MEMMU: Memory Expansion for MMU-Less Embedded Systems

LAN S. BAI, LEI YANG, and ROBERT P. DICK
Northwestern University

Random access memory (RAM) is tightly constrained in the least expensive, lowest-power embedded systems such as sensor network nodes and portable consumer electronics. The most widely used sensor network nodes have only 4 to 10KB of RAM and do not contain memory management units (MMUs). It is difficult to implement complex applications under such tight memory constraints. Nonetheless, price and power-consumption constraints make it unlikely that increases in RAM in these systems will keep pace with the increasing memory requirements of applications.

We propose the use of automated compile-time and runtime techniques to increase the amount of usable memory in MMU-less embedded systems. The proposed techniques do not increase hardware cost, and require few or no changes to existing applications. We have developed runtime library routines and compiler transformations to control and optimize the automatic migration of application data between compressed and uncompressed memory regions, as well as a fast compression algorithm well suited to this application. These techniques were experimentally evaluated on Crossbow TelosB sensor network nodes running a number of data-collection and signal-processing applications. Our results indicate that available memory can be increased by up to 50% with less than 10% performance degradation for most benchmarks.

Categories and Subject Descriptors: D.4.2 [**Storage Management**]: Virtual Memory; E.4 [**Coding and Information Theory**]: Data Compaction and Compression

General Terms: Design, Experimentation, Management, Performance

Additional Key Words and Phrases: Data compression, embedded system, wireless sensor network

ACM Reference Format:

Bai, L. S., Yang, L., and Dick, R. P. 2009. MEMMU: Memory expansion for MMU-less embedded systems. ACM Trans. Embedd. Comput. Syst. 8, 3, Article 23 (April 2009), 33 pages. DOI = 10.1145/1509288.1509295 http://doi.acm.org/10.1145/ 1509288.1509295

23

This work was supported in part by the National Science Foundation under awards CNS-0721978 and CNS-0347941 and in part by NEC Laboratories America.

L. S. Bai and R. P. Dick are currently affiliated with the University of Michigan. L. Yang is currently affiliated with Google.

Author's address: L. S. Bai, University of Michigan; email: lanbai@umich.edu.

DOI 10.1145/1509288.1509295 http://doi.acm.org/10.1145/1509288.1509295

1. INTRODUCTION

Low-power, inexpensive embedded systems are of great importance in applications ranging from wireless sensor networks to consumer electronics. In these systems, processing power and physical memory are tightly limited due to constraints on cost, size, and power consumption. Moreover, many microcontrollers lack memory management units (MMUs). Although the proposed techniques may be used in any memory-constrained embedded system without an MMU, this article will focus on using them to increase usable memory in sensor network nodes with no changes to hardware and with no or minimal changes to applications.

Many recent ideas for improving the communication, security, and in-network processing capabilities of sensor networks rely on sophisticated routing [Karlof and Wagner 2003], encryption [Ganesan et al. 2003], query processing [Gehrke and Madden 2004], and signal processing [Li et al. 2002] algorithms implemented on sensor network nodes. However, sensor network nodes have tight memory constraints. For example, the popular Crossbow MICA2, MICAz, and TelosB sensor network nodes have 4KB or 10KB of RAM, a substantial portion of which is consumed by the operating system (OS) (e.g., TinyOS [Gay et al. 2005] or MANTIS OS [Abrach et al. 2003]). Tight constraints on the cost and power consumption of sensor network nodes make it unlikely for the size of physical RAM to keep pace with the demands of increasingly sophisticated in-network processing algorithms.

In order to reduce cost, sensor network nodes typically avoid the use of dedicated dynamic random access memory (DRAM) integrated circuits; in extremely low-price, low-power embedded systems, RAM is typically on the same die as the processor. Unfortunately, it is not economical to fabricate the capacitors used for high-density DRAM with the same process as processor logic. As a result, static random access memory (SRAM) is used in sensor network nodes. Unlike DRAM, SRAM generally requires six transistors per bit and has high power consumption. Increasing the amount of physical memory in sensor network nodes would increase die size, cost, and power consumption. Some researchers have proposed addressing memory constraints using hardware techniques such as compression units inserted between memory and processor. However, such hardware implementations typically have difficulty adapting to the characteristics of different application data. Moreover, they would increase the price of sensor network nodes by requiring either additional integrated circuit packages or microcontroller redesign. Barring new technologies that allow inexpensive, high-density, low-power, high-performance RAM to be fabricated on the same integrated circuits as logic, sensor network applications will continue to face strict constraints on RAM in the future.

Software techniques that use data compression to increase usable memory have advantages over hardware techniques. They do not require processor or printed circuit board redesign and they allow the selection and modification of compression algorithms, permitting good performance and compression ratio (compressed data size divided by original data size) for the target application. However, software techniques that require the redesign of applications

are unlikely to be used by anyone but embedded systems programming experts. Unfortunately, most sensor network application experts are not embedded system programming experts. If memory expansion technologies are to be widely deployed, they should not require changes to hardware and should require minimal or no changes to applications. Motivated by the previously described observations, we propose a new software-based online memory expansion technique, named MEMMU, for use in wireless sensor networks.

The rest of this article is organized as follows. Section 2 summarizes related work and contributions. Section 3 provides a motivational scenario that illustrates the importance of the proposed technique. Section 4 describes the library and compiler techniques, optimization schemes, as well as the compression and decompression algorithms designed to automatically increase usable memory in sensor network nodes. Section 5 presents the experimental setup, describes the workloads, and discusses the experimental results in detail. Finally, Section 6 concludes the article.

2. RELATED WORK AND CONTRIBUTIONS

The proposed library and compiler techniques to increase usable memory build upon work in the areas of online data compression, wireless sensor networks, and high-performance data compression algorithms.

2.1 Software Virtual Memory Management for MMU-Less Embedded Systems

Choudhuri and Givargis [2005] proposed a software virtual memory implementation for MMU-less embedded systems based on an application level virtual memory library and a virtual memory aware assembler. They assume secondary storage (e.g., EEPROM or flash) is present in the system. Their technique automatically manages data migration between RAM and secondary storage to give applications access to more memory than provided by physical RAM. However, since accessing secondary storage is significantly slower than accessing RAM, the performance penalty of this approach can be very high for some applications. In contrast, MEMMU requires no secondary storage. In addition, its performance and power consumption penalties have been minimized via compile-time and runtime optimization techniques.

2.2 Hardware-Based Code and Data Compression in Embedded Systems

A number of previous approaches incorporated compression into the memory hierarchy for different goals. Main memory compression techniques [Tremaine et al. 2001] insert a hardware compression/decompression unit between cache and RAM. Data are stored uncompressed in cache, and are compressed online when transferred to memory. Main memory compression techniques are used to improve the system performance by providing virtually larger memory. Code compression techniques [Lekatsas et al. 2000] store instructions in compressed format in ROM and decompress them during execution. Compression is usually performed off-line and can be slow, while decompression is done during execution, usually by special hardware, and must be very fast. Code compression

techniques are often used to save space in ROM for embedded systems with tight resource constraints.

2.3 Software-Based Memory Compression

Compressed caching [Douglis 1993; Wilson et al. 1999] introduces a software cache to the virtual memory system. This cache uses part of the memory to store data in compressed format. Swap compression [Tuduce and Gross 2005] compresses swapped pages and stores them in a memory region that acts as a cache between memory and disk. The primary objective of both techniques is to improve system performance by decreasing the number of page faults that must be serviced by hard disks. Both techniques require backing store (i.e., a hard disk) when the compressed cache is filled up. In contrast, MEMMU does not rely on any backing store.

CRAMES [Yang et al. 2005] is an OS controlled, online memory compression framework designed for diskless embedded systems. It takes advantage of the OS virtual memory infrastructure and stores least recently used (LRU) pages in compressed format in physical RAM. CRAMES dynamically adjusts the size of the compressed memory area, protecting applications capable of running without it from performance or energy consumption penalties. Although CRAMES does not require any special hardware for compression/decompression, it does require an MMU. In contrast, MEMMU requires no MMU. MEMMU implements software memory management via its compile-time and runtime techniques and uses numerous optimizations to maintain performance. This capability is necessary for most sensor network nodes and low-cost embedded processors because the majority do not have MMUs.

Biswas et al. [2004] described a memory reuse method that relies upon static liveness analysis. It compresses live globals in place and grows the stack or heap into the freed region when they overflow. Their work aims at improving system reliability by resolving runtime memory shortage errors as a consequence of the difficulty in predicting the size requirement of dynamic memory objects such as stack and heap. In contrast, MEMMU tries to solve a different problem: permitting system operation when the lower bound on memory requirements already surpass physical memory. Therefore, MEMMU has a much bigger memory expansion ratio.

Cooprider and Regehr [2007] proposed an RAM compression technique that targets data elements that have values limited to small sets, which are determined using compile time analysis. In contrast, MEMMU uses online compression of data based on access patterns that are hard to determine at compile time. As a result, MEMMU can be applied to sensor data, generally permitting greater increases in usable memory. Note that Cooprider's and Regehr's technique, and MEMMU, are complementary; they compress different structures and do not significantly interfere with each other.

2.4 Compression for Reducing Communication in Sensor Networks

In many sensor network applications, sensor nodes in the network must frequently communicate with each other or with a central server. Sensor nodes

have limited power sources and wireless communication accelerates battery depletion [Pottie and Kaiser 2000]. In-network data aggregation [Madden et al. 2002; Guestrin et al. 2004] and data reduction via wavelet transform or distributed regression [Hellerstein and Wang 2004; Nath et al. 2004] can significantly reduce the volume of data communicated. However, these techniques are lossy, limiting their application. Recently, researchers have proposed to reduce the amount of data communication via compression [Pereira et al. 2003; Pradhan et al. 2002] in order to reduce radio energy consumption. Our work differs from theirs in that MEMMU focuses on automated memory compression for functionality improvement instead of communication reduction.

2.5 Software-Based Memory Compression Algorithms

LZO [Oberhumer] is a very fast general-purpose compression algorithm that works well on many types of in-RAM data. However, the memory requirement of LZO is at least 8KB, far exceeding the available memory of many low-end embedded systems and sensor nodes. Rizzo et al. [1997] proposed a software-based algorithm that compresses in-RAM data by only exploiting the high frequency of zero-valued data. This algorithm trades off degraded compression ratio for improved performance. Wilson et al. [1999] presented a software-based algorithm called WKdm that uses a small dictionary of recently-seen words and attempts to fully or partially match incoming data with an entry in the dictionary. Yang et al. [2006] designed a software-based memory compression algorithm for embedded systems named pattern-based partial match (PBPM). This algorithm explores frequent patterns that occur within each word of memory and exploits similarities among words.

