables are stored on several

pages. By using a write-back caching policy, changes are not committed until the entry is evicted. This approach works well when there is only a single client, but degenerates in a contentious environment. An alternate approach is a write-through cache, which updates a cache line when written, as well as the original replica. This policy helps ensure fast access on future reads without the consistency problems of the write-back approach. In a small setup with few motes we believe that a write-back policy controlled by a coordinating node would result in the best performance with the least overhead.

Both caching and prefetching may improve the performance of certain applications, but possibly perform unnecessary preparation in others. In the latter situations, the benefits of prefetching and caching and the goal to conserve energy may be a dichotomy. It is the responsibility of the application programmer to determine proper and reasonable parameters through traces and profiling.

Virtualization Interface

As described earlier, memory virtualization techniques typically require hardware assistance (MMU) to effectively maintain the illusion of a linear memory space. Without such a facility, there are several ways to accomplish the same goal. First, a program may employ a virtual machine that interprets instructions and remaps data references. This is the approach taken by Maté, a virtual machine for sensor networks. A second approach is to use relative addressing instructions. This approach is taken by uClinux, a variant of Linux that runs on machines without an MMU. In uClinux, applications are compiled to be relocatable at runtime. The operating system chooses a starting address, and data references are all relative to that address. This approach requires special support from the compiler to generate position independent code. Another approach is to generate a fault on an illegal memory access. Such accesses can be overloaded to implement system calls or virtualize memory. Finally, an application may guard all memory accesses through a special library or compiler preprocessing directives, which redirects array, pointer and memory accesses.

Each of the methodologies described has benefits and drawbacks. The virtual machine approach requires an unnecessarily high performance overhead on code that does not necessarily access remote memory. The relocatable code approach and the memory fault approach require special support from the operating system, hardware, and compiler to properly operate. Whereas the virtualization library approach requires programmer intervention and an implementation would incur additional indirection overhead. To minimize the requirements, we chose to design and implement the virtualization library approach. Although we have indirection overhead on every memory access, we can replace the access instructions with compile-time macros to switch between accesses to remote or local virtual memories with no runtime cost.

Library Interface

Our library exports several familiar functionalities to the C programmer. We attempt to provide similar interfaces to well-established libraries, but we had to make special modifications to adapt to the event-driven programming model of TinyOS. Each operation requires a split-phase asynchronous implementation. First, a user requests a memory operation, then the library processes the data and issues network requests if necessary. Once the VM library has finished its work, it signals the application by raising a TinyOS event. The application then reads the status of the request and continues execution.

In TinyOS, execution units are divided into two classes: tasks and event handlers. Tasks may not be interrupted by other tasks, but they may be preempted by an event handler. The proper way to divide work is to execute short operations that depend on external stimuli in an event handler, and post longer operations as tasks. One way to implement a blocking virtual memory system would be to use a spin lock to suspend execution, waiting for the completion of a memory request. Such an approach may lead to

a code snippet similar to the one depicted in Figure 1.

```
task void doComputation() {
   …
   call VM.readMemory(…);
   while(!ready);
   …
}
event void operationComplete(…) {
   ready = TRUE;
}
```

Figure 1 - Spin lock

The problem with this approach is that the spin lock unnecessarily wastes CPU cycles and energy by waiting on a condition instead of sleeping the processor. This approach also prevents the execution of other independent tasks that might be required to release the lock, potentially causing deadlock. The advantage to the spin lock approach is that programs are executed sequentially in a single-threaded manner. By using the event-driven approach TinyOS advocates, we must rewrite our applications in a way that is conducive to their model. This requires some clever manipulation of program structure and loop unrolling to express programs at the cost of code clarity. Figure 2 demonstrates how a loop may be constructed using the TinyOS approach. The task repeatedly requests memory and processes it, while the event handler posts a task to process the incoming data.

```
task void doComputation() {
   call VM.readMemory(…);
   …
}
event void operationComplete(…) {
   //save received data
   //to a buffer
   post doComputation();
}
```

Figure 2 - Event-based loop

Other Optimizations

In addition to caching, other techniques such as compression, message piggybacking, and differential updates may help reduce the total number of messages exchanged. While considering these options we realized that the benefits may be insignificant, and sometimes nonexistent, due to the fact that most of the time the message packets are completely filled, or the data not easily compressible without the overhead of extra headers and checksums.

4. Core Design

In the Beach system, there are two primary roles: master motes, which request memory resources, and slave motes, which offer and broker these memory resources. Masters maintain a page table with references to remote memory, as well as a cache that improves performance for repeated accesses. Slaves have a persistent data store and an allocation table. In our design, the allocation table and the page table are modeled by the same data structures. The cache and persistent storage structures are also equivalent. Therefore, the same structure assumes different roles depending on functionality of the mote. A single compile-time flag selects the role of a mote. All motes in a mote network run the same binary image, making replication and code updates easy. As a result of our design, running a simulation in the TinyOS simulator (TOSSIM) is very straightforward.

