# Automated Compile-Time and Run-Time Techniques to Increase Usable Memory in MMU-Less Embedded Systems

Lan S. Bai, Lei Yang, and Robert P. Dick
Electrical Engineering and Computer Science Department
Northwestern University
2145 Sheridan Road
Evanston, Illinois 60208
lba292, lya755, dickrp@eecs.northwestern.edu

## ABSTRACT

Random access memory (RAM) is tightly-constrained in many embedded systems. This is especially true for 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 4–10 KB of RAM and do not contain memory management units (MMUs). It is very difficult to implement increasingly 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 requirements of applications.

We propose the use of automated compile-time and run-time techniques to increase the amount of usable memory in MMU-less embedded systems. The proposed techniques do not increase hardware cost, and are designed to require few or no changes to existing applications. We have developed a fast compression algorithm well suited to this application, as well as run-time library routines and compiler transformations to control and optimize the automatic migration of application data between compressed and uncompressed memory regions. These techniques were experimentally evaluated on Crossbow TelosB sensor network nodes running a number of data collection and signal processing applications. The 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

**Keywords:** Embedded System, Wireless Sensor Network, Data Compression

This work was supported in part by the NSF under award CNS-0347941 and in part by NEC Laboratories America.

## 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. For example, eight-bit microcontrollers generally have no memory management units (MMUs). Sensor network nodes are among the embedded systems whose resource and power are most tightly constrained. 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 [11], encryption [5], query processing [8], and signal processing [14] 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 between 4 KB and 10 KB of RAM, a substantial portion of which is consumed by the operating system (OS), e.g., TinyOS [6] or MANTIS OS [1]. Tight constraints on 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 deep trench capacitors used for high-density RAM 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, and hence cost, as well as 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 either by requiring additional integrated circuit packages or by requiring 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 some 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 re-design of applications are unlikely to be used by anybody but embedded systems programming experts. 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.

In this article, we propose a new memory expansion technique, named MEMMU, for use in wireless sensor networks. This technique uses compile-time transformation and run-time library support to automatically manage on-line migration of data between compressed and uncompressed memory regions in sensor network nodes. It provides application developers with access to more usable RAM and requires no or minor changes to application code and no changes to hardware. The proposed technique requires no MMU and has other design features enabling its use in sensor network nodes with extremely tight memory and performance constraints. 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 overhead is small. We plan to release MEMMU for free academic and non-profit use [17].

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 and effectiveness of the proposed technique. Section 4 describes the library and compiler techniques, optimization schemes, as well as the compression or decompression algorithms designed to automatically increase usable memory in sensor network nodes. Section 5 presents the experimental set-up, describes the workloads, and discusses the experimental results in detail. Finally, Section 6 concludes the article.

## 2. RELATED WORK

The proposed library and compiler techniques to increase usable memory were built upon work in the areas of on-line 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 [3] 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 that secondary storage, e.g., EEPROM or Flash, is present in the system. Their technique automatically manages data migration between RAM and the secondary storage to provide applications access to more memory than provided by physical RAM. However, since access to such secondary storage is significantly slower than that to RAM, the performance penalty of this approach can be very high for some applications. In contrast, MEMMU requires no secondary storage and its performance and power consumption penalties have been minimized via various 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 [24] insert a hardware compression/decompression unit between cache and RAM. Data are stored uncompressed in cache, and are compressed on-the-fly when transferred to memory. Code compression techniques [13] 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.

### 2.3 Software-Based On-Line Memory Compression

Compressed caching [4, 26] introduces a software cache to the virtual memory system that uses part of the memory to store data in compressed format. Swap compression [25] 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 a backing store, i.e., hard disks, when the compressed cache is filled up. In contrast, MEMMU does not rely on any backing store.

CRAMES [27] is an OS controlled, on-line memory compression framework designed for disk-less embedded systems. It takes advantage of the OS virtual memory infrastructure and stores 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 does not require the use of an MMU.

Biswas et al. [2] described a memory reuse method that reduces the memory footprint by compressing live globals in place and growing stack or heap into the freed region when they overflow. Their technique relies upon static liveness analysis in order to identify memory locations for reuse. In addition, this memory reuse method cannot compress portions of arrays unless the compiler does independent liveness analysis on each scalar within an array. In contrast, MEMMU compresses pages instead of variables and dynamically selects LRU victim pages for compression.