Many software-based memory compression algorithms are not appropriate for use on sensor network nodes due to large memory requirements or poor performance. For those with sufficiently low overhead, we found none that provides a satisfactory compression ratio for sensor data. The main reasons for this follow:

(1) Zero words are rare in many forms of sensor data.
(2) Many forms of sensor data change gradually with time. As a result, adjacent data elements are often similar in magnitude but have very different bit patterns. Therefore, conventional dictionary-based compression does not work well. We evaluated a partial dictionary match algorithm [Yang et al. 2006] in this application. The compression ratio was much worse than delta compression. The partial dictionary match achieved an 86% compression ratio for trace data, while the proposed delta compression algorithm achieved a 50% compression ratio. We suspect that part of the cause for the poor performance of the dictionary-based algorithm was the high relative penalty for storing dictionary indices when 16-bit words are used; the algorithm performs well in another application in which 32-bit words are used.
(3) The block size used in compression is often restricted in low-cost MMU-less devices, as we will explain later.

We propose a memory compression algorithm that operates with very high performance on the 16-bit data generally found in the memory of MICAz and

TelosB sensor network nodes. The average compression ratio for various types of sensor data is approximately 50%.

2.6 Contributions

The proposed memory expansion technique, MEMMU, expands the memory available to applications by selectively compressing data that reside in physical memory. MEMMU uses compile-time transformations and runtime library support to automatically manage online migration of data between compressed and uncompressed memory regions in sensor network nodes.

MEMMU essentially provides a compressed RAM-resident virtual memory system that is implemented completely in software via compiler transformations and library routines. Its use requires no hardware MMU, and requires few or no manual changes to application software.

Our work makes four main contributions.

(1) It provides application developers with access to more usable RAM and requires no or minor changes to application code and no changes to hardware.
(2) It does not require the presence of an MMU and has other design features that enable its use in sensor network nodes with extremely tight memory and performance constraints.
(3) It has been optimized to minimize impact on performance and power consumption; experimental results indicate that in many applications, such as data sampling and audio signal correlation computation, its performance overhead is less than 10%.
(4) We have released MEMMU for free academic and nonprofit use [MEMMU].

MEMMU was evaluated on TelosB wireless sensor network nodes. The TelosB is an MMU-less, low-power, wireless module with integrated sensors, radio, antenna, and an 8MHz Texas Instruments MSP430 microcontroller. The TelosB has 10KB RAM and typically runs TinyOS.

3. MOTIVATING SCENARIO

In this section, we describe a motivating scenario that illustrates the purpose and operation of MEMMU. Consider an application in which individual sensor nodes react to particular events (e.g., low-frequency vibration) by triggering high-rate audio data sampling. After the sampling is complete, data are filtered and statistics (e.g., variance and mean) are computed and transferred to an observer node. If the raw data are of interest to the observer node, they are requested and transmitted through the network. In existing sensing architectures, the size of the data buffer is tightly constrained. For example, on a Crossbow TelosB sensor node a maximum of 9.5KB RAM is available for buffering. Moreover, sampling rate and duration cannot be increased without redesigning the sensor node hardware or increasing the complexity of application implementation. If, instead, the automated data compression technique proposed in this article is used, portions of sampled data will be automatically compressed whenever they would otherwise exceed physical memory. During filtering (e.g.,

convolution) data are automatically decompressed and recompressed to trade off performance and usable memory. Commonly-accessed data are cached in uncompressed format to maintain good performance. This is achieved without changes to hardware and with no or minimal changes to application code. To the application designer, it appears as if the sensor network node has more memory than is physically present.

Many wireless sensor networks use a store-and-forward technique to distribute information. Therefore, the local memory of a node is used as a shared resource to handle multiple messages traveling along different routes. In order to avoid losing data during communication, a node must generally store already-sent data until it receives an acknowledgment. As a result, the buffer can easily be filled when the communication rate is high, leading to message loss or even network deadlock. With MEMMU, usable local memory can be increased thus reducing the probability of data loss.

4. MEMORY EXPANSION ON EMBEDDED SYSTEMS WITHOUT MMUS

This section describes the design of MEMMU, our technique for memory expansion on embedded systems without MMUs. The main goal of MEMMU is to provide application designers with access to more usable RAM than is physically available in MMU-less embedded systems without requiring changes to hardware and with minimal or no changes to applications. We achieve this goal via online compression and decompression of in-RAM data. In order to maximize the increase in usable RAM and minimize the performance and energy penalties resulting from the technique, it is necessary to solve the following problems:

(1) Determine which pages to compress and when to compress them to minimize performance and energy penalties. This is particularly challenging for low-end embedded systems with tight memory constraints and without MMUs.
(2) Control the organization of compressed and uncompressed memory regions and the migration of data between them to maximize the increase in usable memory while minimizing performance and energy consumption penalties.
(3) Design a compression algorithm for use in embedded systems that has low performance overhead, low memory requirements, and a good compression ratio for data commonly present in MMU-less embedded systems. For example, data sensed, processed, and communicated in sensor network nodes such as audio samples, light levels, temperatures, humidities, and, in some cases, two-dimensional images.

MEMMU divides physical RAM into three regions: the reserved region, the compressed region, and the uncompressed region. The reserved region is used to store uncompressed data of the OS, data structures used by MEMMU, and small data elements. The compressed region and the uncompressed region are both used by applications. Application data are automatically migrated between the compressed and the uncompressed regions. The size of each region is decided by compile-time analysis of application memory requirements and estimated compression ratio. The compressed region can be viewed as a high capacity but

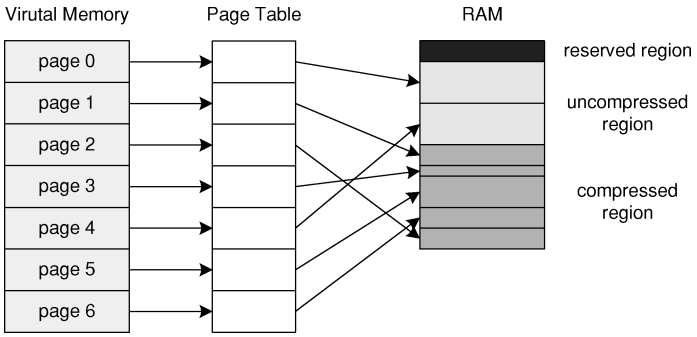

Fig. 1. Memory layout.

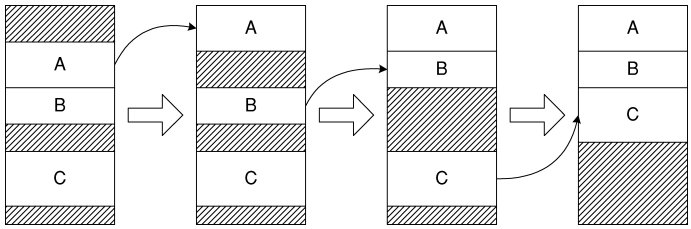

Fig. 2. Memory coalescing.

somewhat slower form of memory, and the uncompressed region can be viewed as a small, high-performance data cache.

Figure 1 illustrates the memory layout of an embedded system using MEMMU. From the perspective of application designers, all memory in the left-most Virtual Memory column is available. Virtual memory is broken into uniform-sized regions called pages. These pages are mapped to the uncompressed or compressed region (shown to the right of Figure 1) via a software-maintained page table. The page number is used as an index into the page table. A memory management mechanism was designed to manage data compression, decompression, and migration between the two regions.

4.1 Handle-Based Data Access

Data elements are accessed via their virtual address handles. The virtual page number of a corresponding virtual address is obtained by dividing the virtual address by the page size. The mapping from virtual page to RAM is stored in a page table maintained as an array. For example, if the content of index n in the array is m, and m is in the range of uncompressed pages, virtual page n is mapped to page m in the uncompressed region. If m is greater than number of uncompressed pages, n is mapped to a page in the compressed region.

When data are accessed via their virtual addresses within an application, MEMMU first determines the status of the corresponding virtual page based on the page table.

(1) If the virtual page maps to an uncompressed page, the physical address can be directly obtained by adding the offset to the address of the uncompressed page. The data element is then accessed via its physical address.

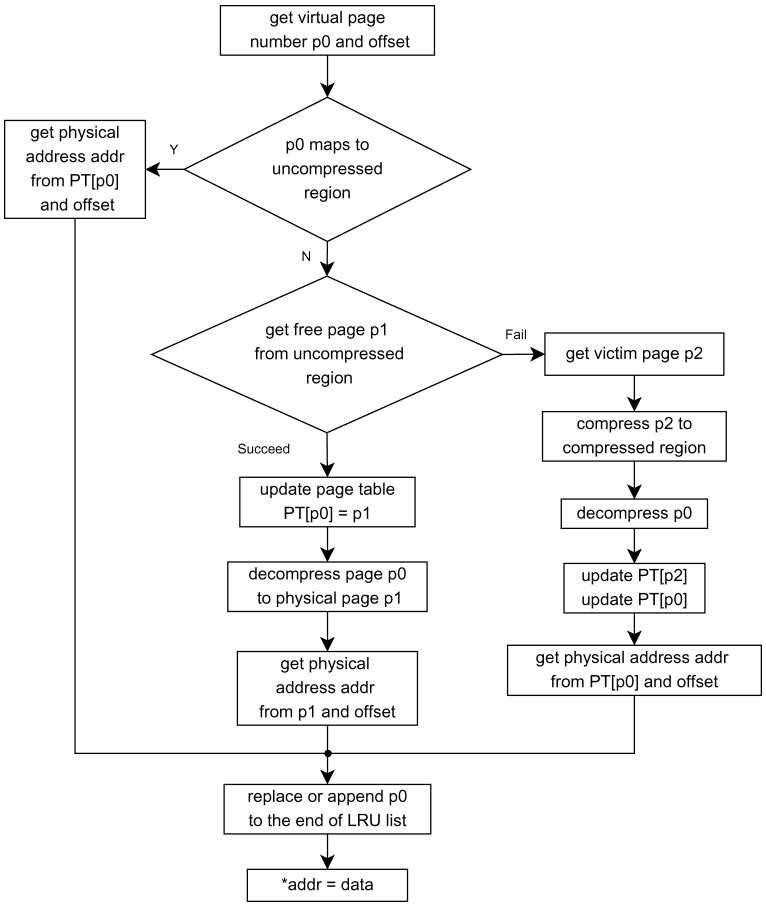

Fig. 3. Write_handle procedure.

(2) If the virtual page has not been accessed before (i.e., no mapping has yet been determined for the virtual page) a mapping from this page to an available page in the uncompressed region is created. If there is no available page in the uncompressed region, a victim page is moved to the compressed region to make an uncompressed page available.
(3) If the virtual page maps to a compressed page, the page is decompressed and moved to the uncompressed region. Again, if there is no available page in the uncompressed region, a victim page is moved to the compressed region to make space for an uncompressed page available.