Masters are responsible for computation. They may access memory resources through four basic operations. These operations are: *malloc*, *read*, *write*, and *free*.

Malloc – is a split-phase asynchronous call which creates a contract between master and slave motes for memory allocation. Memory requests are made to the granularity of a page; larger allocations request multiple pages. In the first *malloc* phase, a request for the amount of memory needed is broadcasted to all first-hop slave motes, which respond indicating their ability to satisfy the request. Each response is sent if the node can fulfill all or part of the memory request. For networks reaching their capacity, partial allocations can allow better utilization of free space. Based on these responses the leader decides whether the offers

received are sufficient and signals a completion code. The offers are processed in the order they were received, guaranteeing that the topologically closest slaves are selected in order to minimize latency. If the responses indicate a valid request, the leader then sends a binding message to the selected slaves, which completes the memory reservation. In response, the slaves acknowledge the request and provide a unique handle (which encodes the mote address) to the master. All subsequent requests must refer to this handle to access memory in the allocated area.

Read – There are two versions of the read command that differ in access granularity. A *readPage* call takes as input the page offset within a handle and a flag specifying whether the data is cacheable, and returns a buffer holding the data associated with that page. Its implementation determines whether the requested data exists in the cache: if the data is found in the cache, the request is satisfied locally, otherwise a lookup is performed in the page table to find the mote responsible for that page. Once the page is located, a request is sent to the remote mote to retrieve the page. A message indicating the result of the operation is returned by the slave mote. If the operation succeeded, the buffer holding the received page is returned, and, depending on the value of the "cacheable" flag, the data is inserted into the local cache (possibly evicting another entry, according to the cache replacement policy). Cache parameters and eviction policies may be selected by the application programmer to optimize the hit rate.

To provide a more flexible programming model we created a wrapper interface around *readPage* that takes as input an offset in the page to start reading from, the size of the read request (in bytes), a buffer where the data retrieved must be placed, and a flag indicating whether the data should be cached or not. Using this wrapper, a program can read any number of bytes, allowing the underlying implementation to determine exactly how many pages are needed to satisfy the request and perform the actual *readPage* operations.

Write – Similar to the read operation, a *writePage* function is provided that operates at the page level. Based on the caching policy employed, the cache is updated and a write request containing the page is sent to the mote responsible for it. The location of the page offset is determined by examining the index of the entry in the page table.

The *writeBytes* function is also provided for added convenience, and allows the programmer to specify the offset, the size (in bytes), and a flag indicating the caching policy preferred. Programmers seeking to optimize the library might define their own replacement policies and prefetch predictors to match memory access patterns.

Both read and write operations transfer data in blocks. Since a TinyOS message has a maximum payload size of 29 bytes, our block size is limited by this value and the overhead of our message header. Currently we use a paging model where the page size is fixed (equal to the block size minus header data), but we provide a discussion about variable size pages, their advantages and disadvantages, in the Future Work section.

Free – This operation works in a similar manner to *malloc*. It iterates through the page table entries to retrieve the host ids and remote indices allocated for that handle, and sends out messages to the respective motes indicating that they should free those specific memory areas. If any entries are also found in the local cache, they are evicted.
Upon receipt of a free request, a remote mote employs a similar algorithm that identifies the pages held by a particular handle, marking them as unused.

5. Implementation Details

We have created a prototype implementation of our virtual memory library (VM) as a NesC module that utilizes the radio to talk to similar master or slave modules in the network. Each VM module contains a page table, a cache and a buffer for holding temporary information received from a read request.

Page Table Format

Each page table consists of a set of page table entries. Handles to memory references are simply indices into the page table shown in Figure 3. Each line contains a host id, describing which mote holds the storage of the contents of pages managed under the handle. Page table entries may form a linked list using ‘next’ pointers that allows for variable-sized handles. A handle allocates memory in quanta of pages, which are listed in the cache id fields. Each page table entry contains a list of cache ids. There is one cache id per page in the handle. The list of cache ids is null-terminated. Each entry of the list either refers to an index in the cache that is currently holding the block, or contains a special value indicating that the cache does not contain the block. The cache id fields implicitly refer to a page by its index into the handle structure, delegating the task of locating the page to the slave node, or the cache. Since the actual management of handle memory is distributed to the slave mote, it is allowed to delegate contents of pages to other motes, forming a topology.

Cache Format

Caches are composed of cache lines of un-typed memory (Figure 4). Each line refers to the contents of a page. Cache validity is determined by checking if a handle in the page table refers to a cache entry. Care is taken by the library not to alias a cache line to multiple handles or pages. Caches may employ replacement policies described previously.