### 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 [21]. In-network data aggregation [16, 9] and data reduction via wavelets or distributed regression [10, 18] can significantly reduce the volume of communicated data. However, these techniques are lossy, limiting their application. Recently, researchers have proposed to reduce the amount of data communication via compression [19, 22] in order to reduce radio energy consumption. In contrast, MEMMU focuses on automated memory compression for functionality improvement instead of communication reduction.

### 2.5 Software-Based Memory Compression Algorithms

LZO [15] is a very fast general-purpose compression algorithm that works well on memory data. However, the memory requirement of LZO is at least 8 KB, far exceeding the available memory of many low-end embedded systems and sensor nodes. Rizzo et al. [23] 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. [26] 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. [28] designed a new software-based memory compression algorithm, named pattern-based partial match (PBPM) for embedded systems. This algorithm explores frequent patterns that occur within each word of memory and takes advantage of the similarities among words by using a small dictionary.

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 are that (1) zero words are rare in sensor data, (2) the similarities among sensor datum are not sufficient even though data often change gradually with time, and (3) the page size is usually significantly smaller in low-cost MMU-less devices than in other embedded systems. 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%.

## 3. MOTIVATIONAL SCENARIO

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. Barring slow EEPROM writes, on a Crossbow TelosB sensor node the buffer cannot exceed approximately 9.5 KB of RAM. Moreover, sampling rate and/or duration cannot be increased without redesigning the sensor node hardware or complicating the 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-used 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 16.4 KB available when, in fact, it still contains only 10 KB of physical RAM (please refer to Section 5).

Furthermore, most wireless sensor networks use a store-and-forward technique to control the flow of 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 in the process of 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 possibility of data loss.

## 4. MEMORY EXPANSION ON EMBEDDED SYSTEMS WITHOUT MMUS

This section describes the design 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 on-line 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 subproblems:

1. Intelligently 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 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 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, those sensed, processed, and communicated in sensor network nodes, e.g., 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, including MEMMU. The compressed region and the uncompressed region are both used by applications. Application data are automatically migrated between these 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 somewhat slower form of memory, and the uncompressed region can be viewed as a small, high-performance data cache.

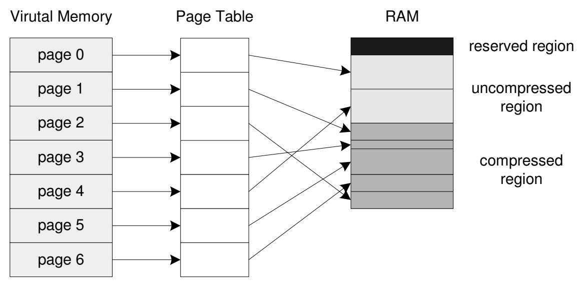

**Figure 1: Memory layout**

Figure 1 illustrates the memory layout of an embedded system using MEMMU. From the perspective of application designers, all pages in the left-most *Virtual Memory* column are available. Memory objects in virtual memory are mapped to the uncompressed or compressed region via a software-maintained page table. A memory management mechanism was designed to manage data compression, decompression, and migration between two regions.

## 4.1 Handle-Based Data Access

Data elements are accessed via their virtual address handles. The floor of virtual address divided by the page size is the corresponding virtual page number. 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* 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 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 of the data 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.

2. If the virtual page has not been accessed before, i.e., no mapping has yet been determined for the virtual page, the proposed technique creates a mapping from this page to an available page in the uncompressed region. If there is no available page in the uncompressed region at that time, 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 compressed page is decompressed and moved to the uncompressed region. Again, if there is no available page in the uncompressed region at that time, a victim page is moved to the compressed region to make 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 2 illustrates the `write_handle` procedure. The three vertical paths prior to the final store instruction correspond to the three situations discussed above. The left path shows the situation in which a