In order to make the procedure transparent to users, and to avoid increasing application development complexity, the routines for these operations are stored in a runtime library and compiler transformations are used to convert memory accesses within unmodified code to library calls. Figure 3 illustrates the `write_handle` procedure. The three vertical paths prior to the final store instruction correspond to the situations discussed previously. The left path shows

the case in which a virtual page *p0* maps to a page *PT[p0]* in the uncompressed region. Its physical address is computed by adding offset to the physical page address. In the other two paths, virtual page *p0* maps to a compressed page. More specifically, in the middle path, a free page *p1* is available in the uncompressed region. The compressed page is decompressed to *p1* and a mapping from *p0* to *p1* is created in the page table. Otherwise, if the uncompressed region is full, as shown in the right path, a victim page *p2* from the uncompressed region is compressed. In that case, the physical page previously used by *p2* is freed and is now used to store decompressed *p0*. Finally, *p0* is mapped to a physical page in the uncompressed region and data are written to the physical address.

4.2 Memory Management and Page Replacement

When the uncompressed memory region is filled by an application, its pages are incrementally moved to the compressed region to make space available in the uncompressed region. When data in the compressed region are later accessed, they are decompressed and moved back to the uncompressed region. Ideally, pages that are unlikely to be used for a long time should be compressed to minimize the total number of compression and decompression events. MEMMU approximates this behavior via an LRU victim page selection policy. The LRU list is doubly linked. Every item in the LRU list stores the associated virtual page handle. Handles are ordered by the sequence of handle references. When a page that is already in the LRU list is accessed, it is relocated to the tail of the list, otherwise the new page is appended to the list. The page at the head of the LRU list is selected for compression. After a victim page is compressed, the corresponding node is removed from the LRU list. Therefore, page handles in the LRU list indicate pages currently residing in the uncompressed region.

Managing the uncompressed memory region is straightforward since pages have uniform sizes. On the contrary, managing the compressed region is complex since page sizes differ. Dynamic memory allocation is used in the compressed region to permit the immediate reuse of space when a page is decompressed and moved back to the uncompressed region. Compressed memory management is akin to heap management. It imposes memory overhead for keeping information such as page sizes and addresses (refer to Section 5 for MEMMU's memory overhead). This overhead is important in embedded systems that contain only a few kilobytes of RAM. We use the best fit policy, which allocates the smallest free slot equal to or larger than the required size. Best fit tends to produce the least fragmentation and minimizes the performance overhead resulting from splitting and merging free slots. Pages that are moved from the compressed region to the uncompressed region to read data and returned to the compressed region without changes have the same compressed size. As a result, they can often be returned to their prior locations in the compressed region, in which they fit exactly. In this case, no free slot merging or splitting will occur. Though best fit needs to scan the whole free slot list, the performance overhead is low because the number of free slots, which is upper-bounded by the number of compressed pages, is small.

4.3 Preventing Fragmentation

Fragmentation is frequently a problem for dynamic memory allocation techniques. Fragmentation can prevent a newly compressed page from fitting in the compressed region, even though the total available memory in that region is sufficient. This situation has the potential to terminate application execution. MEMMU performs memory merging and coalescing to prevent fragmentation.

Free block merging takes place every time a page is decompressed and removed from the compressed region. Free block handles are maintained in a list in order of the physical address of the compressed areas. If a free block is adjacent to its predecessor or successor, these adjacent blocks are merged. This is a well-known memory management technique.

Coalescing occurs when the memory allocator fails to allocate a new block from the free list. In this case, MEMMU locates pages in order of increasing addresses and moves them to the top of the compressed region, or to the bottom of the most-recently moved pages. This process continues until all compressed pages have been moved. Upon completion, a single large free region remains. Figure 2 illustrates this procedure. Rectangles A, B, and C represent three compressed pages and shaded rectangles represent freed blocks. Initially, a request for a size a little bigger than the first free block cannot be satisfied because these free blocks are not continuous. After three iterations of moving A, B, and C upward, all freed blocks are merged into one big free block, and the requested block can be allocated from the big free block. This coalescing algorithm has a time complexity of $\mathcal{O}(n^2)$, where n is the total number of compressed pages. However, since in practice n is usually small, the cost of coalescing is low. For example, a TelosB mote with 10KB RAM and a page size of 256bytes has 40 pages of RAM. In addition to the three pages used for the reserved region (one page used by the operating system and two pages used by MEMMU), it may need 18 compressed pages ($n = 18$) and 19 uncompressed pages to expand the usable memory by $(18/0.5 + 19)/(40 - 1) - 1 = 41\%$. Note that coalescing never imposes a performance penalty unless it is the only remaining alternative permitting the allocation of needed memory. It improves usable memory size for multiple benchmark applications.

4.4 Interrupt Management

The primary target platform for MEMMU is wireless sensor network nodes, which are typically memory-constrained, MMU-less embedded systems. On sensor nodes, hardware interrupts often take place when newly-sensed data arrive. There are two naive approaches to handle interrupts during page misses: (1) disable them when accessing data in memory or (2) allow interrupts at any time. Unfortunately, the first approach would result in interrupt misses when interrupts occur during page misses; the second approach is also dangerous because any access to a page in the compressed region during the execution of an interrupt service routine triggered during a page miss would result in an inconsistent compressed region state. In this section, we describe the potential for missed interrupts in more detail and propose a solution.

Consider an environmental data sampling application in which missing samples is not acceptable. Although the optimization techniques described in

Section 4.5 can be used to reduce the overall execution time overhead, they cannot reduce the worst-case data access delay. In the worst case, the pages of data (except the control data structures stored in the reserved region) referenced in the sampling event handler are all in the compressed region, but there is neither available space in the uncompressed region to decompress these pages nor space in the compressed region to compress a victim page. In this situation, coalescing, compression, and decompression must be performed before each data reference, that is,

$$worst\_case\_delay = N \times (t_{coalesce} + t_{comp.} + t_{decomp.}) \quad (1)$$

where the t values are durations and N is the number of memory references in the sampling event handler. For most applications, the action taken on a sampling interrupt is merely storing the sensed data. Other tasks are posted to process the data later. Therefore, the interrupt handler only has one memory reference that may point to the compressed region. The worst-case coalescing time is encountered when all blocks in the compressed region must be moved upward. This latency can be bounded by the time required to copy the entire compressed region plus the time required by the coalescing algorithm. We measured the worst-case delay on a TelosB wireless sensor node described in Section 2.6, assuming the compression algorithm introduced in Section 4.6 is used. The time required to compress and decompress one 256byte page is 3.2ms. The worst-case coalescing delay on a TelosB mote with a compressed region of 20 pages is 15.7ms. MEMMU should only be used for applications in which the worst-case delay does not violate any hard timing constraints. If the data set accessed in the interrupt handler is small, this delay can be avoided by storing this data set in the reserved region. This is normally the case because the data set is generally a small buffer.

In applications that compute only in response to sampling events, samples will not be missed if the sampling period is longer than the worst-case compression and decompression delay triggered by a sampling event. However, constraining sampling rate is not always an acceptable solution because some applications may require high sampling rates and even infrequent events may occur during a page miss. To solve this problem, a ring buffer may be used. The ring buffer sits in the reserved memory region. When data arrive, they are immediately stored in the ring buffer and a `process_rbuf` task is posted, which moves older data in the ring buffer to the sample buffer. This technique prevents data that arrive during page misses from being dropped. The ring buffer should be large enough to hold the longest-possible sequence of missed samples. Our experiments indicate that an application sampling at 19,600bps (i.e., 2,450 sample per second) requires a ring buffer of at most 20bytes. The use of a ring buffer for high-frequency sampling applications is the only portion of the proposed design flow that requires (minor) changes to user application code. Note that MEMMU does not require the use of a ring buffer when sampling rate is low or when missing some samples is acceptable. MEMMU provides a ring buffer as a convenient and low-overhead method of preventing missed interrupts when necessary. In order to use a ring buffer, one needs to set the ring buffer length based on estimated worst-case delay, insert the `write_rbuf`

function call, and post the `process_rbuf` task to transfer data from the ring buffer to the application data structure.

4.5 Optimization Techniques

In previous sections, we described the basic design components of the MEMMU memory expansion system. With basic, unoptimized MEMMU, every memory access requires

(1) A runtime handle check to determine whether the address being accessed is in the uncompressed region;
(2) Compression and decompression if the address is not in the uncompressed region;
(3) An update to the LRU list; and
(4) Virtual to physical address translation, which includes reading the physical page number from the page table, and operations such as shift and add.

This introduces high execution time overhead that is proportional to the total number of memory accesses. Hence, the basic software virtual memory solution is not practical for many real applications on embedded systems. However, optimization techniques can be used to significantly reduce the number of runtime checks, LRU list updates, and address translations. In this section, we describe several such compile-time optimization techniques. Many of these optimizations are related to existing compiler analysis and loop transformation work [Muchnick 1997; Banerjee 1993; Mckinley et al. 1996]. The proposed optimization techniques are based on the analysis of explicit array access. This will pose no problem for most sensor networking applications. For example, almost all of the contributed applications in the TinyOS source repository use explicit array access. These applications were contributed by numerous research and industry teams. If applications include implicit array accesses via pointers, existing compiler techniques could be used to transform them to explicit accesses [Franke and O'Boyle 2001; van Engelen and Gallivan 2001]. This compiler transformation is not currently supported by LLVM. However, it would be trivial to use such a compiler pass in MEMMU if it becomes available.

(1) *Small object optimization.* If a small data element is used very frequently in the application, it should be assigned to the reserved region at compile-time to eliminate all related handle checks and address translations. The increase in usable memory resulting from allowing the migration of small globals, such as scalars, is generally not sufficient to offset the cost of managing their migration. For example, in the image convolution application shown in Figure 4(a), the small matrix of coefficients, K, is accessed in every iteration of the loop (line 8) and the size of this matrix is small. After moving it to the reserved region, we can eliminate $(W-M+1)\times(H-M+1)\times M\times M$ runtime checks and address translations related to this matrix. Using a reserved region also prevent infrequently used data from occupying the uncompressed region because they are stored in the same page with frequently referenced data. The small object optimization is implemented by modifying

```
Input: 2-D array A[H, W]
Input: 2-D array K[M, M]
Output: 2-D array B[H − M + 1, W − M + 1]
 1: n ← Σ_{p=0}^{M−1} Σ_{q=0}^{M−1} K_pq
 2: for i ∈ {0 ··· H − M} do
 3:   for j ∈ {0 ··· W − M} do
 4:     t ← 0
 5:     for a ∈ {0 ··· M − 1} do
 6:       for b ∈ {0 ··· M − 1} do
 7:         p ← A[i + a][j + b]
 8:         q ← K[a][b]
 9:         t ← t + p × q
10:       end for
11:     end for
12:     B[i][j] ← t/n
13:   end for
14: end for
```

(a)

```
Input: array A allocated by vm_malloc(H × W)
Input: 2-D array K[M, M]
Output: array B allocated by
    vm_malloc((H − M + 1) × (W − M + 1))
 1: n ← Σ_{p=0}^{M−1} Σ_{q=0}^{M−1} K_pq
 2: for i ∈ {0 ··· H − M} do
 3:   Bring in pages to be used in the following
      loop
      to uncompressed region
 4:   for j ∈ {0 ··· W − M} do
 5:     t ← 0
 6:     for a ∈ {0 ··· M − 1} do
 7:       for b ∈ {0 ··· M − 1} do
 8:         p ← read_handle(A + (i + a) × W +
            j + b)
 9:         q ← K[a][b]
10:         t ← t + p × q
11:       end for
12:     end for
13:     write_handle(B+i×(W−K+1)+j, t/n)
14:   end for
15: end for
```

(b)

Fig. 4. Example of (a) original and (b) transformed convolution application.