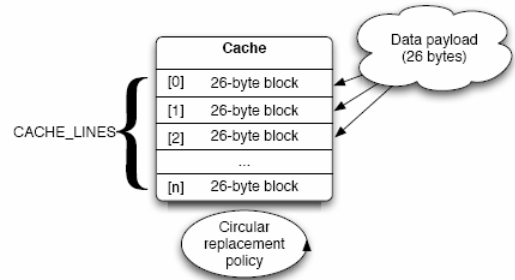
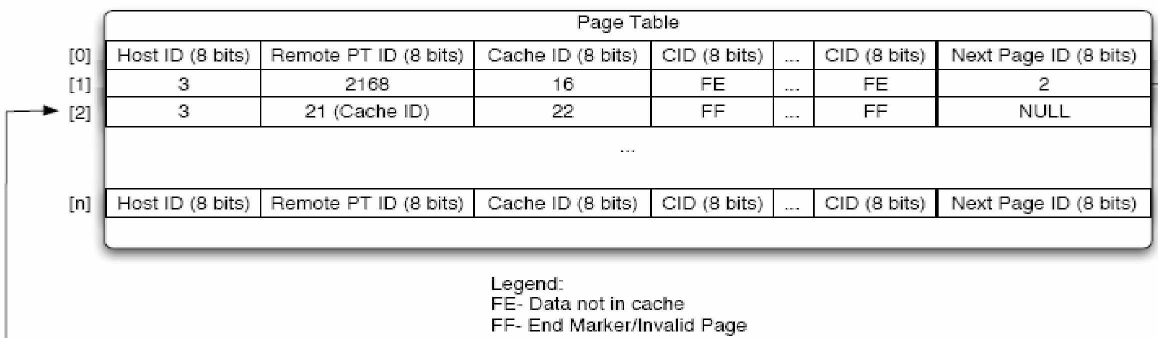


Figure 4 - Cache Diagram

Message Format

Messages are derived from TinyOS messages which allow for a 29 byte data payload. We divide this payload into two sections: the first section includes a common header containing information about the operation type, its return status, and the address of the mote sending the request or reply; the second section contains operation-specific headers providing details about the operation. Figure 5 and Figure 6 show details of the messaging format.

The size of a page is highly dependent on message size because we would like pages to fit into a single message for efficiency and performance reasons.



Page Table

[0]	Host ID (8 bits)	Remote PT ID (8 bits)	Cache ID (8 bits)	CID (8 bits)	...	CID (8 bits)	Next Page ID (8 bits)
[1]	3	2168	16	FE	...	FE	2
[2]	3	21 (Cache ID)	22	FF	...	FF	NULL
			...				
[n]	Host ID (8 bits)	Remote PT ID (8 bits)	Cache ID (8 bits)	CID (8 bits)	...	CID (8 bits)	Next Page ID (8 bits)