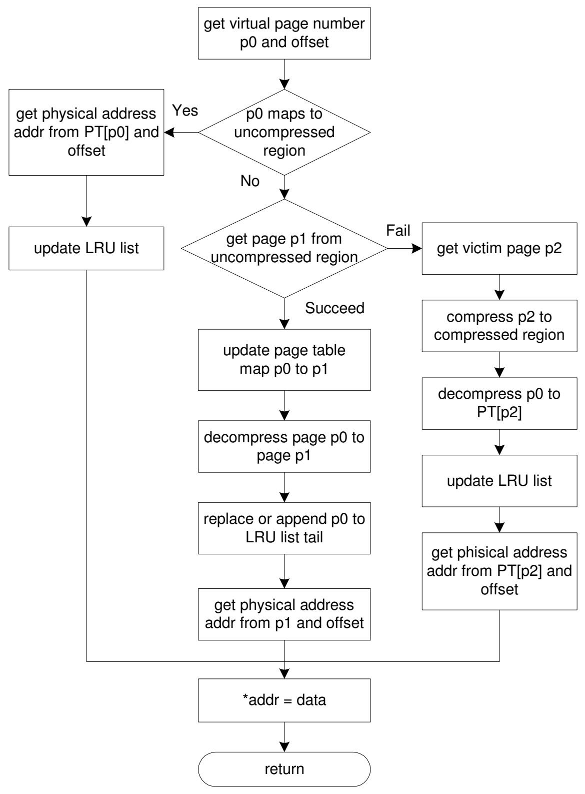
get virtual page number p0 and offset
p0 maps to uncompressed region
Yes
No
get physical address addr from PT[p0] and offset
update LRU list
get page p1 from uncompressed region
Fail
Succeed
get victim page p2
compress p2 to compressed region
decompress p0 to PT[p2]
update LRU list
get phisical address addr from PT[p2] and offset
update page table map p0 to p1
decompress page p0 to page p1
replace or append p0 to LRU list tail
get physical address addr from p1 and offset
*addr = data
return

**Figure 2: Write handle procedure**

virtual page *p0* maps to a page *PT[p0]* in the uncompressed region. Its physical address is computed by adding the physical page address and the offset. In the other two paths, virtual page *p0* maps to a compressed page. In the middle path, a free page *p1* is obtained from 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 exhausted, as shown in the right path, a victim page *p2* from the uncompressed region is compressed, so the physical page previously used by *p2* is freed and used to decompress *p0*. Finally, *p0* will be 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 will be 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 a least-recently used (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 times 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 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 list item is removed from the

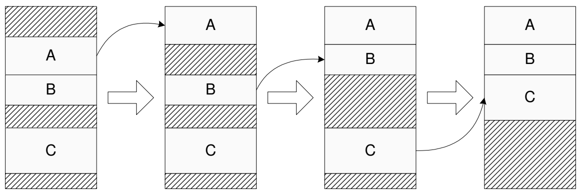

**Figure 3: Memory coalescing**

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. Memory allocation in the compressed region is dynamic because when a page is decompressed and moved back to the uncompressed region, the memory it occupies is released immediately for reuse. Compressed memory management is akin to *heap management*. It imposes memory overhead for keeping information such as page sizes and addresses. Free slots in the compressed region are stored in a free-list. When looking for a fit slot for a compressed page in the free-list, we use the *best fit* policy, which allocates the smallest free slot equal to or larger than the required size.

## 4.3 Preventing Fragmentation

One potential problem with dynamic memory allocation is fragmentation. 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 moved from the compressed region. Free block handles are maintained in a list in order of the physical address of their 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 3 illustrates this procedure. Rectangles A, B, and C represent three compressed pages and shaded rectangles represent freed blocks. After three iterations of moving A, B, and C upwards, all freed blocks are merged into one big free block. This coalescing algorithm has a time complexity of $O(n^2)$, 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 a pagesize of 256 bytes will have 18 compressed pages ($n = 18$), assuming that MEMMU expands memory by 50%.

## 4.4 Interrupt Management

The primary target platform for MEMMU is wireless sensor network nodes, which are typical 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. However, 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.

Consider an application for environmental data sampling in which missing samples is not desirable. Although the optimization techniques described in Section 4.5 can be deployed to reduce the overall execution time overhead, they cannot reduce the worst-case data access delay. In the worst case, the page of incoming data is in the compressed region, but there is neither available space in the uncompressed region to decompress this page nor space in the compressed region to compress a victim page. In this situation, coalescing, compression, and decompression must be performed before the data can be written to memory, i.e.,

$$worst\_case\_delay = t_{coalesce} + t_{comp.} + t_{decomp.}$$

where the $t$ values are durations. The upper bound of coalescing is when all blocks in compressed region are moved upward. This latency can be approximated by the time required to copy the whole compressed region plus the time required by the coalescing algorithm. Using the compression algorithm introduced in Section 4.6, the time of compress and decompress one page is 33 $\mu$s.

Missing sampling events can be avoided as long as the sampling period is longer than the worst-case delay. However, arbitrarily reducing sampling rate is not 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. Based on our experiment, for an application sampling at 19,600 bps, the ring buffer need be at most 16 bytes long. Note that MEMMU does not require the use of a ring buffer when sampling rate is low. However, it provides a convenient and low-overhead method of preventing missed interrupts. In order to use ring buffer, one sets the ring buffer length based on estimated worst-case delay, inserts the `write_rbuf` function call, and posts the `process_rbuf` task to transfer data from ring buffer to the application data structure.