LLVM [Lattner and Adve 2004] to allocate all data structures smaller than a threshold in the reserved region since their sizes add up only to a few percent of the memory required by the application.

(2) *Runtime handle check optimization.* This technique is based on the observation that if a sequence of memory references access the same page, only the first handle check is necessary since the referenced page is sure to be in the uncompressed region on subsequent accesses. This optimization is specific to sequential access patterns, although different increment and decrement offsets are supported. By inserting checks to decide whether the data element to be accessed next is in a different page from the previous one, the number of handle checks for all accesses to the same page can be reduced to one. Performance is improved because the inserted check is relatively faster than reading an element from the array (page table). This can be especially useful for a hardware-triggered sample arrival event that writes data into the buffer, as illustrated by Figure 6. *Data_ready* is a hardware-triggered event. The *if* statement in the optimized code in Figure 6(b) filters all the handle checks mapping to the same page that was checked in the previous reference.

The runtime handle check optimization takes place in a compiler pass, in which LLVM creates two global variables, current page number and previous page number, for each `check_handle` and puts every `check_handle` call in an *if* statement. `Check_handle` is called only when the current page number differs from the previous page number. This technique may introduce overhead in some applications, such as an application that accesses interleaved pages, because the current page number will always be different from the previous page number. Therefore, it is only applied to programs or sections of code that access one array with affine function of induction

```
Variable: array A[N]
1: for i ∈ {0 ··· N} do
2:     A[i] ← x
3: end for
```

(a) Original code.

```
Variable: array A allocated by vm_malloc(N)
1: for i ∈ {0 ··· N} do
2:     check_handle((A + i)/PAGESIZE)
3:     write_handle(A + i, x)
4: end for
```

(b) Transformed code without optimization.

```
Variable: array A allocated by vm_malloc(N)
1: pnum ← A/PAGESIZE
2: for i ∈ {A/PAGESIZE ··· (A + N)/PAGESIZE} do
3:     check_handle(pnum)
4:     for j ∈ {0 ··· PAGESIZE} do
5:         write_handle(A + i × PAGESIZE + j, x)
6:     end for
7:     pnum + +
8: end for
```

(c) Transformed code with loop transformation.

```
Variable: array A allocated by vm_malloc(N)
1: pnum ← A/PAGESIZE
2: for i ∈ {A/PAGESIZE ··· (A + N)/PAGESIZE} do
3:     check_handle(pnum)
4:     base_ptr ← virtual_to_physical(A + i × PAGESIZE)
5:     for j ∈ {0 ··· PAGESIZE} do
6:         *base_ptr ← x
7:         base_ptr + +
8:     end for
9:     pnum + +
10: end for
```

(d) Transformed code with loop transformation and pointer dereferencing.

Fig. 5. Example of optimizations on an array accesses.

variables. Affine functions represent vector-valued functions of the form $f(x_1, ..., x_n) = A_1x_1 + ... + A_nx_n + b$.

(3) *Loop transformation and compile-time elimination of inner-loop checks.* This optimization scheme further reduces runtime handle checks via compile-time loop transformations. It may be applied to loops whose array accesses are affine functions of enclosing loop induction variables. Figure 5(a) illustrates an example of sequential references to an array. At most, *PAGESIZE* references access the same page. Figure 5(b) illustrates the unoptimized solution, which inserts a handle check before every memory reference (line 2) and replaces writes to memory with calls to the `write_handle` routine (line 3). The entire loop requires N handle checks. Figure 5(c) illustrates an optimized solution. Loop transformation is used to break the original loop into nested loops. Iterations of the inner-loop (line 4) access memory inside a single page. Therefore, handle checks for the inner loop can be replaced by one check in the outer loop (line 3). The total number of handle checks is reduced from N to $N/$ *PAGESIZE*. For the sake of simplicity, array *A* shown in Figure 5 is page-aligned. This loop transformation is a type of loop tiling [Muchnick 1997].

The loop transformation technique can also be applied in the following, more general, circumstances.

(a) The loop accesses only one array and the offset is a linear function of the loop induction variable. In the transformed code, every exit from the inner loop implies that the next accessed address is in a different page. When *PAGESIZE* is evenly divided by the stride, the number of iterations of inner loop is constant: *PAGESIZE* divided by stride. However, the number of inner-loop iterations varies if the *PAGESIZE* is not evenly divided by the stride. In that case, variables *start* and *end*

```
1: check_handle(buf + count)
2: write_handle(buf + count, data)
3: count + +
```

(a) Original code.

```
1: cur_page ← (buf + count)/PAGESIZE
2: if cur_page ≠ last_page then
3:     check_handle(cur_page)
4: end if
5: write_handle(buf + count, data)
6: count + +
7: last_page ← cur_page
```

(b) Transformed code with runtime handle check optimization.

Fig. 6. Example code transformation of data_ready(data) function.

```
Variable: array A[M]
1: for i ∈ {0 · · · N} do
2:     A[i × a + b] ← x
3: end for
```

(a) Original code.

```
Variable: array A allocated by vm_alloc(M)
1: for i ∈ {0 · · · N} do
2:     page ← (A + i × a + b)/PAGESIZE
3:     check_handle(page)
4:     write_handle(A + i × a + b, x)
5: end for
```

(b) Transformed code without optimization.

```
Variable: array A allocated by vm_alloc(M)
1: t ← A + b
2: p_min ← (A + b)/PAGESIZE
3: p_max ← (A + a × N + b)/PAGESIZE
4: for page ∈ {p_min · · · p_max} do
5:     check_handle(page)
6:     for j ∈ {start · · · end} do
7:         write_handle(t, x)
8:         t ← t + a
9:         j ← j + a
10:    end for
11: end for
```

(c) Transformed code with loop transformation.

Fig. 7. Loop transformation on sequential memory access with constant stride.

are used to control the iteration count for the inner loop by locating the offset in the referenced page at the beginning or end of the inner loop. Example code is shown in Figure 7. *Start* is calculated via modular division of the first address by *PAGESIZE*; *end* is obtained via modular division of the largest address by *PAGESIZE* for the last iteration and by *PAGESIZE* for other iterations.

(b) The loop accesses n arrays with the same stride, and $2 \times n - 1$ is no larger than the number of pages in the uncompressed region m. Figure 10 shows how a loop accessing arrays A, B, and C is transformed. The numbers in the arrays correspond to virtual page indices. The original loop carries out interleaved accesses to these arrays, from the top to the bottom. The loop is divided based on the page boundaries in the array in which a page boundary is first crossed. The arrows beside array C indicate iterations of the transformed loop. The numbers to the right of the arrows are the pages brought into the uncompressed region before each iteration. For example, at the beginning of third iteration, pages 2, 8, and 14 are brought into the uncompressed region. Pages 7 and 13 should not be compressed because they will be accessed during the second iteration. The dashed box in Figure 10 indicates all of the pages accessed during one iteration. Clearly, regardless of the vertical position of the box, it can overlap at most $2 \times (n-1) + 1$ pages. Therefore, this is the maximum number of pages required in the uncompressed region.

(c) If the loop accesses multiple arrays with different strides, only perform transformation on the arrays that meet conditions (a) or (b).

(4) *Handle check hoisting.* Hoisting handle checks is the process of replacing multiple handle checks inside a loop with one handle check outside the loop. This optimization requires that the total size of the accessed pages is no larger than the size of the uncompressed region. It can be viewed as prefetching pages before entering the loop and locking them in the uncompressed region until an iteration of the loop finishes execution. The smallest and largest addresses accessed for each memory object during one iteration are obtained and the largest possible number of pages between them is computed. Figure 4 gives an example of handle check hoisting. Figure 4(a) is the original code for image convolution. Without handle check hoisting, MEMMU requires $(H - M + 1) \times (W - M + 1) \times (2 \times M \times M + 1)$ handle checks. It can be decided at compile-time that the second inner loop (line 3), which covers three rows of A and one row of B, is the largest loop that can reside in the uncompressed region. Therefore, handle checks are hoisted to the beginning of the second inner loop, as shown in Figure 4(b) line 3. This eliminates at least $(H-M+1)\times(W-M+1)\times(2\times M\times M+1)-(H-M+1)\times 4$ handle checks. Note that at most four pages may be covered in the second loop, two for each array. To maximize performance while maintaining correctness, we start from the innermost loop, and expand outward until the analyzed memory usage in the next loop cannot be accommodated in the uncompressed region or we reach the outermost loop.

(5) *Pointer dereferencing to reduce address translation.* The purpose of the pointer dereferencing optimization is related to that of strength reduction optimizations [Muchnick 1997]: replacing expensive operations with less expensive operations. In particular, it replaces calls to `write_handle` and `read_handle` functions that contain complicated operations for address translation to pointer dereferencing with simple pointer computations. Assume the accessed virtual address is an affine function of a basic induction variable i: $a \times i + b$, a and b are constants. The physical address of the memory reference in question is $phy\_addr = PT[(A + a \times i + b)/PAGESIZE] + (A + a \times i + b)\%PAGESIZE$. $PT[(A + a \times i + b)$ computes the starting address of the physical page, $(A + a \times i + b)\%PAGESIZE$ computes the offset in the page. Normally, this operation cannot be optimized by general strength reduction optimizations. However, if we know that the succeeding reference is in the same page and the state of the page does not change between the references, this operation can be reduced to $phy\_addr = phy\_addr + a \times i.diff$, where $i.diff$ is the constant change for i during each iteration of the loop. Therefore, pointer dereferencing is used after runtime handle check optimization or loop transformation. During runtime handle check optimization, each time a new page is accessed (i.e., inside the *if* statement) a base pointer is computed; the following accesses in the same page dereference the base pointer instead of referring to the page table. After loop transformation, before entering the inner loop, base pointers are computed, and addresses accessed in the inner loop are computed by dereferencing the base pointer. Figure 5(d) shows that this optimization scheme, which is implemented in line 4, 6, and 7, can eliminate $N - N/PAGESIZE$ address translations. The pointer dereferencing optimization replaces calls

to the `write_handle` and the `read_handle` functions with direct access via a pointer.