Legend:
FE- Data not in cache
FF- End Marker/Invalid Page

Figure 3 - Page Table Format

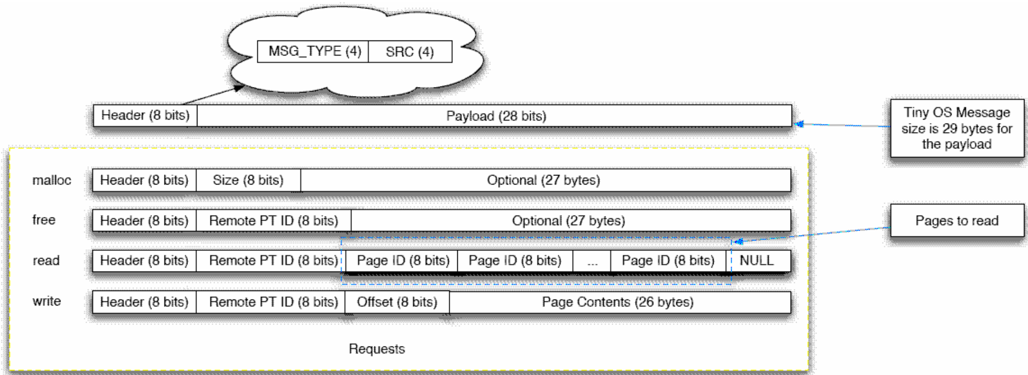


Figure 5 – Messaging Format - Requests

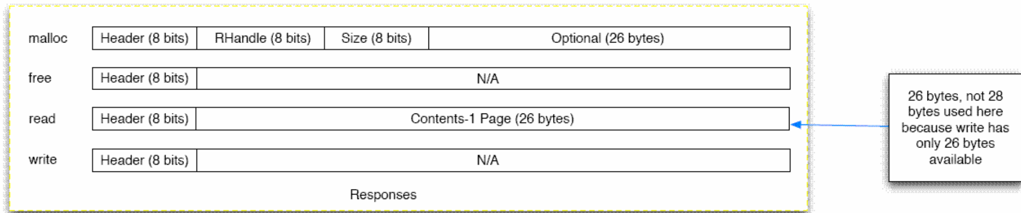


Figure 6 - Messaging Format – Replies

6. Evaluation

6.1 Simulation Results

To evaluate our prototype implementation we created a simple micro benchmark and a trace-driven simulator that makes use of the four defined operations, and calculated the number of messages exchanged between the master and slave motes, as well as the running time of each individual operation averaged over four trials. The pseudo code of our simple test case is described in Figure 7.

```
Task void Test() {
    handle=malloc(10);
    // wait for the
    // operationComplete signal
    write(handle, 0, "hello");
    // wait for the
    // operationComplete signal
    read(handle, 0); //read page 0
    // wait for the
    // operationComplete signal
}
```

Figure 7 - Test code

Table 1 shows the running time of each operation. Be aware that the TOSSIM simulation does not accurately simulate latency, events, or actual performance.

Trial	1	2	3	4	Avg.
MallocL	168	176	167	168	169.75
WriteL	106	109	94	112	105.25
ReadL	198	223	58	160	159.75

Table 1 – TOSSIM micro benchmarks with time scaling enabled (milliseconds)

MallocL, WriteL, and ReadL are measurements of the running time of the respective operations in TOSSIM on a Fujitsu Laptop (Pentium 4 1.8 GHz, 512 MB RAM running Fedora Core 3 Linux). The L means that TOSSIM was running with time scale equal to one (option -l = 1), which is supposed to emulate near real-time operation. Table 2 shows the same readings taken without time scaling.

Trial	1	2	3	4	Avg.
Malloc	150	150	121	152	143.25
Write	171	205	115	172	165.75
Read	71	71	56	73	67.75

Table 2 – TOSSIM micro benchmarks without time scaling enabled (milliseconds)

The average latency seems to be more indicative of the processing time of TOSSIM rather than any real metric. Reads seem to have taken the longest with time scaling on, but the shortest with time scaling off.

We attempted to use another emulator, ATEmu, which simulates TinyOS and the underlying processor on an instruction-by-instruction level. Tasks may be preempted using ATEmu, but TOSSIM does not model this behavior. Unfortunately, we were not able to obtain real performance numbers from ATEmu directly, because it lacked the ability to output debug messages.

In terms of the number of messages exchanged, the *malloc* operation takes four messages to inquire and contact the target node, while the read, write, and free take only two messages, for a total of ten messages.

6.2 TRACE-DRIVEN SIMULATION

From our initial tests, we conclude that the majority of latency in the library is due to radio transmissions, therefore instead of focusing on synthetic benchmarks obtained from a simulator, we wrote a trace-driven simulation of the effects of caching parameters. Since the algorithms and inputs are well-known, implementing the sort in C or TinyOS would make no difference in the memory access patterns. Our implementation generates traces from algorithms run in C on a desktop machine, and replays the same actions on a single master mote. The master then executes the logged operations on in conjunction with a single simulated slave node. Read and write are the only logged operations, and execution of the log is done at the full speed of the simulated mote. This approach allows us to test complex algorithms that may be too error-prone to implement on a real mote, while providing a reasonably accurate simulation.

We present our performance evaluation in Figures 8 through 11. The first figure shows cache hit rate plotted against block size. The graph shows that Bubble sort benefits most from caching, perhaps due to its myopic sorting methodology. Quicksort seems to benefit least because its access pattern is more random through use of the partition function. The most drastic performance gain from an increased block size is in quicksort, this is due to the ability to "see" a larger view of the array to be sorted at any given time. Selection sort and bubble sort benefit less from the larger window. Figure 9 shows the hit rate plotted against the number of cache lines. Not surprisingly, bubble sort has a higher hit rate than quicksort; however, its hit rate remains nearly constant independent of the number of cache lines. This is once again due to the myopic view. Quicksort quickly increases its hit rate with more cache lines because its access pattern is more spread. Figures 10 and 11 show the number of messages and the running times of our traces for various page sizes. Recall that the message payload size is equal to the page size, for simplicity.

These results show that there are diminishing returns with increasing page size for these particular test scenarios. To provide an assessment of the practicality of our approach, we perform two indirect measures. The first measure, number of messages, is meant to give a rough estimate of the power requirements of our library for a given task. In the worst case analysis, running bubble sort on 15 elements takes at most 575 messages over 46 seconds when forced to use the worst-case block size of 1 byte. According to CrossBow, a typical mote is capable of running for 172 hours with a packet sent every four seconds. Slightly extrapolating, a mote might be capable of slightly under 9288000 messages over the lifetime of the two AA batteries. Our application consumes a mere fraction of the available capacity of the device. If we were doing the sorting at the same rate as Crossbow's tests, it would take 38.3 minutes to send requests from the master, and a total of 77 minutes to complete execution.