## 4.5 Optimization Techniques

In the previous sections, we described the basic design components of the MEMMU memory expansion system. Every memory access requires (1) a runtime handle check to ensure the address is in the uncompressed region, (2) an update to the LRU list, and (3) a virtual to physical address translation. This introduces high execution time overhead that is proportional to the total number of memory accesses. Hence, the basic solution is not practical for many real applications on embedded systems. However, the technique can be optimized to significantly reduce the number of runtime checks, LRU list updates, and address translations. In this section, we introduce several such compile-time optimization techniques.

1. **Frequent references optimization:** If a small data structure is used very frequently in the application, it should be allocated to the reserved region at compile time to eliminate all handle checks and address translations. For example, in the image convolution application shown in Figure 4(a), the matrix $K[M][M]$ of coefficients is accessed in every iteration of the loop 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.

2. **Run-time handle check optimization:** This is based on the observation that if a sequence of memory references access a 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 incremental values are supported. By inserting checks to decide whether the data element to be accessed next is in a different page than the previous one, the number of handle checks for all accesses to the same page can be reduced to one. This can be especially useful for a hardware-triggered sample arrival event that writes data into the buffer, as illustrated in Figure 6. *Data_ready* is a hardware-triggered event. The *if* statement in the optimized code filters all the handle checks mapping to the same page that was checked in the previous reference.

3. **Loop transformation and compile-time elimination of inner-loop checks:** This optimization scheme can further reduce runtime handle checks by means of compile-time loop transformations. Access to an array in a loop usually uses an incremental memory reference pattern. Figure 5(a) illustrates an example of sequential reference 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 and replaces writes to memory with our `write_handle` routine. The entire loop requires $N$ handle checks. Figure 5(c) illustrates the optimized solution. With loop transformation, the previous loop is broken into nested loops. The inner loop accesses only memory inside a same page; handle checks for the inner loop can be replaced by one check in the outer loop. The total number of handle checks is reduced from $N$ to $N/PAGESIZE$. For the sake of simplicity, we assume that the array $A$ is aligned with pages. Misaligned arrays can be managed by adding two additional loops to handle the non-aligned head and tail portions of the array.

4. **Handle check hoisting:** By hoisting handle checks, multiple handle checks inside a loop can be replaced 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 and locking them in the uncompressed region until a portion of code finishes execution. Figure 4 gives an example of handle check hoisting. Figure 4(a) is the original code for image convolution. Without handle check hoisting, MEMMU would require $(H-M+1)\times(W-M+1)\times(2\times M\times M+1)$ handle checks. It can be decided at compile time that the second inner loop, 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 (b). This saves 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.

5. **Pointer dereferencing to reduce address translation:** There are usually dependencies among sequences of reference addresses. Many applications use a constant stride in memory reference sequences. If the physical address of memory object $A$ is known, and object $B$ is in the same page as $A$, the physical address of $B$ can be determined by dereferencing and adding an offset to the pointer to $A$ instead of computing results based on page table contents. Figure 5(c) shows that this optimization scheme can eliminate $N-N/PAGESIZE$ address translations.

We are still in the process of evaluating the relative benefits of the proposed optimization techniques and determining which to include in the released version of MEMMU. The *frequent reference optimization* is implemented by modifying LLVM [12] to allocate all small data structures in the reserved region. 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. The *run-time handle check optimization* is carried out in a compiler pass, in which LLVM creates two page number variables, current page number and last page number, for each `check_handle` and puts every `check_handle` in an *if* statement. `Check_handle` is called only when the current page number differs from the previous page number. We are not yet certain whether the *loop transformation* or *handle check hoisting* optimizations are general enough, and improve performance enough, to justify their full automation or inclusion within MEMMU. The *pointer dereferencing* optimization is implemented by replacing calls to the `write_handle` and the `read_handle` functions with direct access via a pointer. The base pointer is computed by translating the first encountered virtual address to a physical address. Subsequent pointers in the same page are computed by adding offsets to the base pointer.