Each application may have a different set of effective optimizations, as shown in Section 5.7. The following policy is followed by MEMMU to determine the optimizations to use for a given application:

(1) Apply small object optimization during the instruction replacement pass by leaving reads and writes of small data structures unchanged.
(2) Apply loop transformation to a loop if the referencing array index is a linear function of the induction variable. Then apply pointer dereferencing.
(3) If the second step is not used for the application, then try handle check hoisting.
(4) If neither the second nor third steps are used, and the loop only accesses a single array sequentially, apply the runtime handle check optimization and pointer dereferencing.

This policy implies a priority order on the proposed optimization techniques. However, this selection order is a heuristic and may not be optimal. Each step is provided in a separate compiler pass. Therefore, one might potentially run the passes in another order to find out the optimal solution for a particular application.

4.6 Delta Compression Algorithm

We developed a high-performance, lossless compression algorithm based on delta compression for use in sensor network applications. This algorithm exploits the similarities between adjacent data elements. Despite its simplicity, the algorithm has high performance and a good compression ratio for sensor data in which adjacent samples are often correlated.

To design an appropriate compression algorithm for sensor data, the regularities of the data must be well understood. For this purpose, we collected numerous types of sensor data (e.g., sound, light, and temperature) from Crossbow MICAz and TelosB sensor network nodes and analyzed their characteristics. Intuitively, sensor data are likely to stay similar during a certain period of time, and within a certain geographic range, hence showing high amounts of temporal and spatial locality. For example, in sensor networks deployed for seabird habitat monitoring [Polastre et al. 2004], sensor nodes may be placed in petrel nests in underground burrows. The temperature and humidity sensed from one sensor node usually changes smoothly during a day, except as a result of storms. In addition, the sensor data of temperature and humidity from adjacent burrows are likely to be similar; these data are usually transmitted within a cluster of nodes before they are sent to the base station. Thus, sensor nodes commonly receive highly-redundant data.

A delta-based compression algorithm exploits regularity in data: The difference between two adjacent data elements (delta) usually requires fewer bits to store than the original data [Engelson et al. 2000]. Our implementation of the delta compression and decompression algorithms are presented in Figure 8.

```
Input: IN word stream
Output: OUT word stream
Variable: DATA word stream, TAPE delta stream
 1: for i ∈ {1, ··· , N} do
 2:    δ ← IN[i] - IN[i-1]
 3:    if log2 δ ≤ MAXBITS then
 4:       TAPE[i] ← δ
 5:    else
 6:       TAPE[i] ← MAGIC_CODE
 7:       DATA[i] ← IN[i]
 8:    end if
 9:    OUT ← pack(TAPE, DATA)
10: end for
```

```
Input: IN word stream
Output: OUT word stream
Variable: DATA word stream, TAPE delta stream
1: DATA, TAPE ← unpack(IN)
2: for TAPE[i] in range of TAPE do
3:    if TAPE[i] = MAGIC_CODE then
4:       OUT[i] ← DATA[i]
5:    else
6:       δ ← TAPE[i]
7:       OUT[i] ← OUT[i-1] + δ
8:    end if
9: end for
```

Fig. 8. Delta compression and decompression.

The algorithms are based on the observation that the majority of the deltas can be stored within a predefined *MAXBITS*; if the delta cannot be stored within *MAXBITS*, (i.e., there is a sudden change in sensed data) the raw data are stored, and a *MAGIC_CODE* is recorded to indicate this abnormality. The algorithm also adapts to the compressibility of pages by means of early termination. When the number of deltas that exceed *MAXBITS* is above a certain threshold, causing the "compressed" page to exceed its original size, the algorithm terminates and reports the compressed page size as zero, indicating that this page is not compressed.

In order to identify the *MAXBITS* value that provides the best compression ratio, we analyzed the sample sound data collected by the Crossbow MICAz sensor node. Since the analog-to-digital converter (ADC) on the MICAz generates a 10-bit output, the compression algorithm reads in 2bytes (16bits) at a time and computes the delta on a 2-bytes basis. Figure 9 shows that 95% of the deltas can be represented using 6 bits. Therefore, in our implementation, *MAXBITS* is set to six. Please note that this value may vary depending on the underlying hardware of the sensor node (i.e., the bit width of the ADC).

4.7 Page State Preservation

The optimization techniques proposed in Section 4.5 improve performance by eliminating runtime handle checks and address translations associated with memory references to pages that have been brought into the uncompressed region. They depend on compile-time knowledge and assignment of page status. However, in an event-driven system where an interrupt can preempt a task, an interrupt handler can potentially cause the compression of a page that is being used by a task. If the task resumes after the location of the page changes, an error would occur. This makes the loop transformation and handle check hoisting optimizations unusable. To resolve this problem, we lock pages for which memory references are optimized in the uncompressed region. This is done by introducing a 1-bit flag for each page in the LRU list to indicate whether it is locked. Procedures `lock_handle` and `unlock_handle` are added to MEMMU library to lock a page in the uncompressed region and release the lock. When interrupt handlers can access memory objects outside the reserved region, loop transformation and handle check hoisting need to replace `check_handle` with

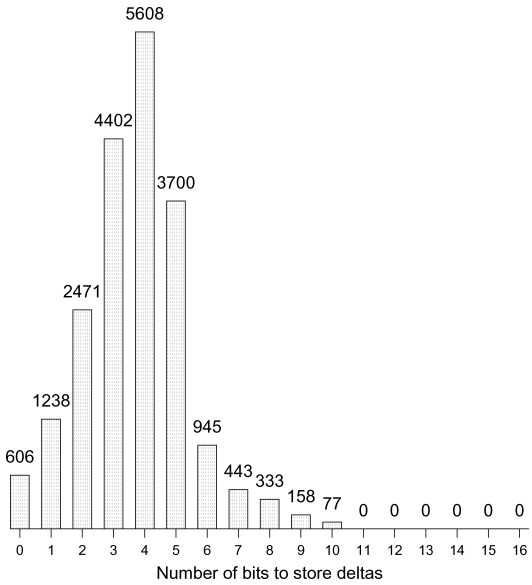

Fig. 9. Histogram of sensor data delta values.

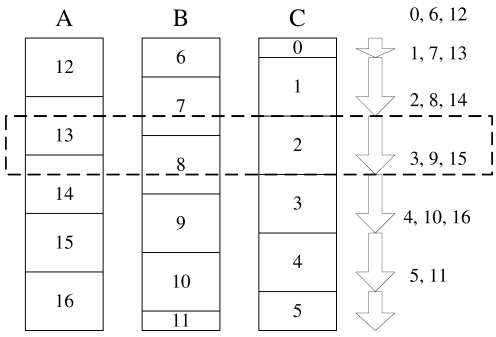

Fig. 10. Example of loop transform on multiple arrays.

`lock_handle` and insert `unlock_handle` after exiting from the optimized inner loop. For example, in Figure 5(c) and (d), `check_handle(pnum)` in line 3 will be replaced with `lock_handle(pnum)`, and `unlock_handle(pnum)` will be inserted after line 6 and line 8, respectively. In TinyOS, tasks do not preempt each other, so the page locking strategy is only required when interrupts can cause data to be moved between the memory regions. In other words, if after applying small object optimization and the ring buffer technique, interrupt handlers only access memory objects in the reserved region, all the optimization techniques discussed in Section 4.5 will still be effective. The page-state preservation strategy can be generalized to multithreaded system by locking pages currently used by each thread. However, the concurrent execution of many threads accessing different pages may degrade the memory-expansion ratio by requiring a larger uncompressed region to allow pages simultaneously used by threads to stay uncompressed.

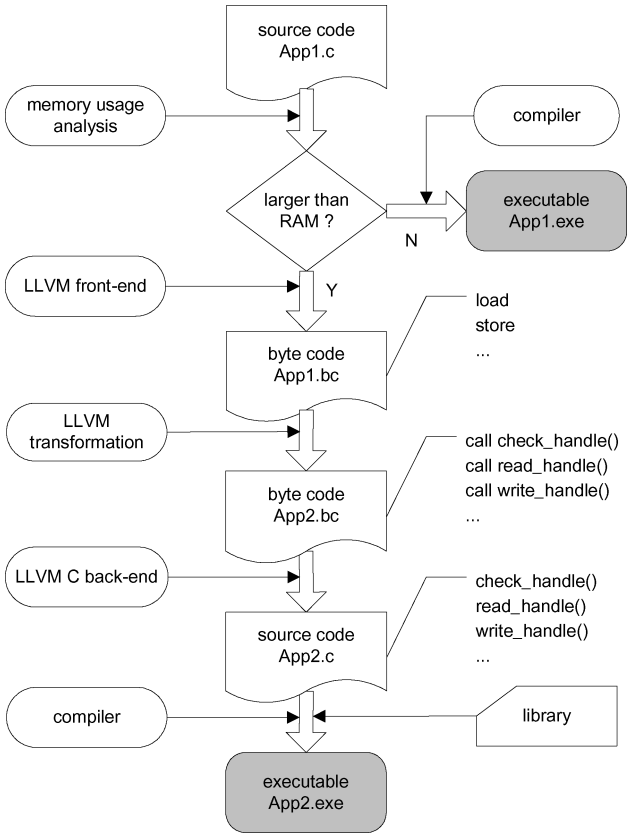

Fig. 11. Overview of technique.

4.8 Summary

Figure 11 illustrates the procedure for using the MEMMU system to automatically generate an executable from mid-level or high-level language source code such as ANSI C. First, the memory requirements of the application are analyzed. If these requirements are smaller than physical RAM, compression is not necessary and therefore no transformations are performed. Otherwise the application code is compiled to byte code by the LLVM compiler. After that, memory load and store instructions are replaced with calls to our handle access functions (i.e., `check_handle`, `read_handle`, and `write_handle`). Other transformations are performed to enable the optimizations described in Section 4.5. A call to a memory initialization routine is also inserted at the beginning of the byte code. The modified byte code is then converted back to high-level language via the LLVM back-end. Finally, the modified application is compiled with the extended library containing our handle access functions to generate an executable.

In the memory initialization routine, physical memory is divided into three regions. The size of each region is computed based on the application memory requirement and the estimated compression ratio of MEMMU (i.e., the average

compression ratio for the many pages of data that may be in use at any point in time). Since the runtime data compression ratio cannot be accurately decided at compile-time, it is possible for the runtime compression ratio to be worse than the predicted compression ratio, causing execution to stop when both memory regions are full. Therefore, it is suggested that users determine the compression ratio based on sample data of their application and set the MEMMU compression ratio appropriately. This process could potentially be automated by running the selected compression algorithm on sample data sets.