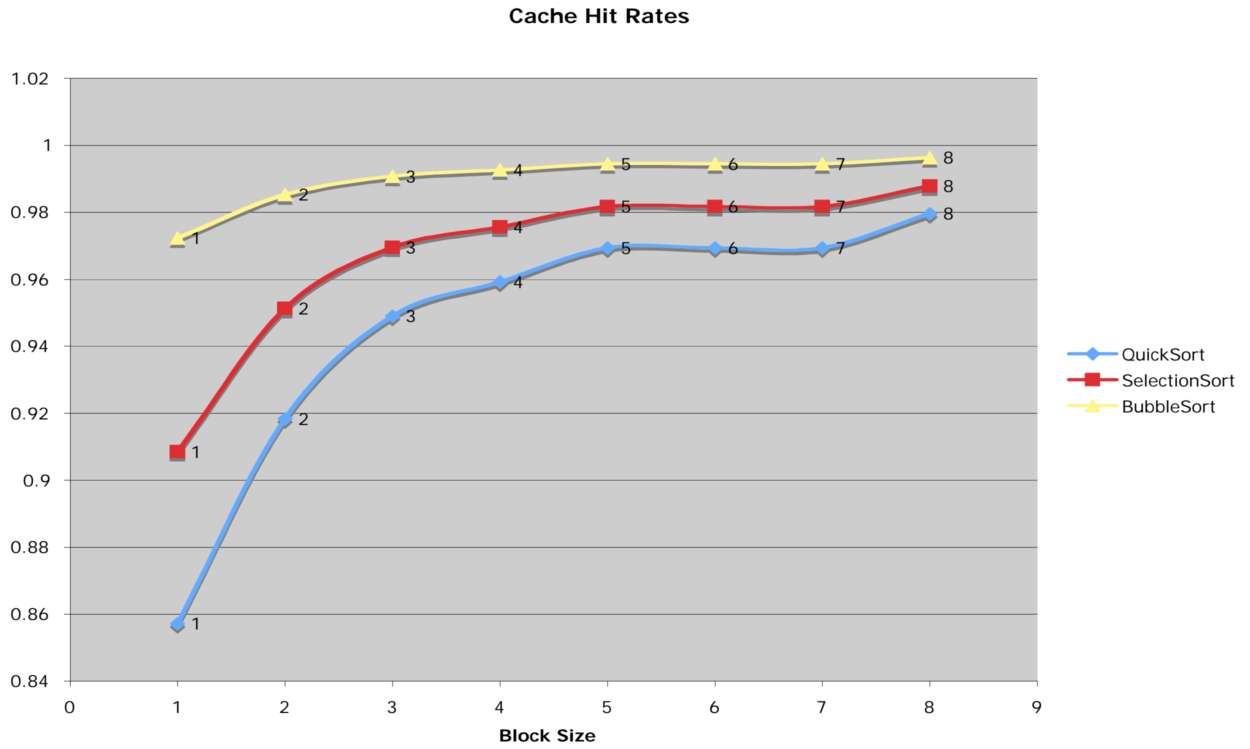


Figure 8 - Cache Hit Rate vs. Block Size

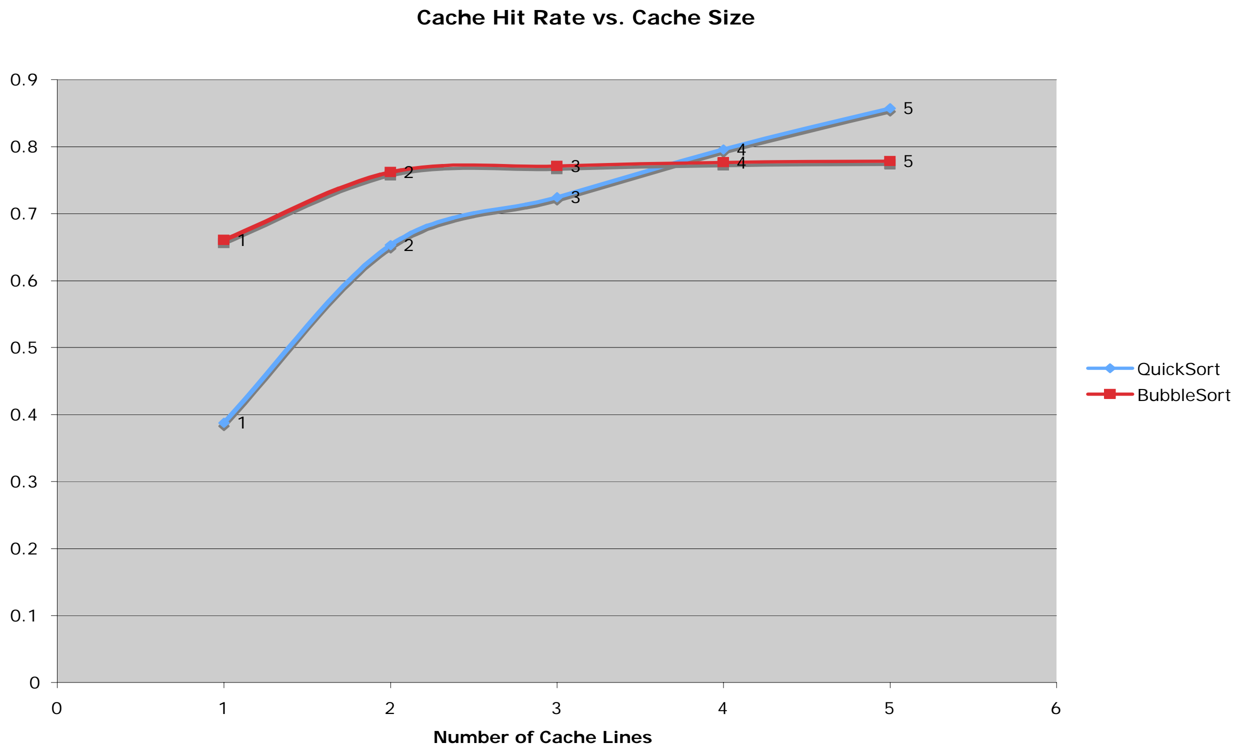


Figure 9 - Cache Hit Rate vs. Number of Cache Lines

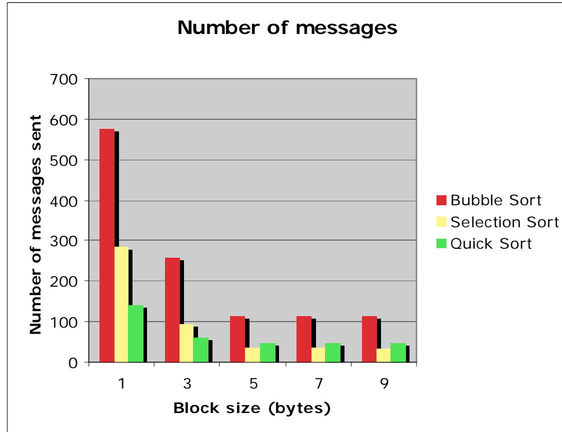


Figure 10 – Number of messages sent for the different trace simulations with varying block size

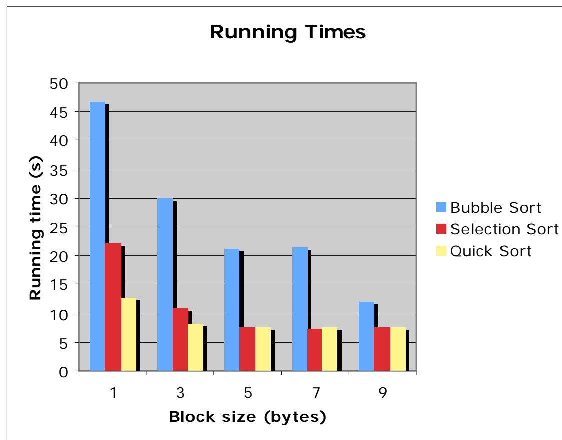


Figure 11 – Running times for the traces with different block sizes

Our second metric is running time in TOSSIM. We measure the running time of these traces using the UNIX *time* command. As the graphs show, the worst case performance is 46 seconds, a factor of improvement over the worst-case scenario of sending four packets every minute (as described in the CrossBow documentation).

7. ADVANCED FEATURES

Since our VM approach does not dictate a routing substrate, we have assumed a simple topology until now. This has resulted in a simplistic one-hop strategy that is suitable for applications which have small marginal requirements, and the need for high performance. One simple approach for expansion is to allow for partial memory allocations on each node. Such a scheme would alleviate contention, and better distribute data across the network, resulting in benefits due to parallel accesses. This is similar to a RAID0 striping arrangement. We see that the benefits multiply when the access medium is slow in nature, such as flash memory. The network may be several times faster than long-term storage on certain devices, especially when certain motes only wake up during specific time intervals.