## 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.

```
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: $n \leftarrow \sum_{p=0}^{M-1} \sum_{q=0}^{M-1} K_{pq}$
 2: for $i \in \{0 \cdots H-M\}$ do
 3:   for $j \in \{0 \cdots W-M\}$ do
 4:     $t \leftarrow 0$
 5:     for $a \in \{0 \cdots M-1\}$ do
 6:       for $b \in \{0 \cdots M-1\}$ do
 7:         $p \leftarrow A[i+a][j+b]$
 8:         $q \leftarrow K[a][b]$
 9:         $t \leftarrow t + p \times q$
10:       end for
11:     end for
12:     $B[i][j] \leftarrow t/n$
13:   end for
14: end for
```

(a)

```
Input: array $A$ allocated by vm_malloc($H \times W$)
Input: 2-D array $K[M,M]$
Output: array $B$ allocated by
    vm_malloc($(H-M+1) \times (W-M+1)$)
 1: $n \leftarrow \sum_{p=0}^{M-1} \sum_{q=0}^{M-1} K_{pq}$
 2: for $i \in \{0 \cdots H-M\}$ do
 3:   Bring in pages to be used in the following loop
      to uncompressed region
 4:   for $j \in \{0 \cdots W-M\}$ do
 5:     $t \leftarrow 0$
 6:     for $a \in \{0 \cdots M-1\}$ do
 7:       for $b \in \{0 \cdots M-1\}$ do
 8:         $p \leftarrow read\_handle(A + (i+a) \times W + j + b)$
 9:         $q \leftarrow K[a][b]$
10:         $t \leftarrow t + p \times q$
11:       end for
12:     end for
13:     write_handle($B + i \times (W - K + 1) + j,\ t/n$)
14:   end for
15: end for
```

(b)

**Figure 4: Example of (a) original and (b) transformed convolution application**

```
Variable: array $A[N]$
1: for $i \in \{0 \cdots N\}$ do
2:   $A[i] \leftarrow x$
3: end for
```

(a)

```
Variable: array $A$ allocated by vm_malloc($N$)
1: for $i \in \{0 \cdots N\}$ do
2:   check_handle($A + i$)
3:   write_handle($A + i,\ x$)
4: end for
```

(b)

```
Variable: array $A$ allocated by vm_malloc($N$)
1: $pnum \leftarrow N/PAGESIZE$
2: for $i \in \{0 \cdots pnum\}$ do
3:   check_handle($A + i \times PAGESIZE$)
4:   for $j \in \{0 \cdots PAGESIZE\}$ do
5:     write_handle($A + i \times PAGESIZE + j,\ x$)
6:   end for
7: end for
```

(c)

```
Variable: array $A$ allocated by vm_malloc($N$)
1: $pnum \leftarrow N/PAGESIZE$
2: for $i \in \{0 \cdots pnum\}$ do
3:   check_handle($A + i \times PAGESIZE$)
4:   $base\_ptr \leftarrow$ virtual_to_physical($A + i \times PAGESIZE$)
5:   for $j \in \{0 \cdots PAGESIZE\}$ do
6:     $*base\_ptr \leftarrow x$
7:     $base\_ptr{+}{+}$
8:   end for
9: end for
```

(d)

**Figure 5: Example of optimizations on an array access**

**Algorithm 1** Delta compression

```
Input: IN word stream
Output: OUT word stream
Variable: DATA word stream, TAPE delta stream
 1: for $i \in \{1, \cdots, N\}$ do
 2:   $\delta \leftarrow IN[i] - IN[i-1]$
 3:   if $\log_2 \delta \leq MAXBITS$ then
 4:     $TAPE[i] \leftarrow \delta$
 5:   else
 6:     $TAPE[i] \leftarrow MAGIC\_CODE$
 7:     $DATA[i] \leftarrow IN[i]$
 8:   end if
 9:   $OUT \leftarrow$ pack($TAPE, DATA$)
10: end for
```

**Algorithm 2** Delta decompression

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

For example, in sensor network deployed for seabird habitat monitoring [20] 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 contain 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. Our implementation of the delta compression and decompression algorithms are presented in Algorithm 1 and Algorithm 2. The algorithms are based on the observation that the majority of the deltas can be stored within a pre-defined *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

```
1: check_handle($buf + count$)
2: write_handle($buf + count,\ data$)
3: $count{+}{+}$
```

(a)

```
1: $cur\_page \leftarrow (buf + count)/PAGESIZE$
2: if $cur\_page \neq last\_page$ then
3:     check_handle($buf + count$)
4: end if
5: write_handle($buf + count,\ data$)
6: $count{+}{+}$
7: $last\_page \leftarrow cur\_page$
```

(b)