For any compression algorithm, it is possible to construct an input that will result in a compression ratio greater than 1. Similarly, given any predicted application average compression ratio, it is possible to construct a sequence of inputs on which compression will exceed the ratio. The frequency of encountering such a sequence of inputs in the field depends strongly on the application. For many applications, such an event will be rare. For example, the compression ratio for individual pages of the vibration data and temperature data shown in Section 5.8 never exceed 78.1% and 44.5%, respectively, during 6 months of measurement. Section 5.8 also shows that when the estimated compression ratio is set to 1.05 $\times$ the average-page compression ratio, this results in a very low probability of memory exhaustion for this application: 0.38% or 5.5×10^{-7}% every 30 minutes. Although it is important that the probability of memory exhaustion be low, we believe that it need not be zero in many applications. For example, if this probability is orders of magnitude lower than that of node hardware failure [Szewczyk et al. 2004], its impact on system reliability will be negligible. If an application required zero probability of memory exhaustion, but the designers still want the functionality and ease-of-design benefits MEMMU can bring, it would be possible to migrate data to secondary storage in the rare event of memory exhaustion (e.g., by using the technique proposed by Choudhuri and Givargis [2005]). Combined with MEMMU, this would eliminate the risk of memory overuse at the cost of extremely-rare performance penalties when secondary storage must be used.

In our experiments, MEMMU is tested on TelosB motes running TinyOS [Gay et al. 2005]. TinyOS and its applications are written in nesC [Gay et al. 2003]. NesC is an extension to the C programming language that supports the structure and execution model of TinyOS. Ncc is the NesC compiler for TinyOS. TinyOS itself does not support dynamic memory allocation, so there are only stack and global variables in the nesC program; this simplifies analysis of application memory requirements.

LLVM does not have a nesC front-end. As a result, one of three possible flows may be used. In the first, a mote development environment based on ANSI C, such as MANTIS OS, may be directly used with LLVM. In the second, the ANSI C computation-intensive portion of the application is manually extracted from the nesC code, provided to LLVM for transformation, and reinserted in the nesC code before compilation with ncc. We used this approach for the experiments presented in Section 5. However, we have subsequently developed a fully automated flow. First, the nesC program is transformed to C by ncc. Then the C program is transformed to byte code by llvm-gcc and MEMMU compiler passes are applied. Finally, the LLVM C-back-end transforms the byte code back to a

C program, and the C program is compiled to an executable by ncc. This flow is complicated by the fact that ncc inserts inline assembly, which LLVM C-backend does not yet support. We have, therefore, developed a script to temporarily associate inline assembly with dummy function calls, permitting restoration after LLVM transformation passes.

5. EXPERIMENTAL RESULTS

This section presents the results of evaluating MEMMU using five representative wireless sensor network applications. These benchmarks were executed on a TelosB wireless sensor node. The TelosB is an MMU-less, low-power, wireless module with integrated sensors, radio, antenna, and an 8MHz Texas Instruments MSP430 microcontroller. The TelosB has 10KB RAM and typically runs TinyOS. The benchmarks are tested with three system settings: running the original applications without MEMMU, with an unoptimized version of MEMMU, and with an optimized version of MEMMU. Four metrics were evaluated: average power consumption, execution time, processing rate, and memory usage. We measured total memory usage, memory used by MEMMU, and division between memory regions. Processing rate is defined as application data size divided by execution time. Power measurements were taken using a National Instruments 6034E data acquisition card attached to the PCI bus of a host workstation running Linux. Power was computed based on the measured voltage across a 10Ω resistor in series with the power supply. The average power of duty cycle-based applications is calculated using the following equation.

$$P_{average} = \frac{P_{active} \times t_{active} + P_{idle} \times t_{idle}}{t_{active} + t_{idle}} \tag{2}$$

All of LLVM's optimizations are turned off to ensure all the overheads and savings are entirely due to MEMMU. The experimental results show that, with the exception of the image convolution benchmark, the execution time overheads of all other benchmarks are below 10%. In Sections 5.1–5.5, we will describe each benchmark and discuss the corresponding results in detail.

5.1 Sound Filtering

The first example application is sound filtering. When the hardware timer periodically fires, the mote starts one-dimensional filtering on collected audio data. The MSP430 microcontroller automatically puts itself into a low-power mode when the task stack is empty and wakes up when the next timer event arrives. As shown in Figure 12, the power waveform is similar to a square wave. For this benchmark, we assume fixed application and input data sizes (buffer sizes) and compare the memory usage to determine the amount of memory saved by using MEMMU.

Table I shows results for this benchmark when running under three system settings. The memory reduction achieved by MEMMU is 9,935 − 7,243 = 2,692 bytes, which is 27% of the original memory requirement. The saved memory is available to store other data, which may be larger than 2,692 bytes as a result of compression. For this benchmark, small object optimization, loop

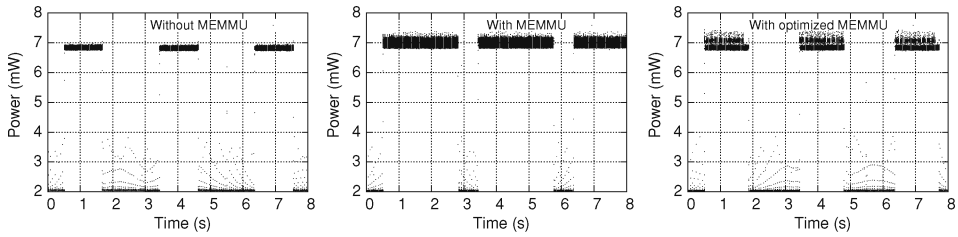

Fig. 12. Power consumption of the sound-filtering benchmark using three settings.

Table I. Filtering Benchmark

	RAM usage (B)	Buffer size (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Active power (mW)	Average power (mW)
Orig.	9,935	9,728	0	0	0	1.24	6.77	3.94
Unopt.	7,243	9,728	518	3,840	2,560	2.31	6.97	5.92
Opt.	7,243	9,728	518	3,840	2,560	1.35	6.80	4.27

Table II. Overhead of MEMMU Functions

Function name	Compress	Decompress	Swap_in	Swap_out	Check_handle
Percentage of overhead (%)	67.07	0	17.32	15.44	0.17

transformation, and pointer dereferencing were applied. The processing time and active power consumption overheads of unoptimized MEMMU are 86.3% and 3.0%, while after optimization, the overheads are reduced to 8.9% and 0.4%, respectively. Figure 12 depicts the power consumption under the three system settings. According to Equation 2, there are two causes of increased average power consumption. First, the mote stays in active mode longer when MEMMU is used. Second, active power consumption increases slightly as a result of MEMMU's computations.

Table II shows the performance overhead from calling MEMMU functions when the optimized version of MEMMU is used. This breakdown in performance overhead was determined by sampling the program counter at a period of 100Hz during application execution using these data to compute the percentage of execution time spent in each function. Over half of the overhead comes from `compress`; 17.32% and 15.44% may be attributed to `swap_in` and `swap_out`, which contain the instructions to search for free pages and update the page list. `Check_handle` calls `swap_in` and `swap_out` if the checked page is compressed and no free page in the uncompressed region is available. `Swap_in` calls `swap_out` if there is no space in the uncompressed region. `Swap_out` calls `compress` to compress a victim page. Note that decompression is very efficient. Therefore, the overhead from decompression is close to 0.

We also use this benchmark to evaluate the changes in performance as the memory required by the application increases (i.e., as the memory expansion ratio of MEMMU increases). Figure 13 shows the increase in performance (processing rate) as a function of data size in the filtering benchmark using the optimized version of MEMMU. The total physical memory usage stays constant. The left-most point shows the base case, in which the physical memory

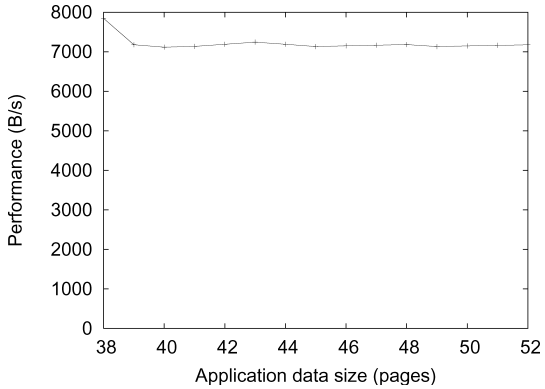

Fig. 13. Relation between performance and application data size.

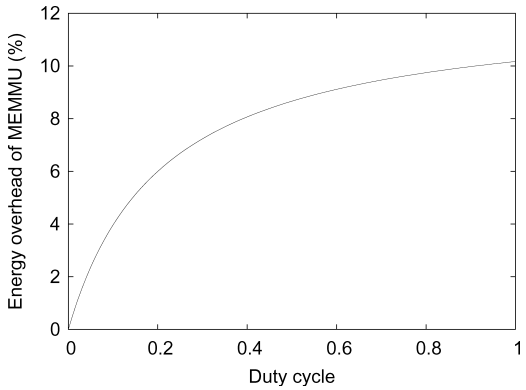

Fig. 14. Energy overhead of MEMMU as a function of duty cycle.

is sufficient to run the application. In this case, MEMMU is not used. Each of the other points in the figure corresponds to an optimal memory division that minimizes the performance overhead, while meeting the memory requirement. The results show that the performance penalty stays almost constant, despite increasing application data size. Therefore, even though a larger compression region is needed as application data sets grow, the performance overhead of MEMMU is fairly stable.

5.2 Image Convolution

Our second example application is a convolution algorithm in which a large matrix is convolved with a 3×3 coefficient kernel matrix. Note that 2-D convolution is used for graphical images. In order to permit consistent input to allow fair comparisons for each test case, the input images were generated by scaling the same image to different sizes; a gray-scale image of a cloudy sky was used. The input images were transferred to the mote via USB. Table III compares the input and output image sizes, RAM usage, processing rate, execution time, and average power consumption of the benchmark application under three settings. The results indicate that using the same amount of physical RAM, MEMMU

Table III. Convolution Benchmark

	RAM usage (B)	Input image (B)	Output image (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Proc. rate (B/s)	Active power (mW)
Orig.	9,739	4,900	4,624	0	0	0	1.50	6,349	6.57
Unopt.	9,739	6,084	5,776	638	6400	2304	4.47	2,653	6.82
Opt.	9,739	6,084	5,776	638	6400	2304	2.88	4,118	6.75

Table IV. Light Sampling Benchmark

	RAM usage (B)	Buffer size (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Proc. rate (B/s)	Active power (mW)
Orig.	9,474	9,040	0	0	0	4.39	2,059	57.44
Unopt.	9,474	13,200	603	5,120	3,328	6.53	2,021	58.61
Opt.	9,474	13,200	603	5,120	3,328	6.47	2,040	58.11

allows the application to handle images that require more memory than is physically available: The unmodified TelosB can only handle an input image smaller than 4.8KB, while MEMMU allows the mote to process images that are 25% larger (6KB). Since the delta compression algorithm is less efficient for 8-bit images, the compression ratio in this case is 62.4%. We believe a lossy compression algorithm designed for image data would permit a higher usable memory improvement ratio.