To implement partial allocation, each slave node must use a reply message that indicates it wishes to allocate only a subset of the requested amount. Recall that our allocation procedure is two-phase. The first phase solicits replies from able servants, while the second phase commits to the allocation by sending an allocation commit message. Using this protocol, the solicitation phase may ask all nodes which have idle capacity to contribute, and then allocate based on partial results. Each page table entry contains a host id, remote page table id, a next pointer to another page table entry, and a list of cache entries. In the partial allocation scheme, each constituent node would occupy a page table entry, with values that are relevant to the node. Since the next pointer does not fix a particular host and handle id, this functionality may be used to reference different nodes and remote handle ids within the same local handle. The overall effect of this approach is that we can have an arbitrary mapping of remote memory locations on many nodes to a single local handle. As memory accesses increase, the local cache can ensure that many of the hits will incur low-cost.

We have currently implemented a subset of the features required for a partial allocation scheme. Our implementation uses a two-phase *malloc* as described above, and a set of wrappers around the block-based read and write operations. We have found that partial allocation allows the use of several effects. First, with partial allocation, large memory allocations become possible among a group of motes. If a fair allocation scheme is used to budget the size of each individual allocation, this can lead to hotspot reduction, and parallelism benefits described earlier.

We faced several implementation issues regarding our use of the TOSSIM simulator with partial allocation. Using just two motes, there is little chance of race conditions and packet loss. When partial allocation is used, many messages may be sent in response to a single read message, which can specify multiple blocks. Since TOSSIM is a discrete event simulator, it tends to have two negative effects. First, messages may be lost instead of queued when many responses are received. We remedy this by waiting for a reply on each request. Secondly, events may be delivered out of order. We correct this by carefully blocking the system execution until a correct reply is received. Both of these effects lead to a suboptimal messaging system where there is a lot of blocking time and extra messages sent. Had TOSSIM queued messages and guaranteed total ordering, these measures would not be necessary.

8. Possible Extensions

Our current design forms the necessary basis to address and manage virtual memory primitives in a small tightly-knit mote network. In the event that a computation requires greater resources, a developer has several options. Among the available choices is the option to convert the problem into a specialized distributed algorithm, such as those used by TAG and Maté. If generality is important, our system may be extended to provide marginal benefits in cases that are slightly larger. We describe possible extensions in our system in this section with focus on simplicity of the protocol.

One major improvement can be the inclusion of multi-hop neighbors. Using the same mechanism for partial allocations, a slave node may delegate a memory allocation to a second-level slave. Since the data structures on master and slave nodes are equivalent, this method requires minimal changes. Upon a memory access, a master accesses the page table index associated with a handle. The master then looks up the remote address, handle and offset to send to the slave mote. Upon receipt, the slave mote repeats the same steps until a leaf node is reached. Once a leaf is reached, the real address is located in the cache/backing store, and then the data is percolated back to the originating master.

Although this approach may seem slow, there are several techniques that may help. First, if the nodes allocate only in an increasing direction away from the master, due to lack of geographically closer viable nodes, then performance can be guaranteed by hop minimization. A simple approach would be to use a geographic hash table to initially set up allocation preferences. Secondly, a routing substrate which respects geographically and reliability-oriented nodes first can aide the allocation mechanism in making an optimal node choice.

Using the multi-hop scheme, it might be expected that frequent access to data structures may be slow. We debunk this myth by exemplifying through P2P file sharing systems. First, it is expected that our static allocation of data within a network limits flexibility of location. Secondly, long chains might damage reliability characteristics of data storage in event of a failure. While the former may seem true at a first glance, the actual case is that as data is accessed along the multi-hop route, each node caches the access. The overall effect is similar to that seen in Freenet, where files are migrated and replicated along access paths [10]. While we do not explicitly replicate, this has the effect of a multilevel cache, where local events on the primary cache have little effect on higher level caches, ensuring a high hit rate on second and third level caches. There may be much unnecessary cache duplication overhead, but we can combat this effect by using a probabilistic caching scheme, where the probability that a remote entry is cached is based on a parameter p. This allows long chains of length n to occupy approximately np space in the caches, greatly reducing the crowding effect. The latter argument relating to reliability may also be less severe than previously thought. Since our system is designed for short-term allocation, and many memory accesses have good cache locality through our multi-tier cache, the most important data is likely to be replicated many times in lower-level caches. This implies that a mote network using our system may have many leaf-

level caches, but fewer supernodes near the master. This may lead to a network that has properties similar to a power-law or exponential network. The random failure of a few nodes is likely to only affect leaf nodes, and most of the network remains intact, with the common data still fresh in the caches. While we do not yet have detailed simulation traces, we argue that the similarity in formation to existing networks, such as Gnutella offers us temporary immunity to some degrees of failure.