**Figure 6: Example of (a) original and (b) transformed `data_ready(data)` function**

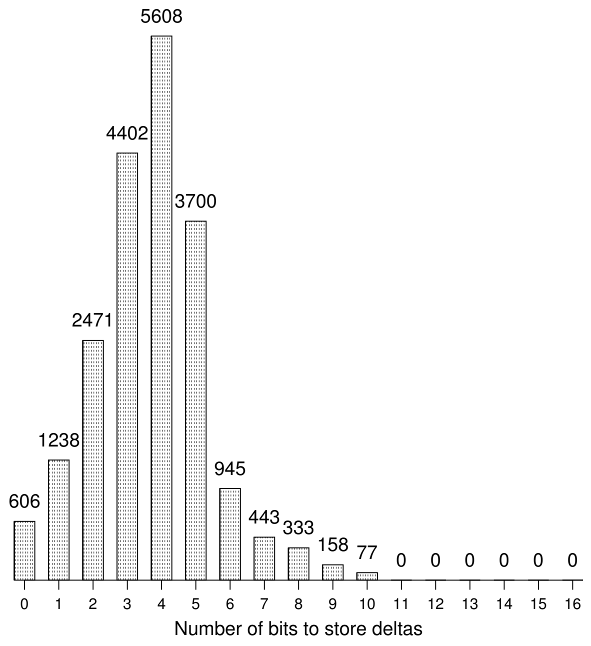

**Figure 7: Histogram of compression bits**

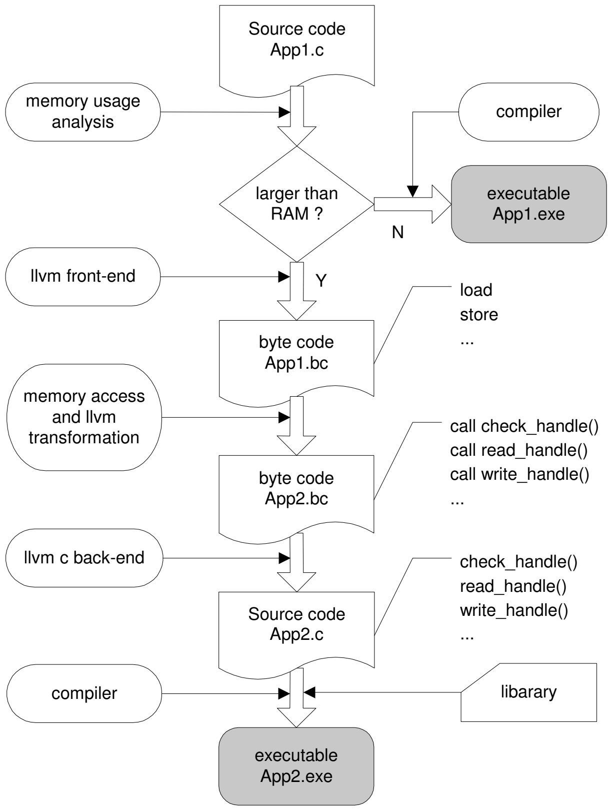

**Figure 8: Overview of technique**

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 before compression, 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 statistically 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 2 bytes (16 bits) at a time and computes the delta on a 2 bytes basis. Figure 7 shows that 95% of the deltas can be represented using six 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 Summary

Figure 8 illustrates the procedure for automatically generating an executable from mid/high-level language source code such as ANSI C with MEMMU. First, the memory requirement of the application is analyzed. If it is smaller than physical RAM, compression is not necessary and therefore no transformations are performed. Otherwise the application code is transferred to LLVM 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`, meanwhile 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 definition of our handle accessing functions to generate an executable.

In the memory initialization routine, physical memory is divided into three regions and the size of each region is computed based on the application memory requirement and the estimated compression ratio of MEMMU. Since runtime data compression ratio cannot be accurately decided at compile time, it is possible that the runtime compression ratio is higher than the predicted compression ratio, causing execution to stop when both memory regions are full. Predicting the worst-case compression ratio may guarantee the execution of application, however it can degrade performance as a result of more frequent

**Table 1: Filtering Benchmark**

| | Orig. app. | Unopt. MEMMU | Opt. MEMMU |
|---|---|---|---|
| RAM usage (B) | 9,935 | 7,243 | 7,243 |
| Buffer size (B) | 9,728 | 9,728 | 9,728 |
| Processing time (s) | 1.24 | 2.31 | 1.35 |
| Active Power (mW) | 6.77 | 6.97 | 6.80 |
| Average power (mW) | 3.94 | 5.92 | 4.27 |

**Table 2: Convolution Benchmark**

| | Orig. app. | Unopt. MEMMU | Opt. MEMMU |
|---|---|---|---|
| RAM usage (B) | 9,739 | 9,739 | 9,739 |
| Input image size (B) | 4,900 | 6,084 | 6,084 |
| Output image size (B) | 4,624 | 5,776 | 5,776 |
| Execution time (s) | 1.50 | 4.47 | 4.07 |
| Processing rate (B/s) | 6,349.33 | 2,653.24 | 2,914.00 |
| Power (mW) | 6.57 | 6.82 | 6.75 |

compression and decompression. Therefore, it is suggested that users determine the compression ratio on sample data of their application and set the MEMMU compression ratio appropriately.