Unfortunately, the increase in image size imposes a cost. Using MEMMU results in a 58.2% decrease in processing rate and 3.8% increase in power consumption. After applying small object optimization and handle check hoisting, the processing rate penalty was reduced to 35.1% and the power consumption penalty was reduced to 2.1%. Please note that the image convolution benchmark was the only benchmark for which MEMMU had a performance overhead higher than 10% after optimization. The performance penalty reduction is smaller compared to other applications because pointer dereferencing cannot be used to reduce the penalty caused by address translation.

5.3 Data Sampling

The third example application is sensor data sampling. In this application, the mote senses the light level every 1ms and stores the data to a buffer. When the buffer is full, its contents are sent via the wireless transmitter. Small object optimization, handle check hoisting, and pointer dereferencing were applied to this benchmark. Table IV shows that with MEMMU, the buffer size is increased by 46.0% without increasing physical memory usage. The average power consumption overheads are 2.0% and 1.1% for unoptimized and optimized MEMMU, respectively. The processing time and processing rate measure the time and speed of transmitting the data in the buffer. The processing rate is reduced by 1.8% with unoptimized MEMMU. Optimizations reduced the performance overhead to 0.9%.

Table V. Covariance Matrix Computation Benchmark

	RAM usage (B)	Buffer size (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Proc. rate (B/s)	Active power (mW)
Orig.	9,643	9,430	0	0	0	0.47	19,895	5.22
Unopt.	9,643	13,056	602	5,120	3,584	1.44	9,067	5.40
Opt.	9,643	13,056	602	5,120	3,584	0.72	18,133	5.36

Table VI. Correlation Computation Benchmark

	RAM usage (B)	Signal size (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Proc. rate (B/s)	Active power (mW)
Orig.	6,669	6,460	0	0	0	7.98	810	5.34
Unopt.	6,669	9,728	543	4532	1536	28.3	344	5.36
Opt.	6,669	9,728	543	4532	1536	13.00	748	5.35

5.4 Covariance Matrix Computation

The fourth example application is covariance matrix computation. This application is useful in statistical analysis and data reduction. For example, it is the first stage of principal component analysis. Each vector contains a number of scalars with different attributes (e.g., different types of sensor data). Small-object optimization, runtime handle check optimization, and pointer dereferencing were applied to this benchmark. Table V shows that MEMMU permits more vectors to be processed at a single time: the buffer size increases by 38.5%. Although the performance penalty of unoptimized MEMMU is large (the processing rate is decreased by 54.4%), optimizations reduce it greatly. The processing rate using the optimized version of MEMMU is only 8.9% lower than the original application. The average power consumption penalties of both unoptimized and optimized MEMMU are below 4%.

5.5 Correlation Calculation

The last example application performs sound propagation delay estimation based on correlation calculation. This application is used to determine the relative locations of sensors. Small object optimization, runtime handle check optimization, and pointer dereferencing were applied to this benchmark. As shown in Table VI, MEMMU increases the size of the input data by 50.6%. Although the unoptimized version of MEMMU reduces the processing rate by 57.5%, the optimized MEMMU reduces the processing rate by only 7.6%. The penalties to average power consumption of both unoptimized and optimized MEMMU are no more than 0.5%.

5.6 Overhead of Code Size

Table VIII shows the increase in code size for each benchmark. On average, executables generated with MEMMU transformations are 30% larger than those directly compiled from the original source code. Nevertheless, the code size increase does not lead to flash memory size increase in current architectures because most sensor network nodes provide sufficient flash memory (e.g., the

Table VII. Comparison of Optimization Techniques

	Run time of benchmarks with different MEMMU optimizations (s)					
Benchmark	Unopt. MEMMU	Runtime handle check	Handle check hoisting	Loop trans.	Runtime handle check & pointer deref.	Loop trans. & pointer deref.
Filtering	1.84	1.25	1.30	1.18	1.20	1.12
Sampling	5.39	5.38	N.A.	N.A.	5.37	N.A.
Correlation	21.11	22.50	N.A.	22.53	15.20	12.94
Covariance	1.12	0.86	0.83	N.A.	0.53	N.A.
Convolution	2.88	2.63	1.97	N.A.	N.A.	N.A.

Table VIII. Code Size Overhead Introduced by MEMMU

Code size	Filtering	Convolution	Sampling	Covariance	Correlation
Original (B)	16,020	16,725	15,282	16,400	16,919
With MEMMU (B)	20,888	21,882	18,630	21,631	22,019
Overhead (%)	30.4	30.8	21.9	31.9	30.1

TelosB has 48KB of program flash memory and the MicaZ has 128KB of program flash memory). Therefore, the overhead of code size can be neglected unless the amount of code memory becomes a tight constraint. This is not expected in the near future due to the high density of floating-gate technologies such as EEPROMs and flash memory, relative to SRAM.

5.7 Comparisons on Different Optimization Techniques

To understand the relative benefits of the proposed optimization techniques, we compare the improvement in performance by applying these approaches individually and in combination to five benchmarks. Table VII shows the execution time of the applications with unoptimized MEMMU and MEMMU augmented with different optimization techniques. "N.A." indicates that an optimization technique cannot be applied to the corresponding benchmark. For instance, loop transformation cannot be used for the sensor data sampling application because the program is an implicit loop that executes the next iteration only when a hardware-triggered event occurs; there is no explicit loop structure in the code that can be transformed. Note that the runtime handle check optimization increases the execution time of the unoptimized MEMMU for the correlation computation benchmark because this application carries out interleaved access to two arrays. Generally, loop transformation with pointer dereferencing outperforms other optimization techniques because this combination can achieve the largest reduction in the number of handle checks and address translations.

5.8 Compression Ratio Estimation and Probability of Memory Exhaustion

As discussed in Section 4.8, the division between the compressed and the uncompressed regions is based on an estimated compression ratio. Underestimating the compression ratio will result in failure due to memory exhaustion. We will now use a statistical technique to analyze the probability of running out of memory for a real-world data set. The input data are vibration samples

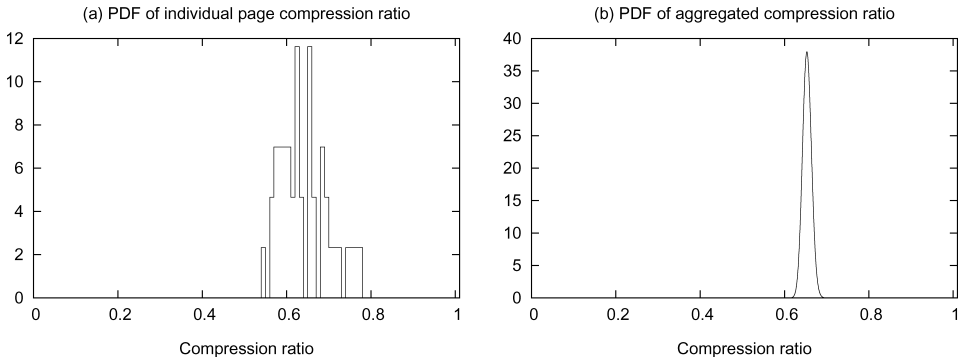

Fig. 15. Aggregated compression ratio analysis on vibration data.

gathered from a wireless sensor network deployed in a building for infrastructure health monitoring [Dowding et al. 2005]. We divide the data into 256-byte pages and compress them with the delta compression algorithm, described in Section 4.6. The probability density function (PDF) of the page compression ratios is shown in Figure 15(a). The average compression ratio of an individual page is 64.7% and the standard deviation is 0.058. For a compressed region containing 30 compressed pages, we derive the average compression ratio by convolving the PDF of the page compression ratio by the number of compressed pages. Figure 15(b) shows the PDF of the aggregated compression ratio of pages in the compressed region. It still has an average of 64.7%, but with a much smaller standard deviation: 0.01. The standard deviation of the aggregated compression ratio decreases as the number of compressed pages increases due to the Law of Large Numbers. If we set the target compression ratio to 1.05 $\times$ the average compression ratio of individual pages (i.e., 67.9%), the probability of the aggregated compression ratio exceeding our target compression ratio every time the data in the compressed region change is 0.38%. This probability drops to 1.74×10^{-6}% if we set the target compression ratio to 1.1 $\times$ the average compression ratio of individual pages. If we use the data sampling period, 30 minutes, to approximate the period of updating the compressed region, Mean Time To Failure (MTTF) can be computed by dividing the sampling period by the failure probability. The MTTF increases from 131.6 hours to 2.87×10^7 hours when we slightly increase target compression ratio from 67.9% to 71.2%. The same analysis is done with temperature data gathered from the same system. Figure 16 shows the results. The average compression ratio for an individual page is 38.6% and the standard deviation is 0.009. The standard deviation of the average compression ratio is 0.002. The probability of running out of memory every 30 minutes is 5.5×10^{-7}%, when the estimation compression ratio is 1.05$\times$ the average. The MTTF is 9.1×10^7 hours.

This analysis is based on the assumption that compression ratios of pages in the compressed region are independent. Computing the correlation among pages in the compressed region is challenging and complex due to the interaction among sampling and computation. However, we can get a fairly conservative estimate of the correlation by observing that, for most applications,

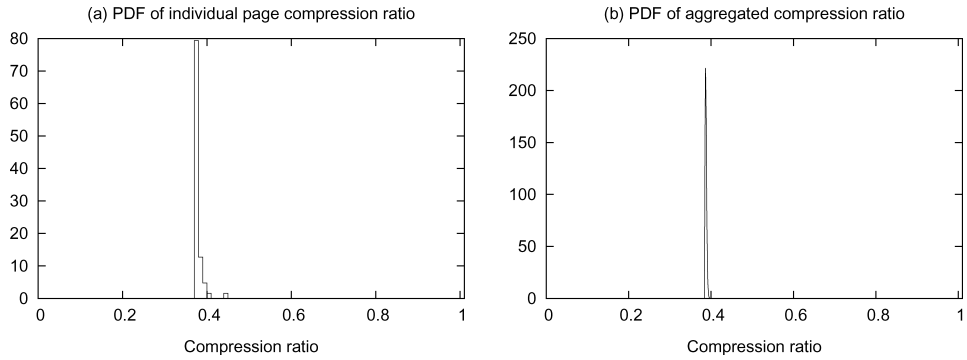

Fig. 16. Aggregated compression ratio analysis on temperature data.

adjacent pages of sampled data have greater compression correlation than those that are separated by more time. We computed the correlation of compression ratios of neighboring pages, and they are quite low (0.125 and 0.122) for the vibration and temperature monitoring applications.

5.9 Summary

To summarize, MEMMU reduces the physical memory requirements of applications by 27% or expands usable memory by up to 50%. The performance overhead of unoptimized MEMMU ranges from 57.5% to 86.3%. For four of the five benchmarks, optimization techniques reduce the performance overhead to below 10%. However, the image convolution application is an exception. Its performance overhead after optimization is 35.1% because the pointer dereferencing optimization technique cannot be used. There is a trade-off between memory expansion proportion and performance. Larger usable memory is obtained by using a larger compressed memory region, but this results in more compression/decompression and data migration operations, reducing speed.