Another method to enhance fault tolerance is to borrow schemes from distributed file systems. In many distributed systems, there is the concept of a primary replica, which manages pointers and replication policies of specified data. We can use this approach to delegate responsibility of a shared page to a single slave mote. This slave can then request an identical allocation on another slave, and keep pointers to a secondary group of slaves, whom replicate the same data. Heartbeats among this group can guarantee liveness, and upon node failure, a new master may be selected. Protocols in distributed hash tables with group membership are particularly suited for this purpose.

Finally, our single-master approach may not be completely scalable for larger computations; we can partially alleviate this by having multiple master nodes. Since a memory reference is a tuple containing a node address and a handle, this access tuple may be passed around the network as a capability to other nodes such as slaves to aid in computation. Even though this does not provide any direct parallelization benefits, it does partially help address the issue of fragmentation of the global memory pool across all motes. One perspective to view our framework is to compare the memory allocation we offer against that of a network without the virtualization system. In such a network, there are a high percentage of motes with unused memory resources, and the fragmentation of these resources can inhibit overall operation. We allow users to partially recover reasonable-size chunks of these resources to continue computation.

9. Conclusions

In this paper we presented the *Beach system* as a first step towards the virtualization of resources provided by these small computing devices, called motes. We introduced a low overhead virtualization library to automatically manage remote memory on mote devices. Using a systems approach, our library breaks memory into user-specified fixed-sized quanta that are allocated by applications. Through several mechanisms such as partial allocation, we can reclaim memory wasted in a mote network due to fragmentation. This allows small sporadic computations, which have large temporary memory requirements, to run effectively. A caching system avoids excessive utilization of the radio, and allows motes to gain better access to most recently used material. We implemented our system and found that our caching scheme significantly reduces the number of radio transmissions for common sorting benchmarks.

By specially designing our page tables, we can indirect blocks between slave nodes. We can use this indirection technique to form a tree of participants whom contribute small amounts of memory to the network. Specialized master replicas can be designed to ensure fault tolerance in the system without significant performance penalties. Through the use of hierarchical allocation and master replicas, we can build a scalable and reliable system to redistribute memory in a mote network.

Our work represents an initial attempt at designing a system to handle efficient resource reallocation within a mote network. With partial allocation and caching, we already see a great improvement in the capabilities of motes as a viable platform for larger computations. Unlike TAG and in-network computation, our system can be used to run code that is difficult or impossible to transform into an in-network processing equivalent. This allows programmers to use traditional algorithms on mote networks without excess regard to resource considerations.

10. Ongoing Deployment Attempts

We have begun some initial testing with mote hardware from both Crossbow and Intel. At present, we do not have reliable measurements, but we expect to be able to obtain them soon. We had trouble with operating the radios on the Crossbow motes we received, so we opted to try out the next generation Intel motes. At the time of this writing we are early in the testing phase since we only received the new hardware very recently. So far we have experienced a number of problems with the build process, and found the lack of documentation on the iMote TinyOS Bluetooth radio interface model to be a significant shortcoming of these motes. We did have limited success with the demo applications and porting some of the basic TinyOS examples, such as CntToRfm. We are actively exchanging messages with other researchers on the TinyOS mailing lists and community resources to address the various issues we've encountered. Given that the iMotes are still in an early beta phase, maturity of the product over time may help to ease our efforts.

11. References

[1] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for linux clusters. In *Proceedings of Euro-Par'03 International Conference on Parallel and Distributed Computing*, Aug 2003.

[2] M. D. Flouris and E. P. Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing, Special Issue on I/O in Shared-Storage Clusters.*, 2(4):281-293, Jun 1999.

[3] E. Lattanzi, A. Acquaviva, A. Bogliolo. Proximity services supporting network virtual memory in mobile devices. In *Proceedings of the 2nd ACM international workshop on Wireless mobile applications and services on WLAN hotspots*, pages 119-126. ACM Press, 2004.

[4] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[5] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Bewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[7] P. Buonadonna, J. Hill, and D. Culler. Active Message Communication for Tiny Networked Sensors. *Infocomm*, 2001.

[8] P. Levis, D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Dec 2002.

[9] P. Levis, D. Gay, D. Culler. Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines. *In 6th Symposium on Operating Systems Design and Implementation*, 2004.

[10] Clarke, I., Sandberg, O., Wiley, B., Hong, T. Freenet: Distributed Anonymous Information Storage and Retrieval System. In *Lecture Notes in Computer Science*, 2001.

[11] J. Kubiatowicz, et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Architectural Support for Programming Languages and Operating Systems*, Nov 2000.

[12] Madden, S., et al. TAG: A Tiny AGregation service for ad-hoc sensor networks. OSDI, 2002.

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCOM 2000)*, Aug 200.

[14] B. Karp and H. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Mobile Computing and Networks (MobiCOM* 2000), pp. 243–254.

[15] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proceedings of the Hawaii International Conference on System Sciences*, Jan 2000.