In our experiments, MEMMU is tested on TelosB motes that support TinyOS [6]. TinyOS and its applications are written in nesC [7]. NesC is an extension to the C programming language that supports the structure and execution model of TinyOS. TinyOS itself does not support dynamic memory allocation, so there are only stack and global variables in the nesC program; this makes analysis of the application memory requirement easier. Unfortunately LLVM does not have nesC front-end and back-end, so users must extract the application kernel and compile it with our modified LLVM, then put the generated C code into original nesC program and compile it with the ncc compiler to generate executable running on the mote.

## 5. EXPERIMENTAL RESULTS

This section presents the evaluation results of MEMMU using five representative applications in the wireless sensor network domain. 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 MSP430 microcontroller. It has 10 KB RAM and typically runs TinyOS. The benchmarks are tested under 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: memory usage, average power consumption, execution time, and processing rate. 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 $10\,\Omega$ resistor in series with the power supply. The experimental results show that, with the exception of the image convolution benchmark, the execution time overhead of all other four benchmarks are below 10%.

### 5.1 Sound Filtering

The first benchmark is a sound filtering application. When timer periodically fires, 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 9, the power waveform is close to 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 1 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, optimization methods 1, 3, and 4 were applied. The processing time and average power consumption overhead of unoptimized MEMMU are 86.3% and 50.3%, while after optimization, the overheads are reduced to 8.9% and 8.4%, respectively. Figure 9 depicts the power consumption under three scenarios. We believe the power overhead mainly comes from the following two sources.

1. The mote stays in active mode longer when MEMMU is used. The original application has the shortest data processing time while unoptimized MEMMU has an 86.3% execution time penalty. The optimization techniques significantly reduce the performance penalty, increasing the processing rate by 41.6% compared to the unoptimized version.

2. Active power consumption also increases slightly as a result of MEMMU's computations; the optimized MEMMU has a 0.4% active power penalty.

### 5.2 Image Convolution

Our second benchmark implements a convolution algorithm in which a matrix is convolved with a $3 \times 3$ coefficient kernel matrix. Note that 2-D convolution needs to be used for visual images. In order to permit consistent input to allow fair comparisons for each test case, identical images were transferred to the mote via USB. The input is a gray-scale image of a cloudy sky. Table 2 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 allows the application to handle images that require more memory than is physically available: the unmodified TelosB can only handle an input image of size smaller than 4.8 KB, while MEMMU allows the mote to process images that are 25% larger (6 KB). Since the delta compression algorithm is less efficient for 8-bit images, the compression ratio in this case is 62.4%. We believe a more efficient compression algorithm designed for image data will result in a higher usable memory improvement ratio.

Unfortunately, the increase in image size comes with a cost: using MEMMU results in a 58.2% decrease in processing rate and 3.8% increase in power consumption. To reduce performance and power consumption penalties, optimization meth-