Please note that we were quite conservative in our evaluation of MEMMU. The original goal of MEMMU is to expand memory allowing applications requiring more memory than physically present to still run. However, if we were to only test such large benchmarks, the outcome would often be “crash” for a system without MEMMU and “finish execution” for a system with MEMMU. Such an evaluation scheme would not illustrate the impact of MEMMU on performance. Therefore, we reduced the data set size of the application running without MEMMU and compared the data processing rates of the smaller applications with those of more demanding applications running with MEMMU.

The energy consumption overhead imposed by MEMMU depends on the duty cycle and communication activity of the applications. Duty cycle is the fraction of time that the wireless sensor mote is active. An upper-bound on the energy overhead can be derived from our average active power overhead and runtime overhead. This upper-bound is 12%. Many real-world applications have duty cycles lower than 10% in order to maximize the life time of the system [Hartung et al. 2006; Tolle et al. 2005]. In this case, the energy consumption overhead of MEMMU decreases as the system spends more time in idle mode. Note that

the most direct alternative to using MEMMU is using a sensor network node with more RAM. This may be impossible, due to the limited types of nodes available. However, even if it is possible, increasing memory quantity increases its power consumption. An analysis with CACTI [Tarjan et al. 2006] indicates that for a 180nm process, doubling the amount of memory from 10KB to 20KB increases read and write energy consumption by 50% and 30%, respectively. Leakage power is also increased, although leakage will only be a serious problem if future sensor network node processors are fabricated using finer process technologies such as 90nm or 65nm. The power consumption during wireless data transmission is approximately 10× as high as when the radio is turned off for TelosB and 3.8× as high for MicaZ [Polastre et al. 2005]. For applications that require periodic data transmission to a base station, or constant data exchange among nodes, the energy overhead of MEMMU will be negligible. Given 8% runtime overhead and 4% computation power overhead, Figure 14 shows the energy overhead of MEMMU as a function of duty cycle assuming 2% of the time is spent transmitting. For applications with duty cycles lower than 10%, MEMMU has an energy overhead smaller than 4%.

6. CONCLUSIONS

We have described MEMMU, an efficient software-based technique to increase usable memory in MMU-less embedded systems via automated online compression and decompression of in-RAM data. A number of compile-time and runtime optimizations are used to minimize its impact on the performance and power consumption. Different optimization approaches may impact performance in different ways, depending on application memory reference patterns. An efficient delta-based compression algorithm was designed for sensor data compression. MEMMU was evaluated using a number of representative wireless sensor network applications. Experimental results indicate that the proposed optimization techniques improve MEMMU's performance and that MEMMU is capable of increasing usable memory by 39%, on average, with less than 10% performance and power consumption penalties for all but one application. We have released MEMMU for free academic and nonprofit use [MEMMU].

ACKNOWLEDGMENTS

We would like to thank Siddharth Choudhary and Tony Givargis for sharing their technical report [Choudhuri and Givargis 2005] and their helpful observations on software-controlled virtual memory. We would also like to thank Matthew Simpson, Bhuvan Middha, and Rajeev Barua for sharing a preprint of their inspiring paper on segment protection [Simpson et al. 2005] as well as Charles Dowding and Mat Kotowsky for sharing their data [Dowding et al. 2005].

REFERENCES

Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., and Han, R. 2003. MANTIS: system support for MultimodAl NeTworks of in-situ sensors. In *Proceedings of the International Workshop on Wireless Sensor Networks and Applications*. ACM, New York, 50–59.

Banerjee, U. 1993. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Boston, MA.

Biswas, S., Simpson, M., and Barua, R. 2004. Memory overflow protection for embedded systems using runtime checks, reuse and compression. In *Proceedings of the International Conference on Compilers, Architecture & Synthesis for Embedded Systems (CASES'04)*. ACM, New York, 280–291.

Choudhuri, S. and Givargis, T. 2005. Software virtual memory management for MU-less embedded systems. Tech. rep., Center for Embedded Computer Systems, University of California, Irvine.

Cooprider, N. and Regehr, J. 2007. Online compression for on-chip RAM. In *Proceedings of the Programming Languages Design and Implementation*. ACM, New York.

Douglis, F. 1993. The compression cache: Using online compression to extend physical memory. In *Proceedings of the USENIX Conference*. 519–529.

Dowding, C. H. and McKenna, L. M. 2005. Crack response to long-term and environmental and blast vibration effects. *J. Geotech. Geoenviron. Eng. 131*, 9, 1151–1161.

Engelson, V., Fritzson, D., and Fritzson, P. 2000. Lossless compression of high-volume numerical data from simulations. In *Proceedings of the Data Compression Conference.* IEEE, Los Alamitos, CA, 574.

Franke, B. and O'Boyle, M. 2001. Compiler transformation of pointers to explicit array accesses in DSP applications. In *Proceedings of the International Conference on Compiler Construction*. Springer, Berlin, Germany, 69–85.

Ganesan, P., Venugopalan, R., Peddabachagari, P., Dean, A., Mueller, F., and Sichitiu, M. 2003. Analyzing and modeling encryption overhead for sensor network nodes. In *Proceedings of the International Conference on Wireless Sensor Networks and Applications*. ACM, New York, 151–159.

Gay, D., Levis, P., and Culler, D. 2005. Software design patterns for TinyOS. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, New York, 40–49.

Gay, D., Levis, P., Culler, D., and Brewer, E. 2003. nesC 1.1 language reference manual. http://nescc.sourceforge.net/papers/nesc-ref.pdf.

Gehrke, J. and Madden, S. 2004. Query processing in sensor networks. *Pervasive Comput. 3*, 1, 46–55.

Guestrin, C., Bodi, P., Thibau, R., Paski, M., and Madde, S. 2004. Distributed regression: an efficient framework for modeling sensor network data. In *Proceedings of the International Symposium on Information Processing in Sensor Networks*. ACM, New York, 1–10.

Hartung, C., Han, R., Seielstad, C., and Holbrook, S. 2006. FireWxNet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. ACM, New York, 28–41.

Hellerstein, J. M. and Wang, W. 2004. Optimization of in-network data reduction. In *Proceedings of the International Workshop on Data Management for Sensor Networks*. ACM, New York, 40–47.

Karlof, C. and Wagner, D. 2003. Secure routing in wireless sensor networks: attacks and countermeasures. *Elsevier's AdHoc Networks J. 1*, 2–3, 293–315.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, New York, 75–86.

Lekatsas, H., Henkel, J., and Wolf, W. 2000. Code compression for low power embedded system design. In *Proceedings of the Design Automation Conference*. IEEE, Los Alamitos, CA, 294–299.

Li, D., Wong, K., Hu, Y., and Sayeed, A. 2002. Detection, classification, and tracking of targets. *Signal Process. Mag. 19*, 2, 17–29.

Madden, S., Franklin, M., Hellerstein, J., and Hong, W. 2002. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the Symposium on Operating Systems Design and Implementation.* ACM, New York, 131–146.

Mckinley, K. S., Carr, S., and wen Tseng, C. 1996. Improving data locality with loop transformations. *ACM Trans. Program. Lang*. Syst. 424–453.

MEMMU. Memory expansion on embedded systems without MMUs. http://robertdick.org/tools/html.

MUCHNICK, S. S. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, St. Louis, MO.

NATH, S., GIBBONS, P. B., SESHAN, S., AND ANDERSON, Z. R. 2004. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the International Conference on Embedded Networked Sensor Systems*. ACM, New York, 250–262.

OBERHUMER, M. F. LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo.

PEREIRA, C., GUPTA, S., NIYOGI, K., LAZARIDIS, I., MEHROTRA, S., AND GUPTA, R. 2003. Energy efficient communication for reliability and quality aware sensor networks. Tech. rep., University of California at Irvine.

POLASTRE, J., SZEWCZYK, R., AND CULLER, D. 2005. Telos: enabling ultra-low power wireless research. In *Proceedings of the International Symposium on Information Processing in Sensor Networks*. ACM, New York.

POLASTRE, J., SZEWCZYK, R., MAINWARING, A., CULLER, D., AND ANDERSON, J. 2004. Analysis of wireless sensor networks for habitat monitoring. In *Proceedings of the Wireless Sensor Networks Symposium*. ACM, New York, 399–423.

POTTIE, G. J. AND KAISER, W. J. 2000. Wireless integrated network sensors. *Comm. ACM 43*, 5, 51–58.

PRADHAN, S. S., KUSUMA, J., AND RAMCHANDRAN, K. 2002. Distributed compression in a dense microsensor network. *IEEE Signal Process. Mag. 19*, 2, 51–60.

RIZZO, L. 1997. A very fast algorithm for RAM compression. *Operat. Syst. Rev. 31*, 2, 36–45.

SIMPSON, M., MIDDHA, B., AND BARUA, R. 2005. Segment protection for embedded systems using runtime checks. In *Proceedings of the International Conference on Compilers, Architecture & Synthesis for Embedded Systems*. ACM, New York, 25–27.

SZEWCZYK, R., POLASTRE, J., MAINWARING, A., AND CULLER, D. 2004. Lessons from a sensor network expedition. In *Proceedings of the 1st European Workshop on Sensor Networks*. Springer, Berlin, Germany.

TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. P. 2006. CACTI 4.0. Tech. rep., HP Laboratories.

TOLLE, G., POLASTRE, J., SZEWCZYK, R., CULLER, D., TURNER, N., TU, K., BURGESS, S., DAWSON, T., BUONADONNA, P., ET AL. 2005. A macroscope in the redwoods. In *Proceedings of the International Conference on Embedded Networked Sensor Systems*. ACM, New York, 51–63.

TREMAINE, B., FRANASZEK, P. A., ROBINSON, J. T., SCHULZ, C. O., SMITH, T. B., WAZLOWSKI, M., AND BLAND, P. M. 2001. IBM memory expansion technology. *IBM J. Res. Dev. 45*, 2, 271–285.

TUDUCE, I. C. AND GROSS, T. 2005. Adaptive main memory compression. In *Proceedings of the USENIX Conference*. 237–250.

VAN ENGELEN, R. A. AND GALLIVAN, K. A. 2001. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'01)*. IEEE, Los Alamitos, CA, 80.

WILSON, P. R., KAPLAN, S. F., AND SMARAGDAKIS, Y. 1999. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Conference*. 101–116.

YANG, L., DICK, R. P., LEKATSAS, H., AND CHAKRADHAR, S. 2005. CRAMES: Compressed RAM for embedded systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. IEEE, Los Alamitos, CA.

YANG, L., LEKATSAS, H., AND DICK, R. P. 2006. High-performance operating system controlled memory compression. In *Proceedings of the Design Automation Conference*. ACM, New York, 701–704.

Received July 2007; revised May 2008; accepted August 2008