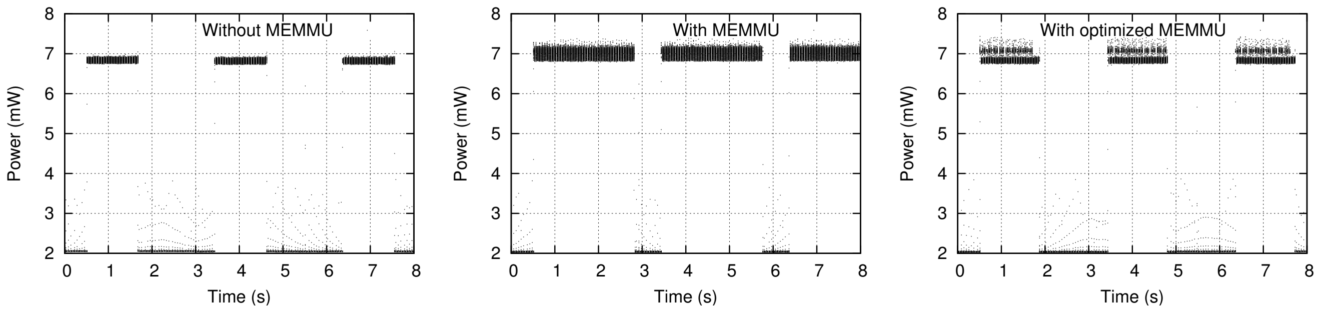

**Figure 9: Power consumption of the sound-filtering benchmark under three settings**

**Table 3: Light Sampling Benchmark**

| | Orig. app. | Unopt. MEMMU | Opt. MEMMU |
|---|---|---|---|
| RAM usage (B) | 9,735 | 9,735 | 9,735 |
| Buffer size (B) | 9,488 | 13,312 | 13,312 |
| Processing time (s) | 8.20 | 11.89 | 11.82 |
| Processing rate(B/s) | 1,157.07 | 1,119.60 | 1,126.23 |
| Power (mW) | 99.50 | 105.35 | 105.04 |

**Table 4: Covariance Matrix Comp. Benchmark**

| | Orig. app. | Unopt. MEMMU | Opt. MEMMU |
|---|---|---|---|
| RAM usage (B) | 9,643 | 9,643 | 9,643 |
| Buffer size (B) | 9,430 | 13,056 | 13,056 |
| Processing time (s) | 0.47 | 2.53 | 0.68 |
| Processing rate (B/s) | 19,894.51 | 5,170.69 | 19,200.00 |
| Power (mW) | 103.88 | 122.46 | 122.46 |

ods 1, 4, and 5 (as described in Section 4.5) were applied to this benchmark. After optimization, the processing rate and power consumption overhead was reduced to 54.1% and 2.7%, respectively. Please note that the image convolution benchmark was the only benchmark for which MEMMU has a performance overhead higher than 10% after optimization. The performance penalty for this benchmark is especially high. We suspect that this is caused by frequent updates to the LRU list resulting from memory access sequences that frequently change from page to page.

## 5.3 Data Sampling

The third benchmark is a sensor data sampling application. In this benchmark, the mote periodically senses the light level and transmits the sensed data when its buffer is full. Related applications might first attempt to find trends and patterns in the data, making a large buffer size useful. Optimization methods 1, 3, and 5 were applied to this benchmark. Table 3 shows that with MEMMU, buffer size is increased by 40.3% with 3.2% decrease in processing rate and 5.9% overhead on power consumption. Furthermore, optimization techniques reduced the processing rate decrease to 2.7% and the power consumption overhead to 5.6%.

The optimization techniques do not make a big difference for this benchmark because the basic technique without optimization works well. There are two reasons for its good performance. First, part of the execution delay is hidden by the sampling period. Second, the data sampling application writes to the buffer in a strictly sequential pattern, resulting in relative less compression and decompression.

## 5.4 Covariance Matrix Computation

The fourth benchmark is a covariance matrix computation application. 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. Optimization methods 1, 2, and 5 were applied to this benchmark. Table 4 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 huge (the processing rate is decreased by 74%), it is greatly reduced with optimizations. The processing rate using optimized technique is only 3.5% lower than the original application, while the amount of usable memory increases by 38.5%. The penalties on average power consumption of both unoptimized and optimized MEMMU are 17.9%.

**Table 5: Correlation Computation Benchmark**

| | Orig. app. | Unopt. MEMMU | Opt. MEMMU |
|---|---|---|---|
| RAM usage (B) | 6,669 | 6,669 | 6,669 |
| Signal size (B) | 6,460 | 9,728 | 9,728 |
| Processing time (s) | 7.98 | 28.3 | 13.00 |
| Processing rate (B/s) | 809.52 | 343.75 | 748.31 |
| Power (mW) | 102.87 | 103.39 | 103.35 |

## 5.5 Correlation Calculation

The last benchmark performs sound propagation delay estimation based on correlation calculation. This application is used to determine the relative locations of sensors. Optimization methods 1, 2, and 5 were applied to this benchmark. As shown in Table 5, MEMMU increases the size of the input data by 50.6%. Although unoptimized MEMMU lowers 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 as low as 0.5%.

## 6. CONCLUSIONS

We have described MEMMU, an efficient software-based technique to increase usable memory in MMU-less embedded systems via automated on-line compression and decompression of in-RAM data. A number of compile-time and run-time optimizations are used to minimize its impact on the performance and power consumption of the target systems. 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 optimization technique effectively improve MEMMU's performance and that MEMMU is capable of significantly increasing usable memory with small performance and power consumption penalties. We plan to release MEMMU for free academic and non-profit use [17].

## 7. REFERENCES

[1] ABRACH, H., BHATTI, S., CARLSON, J., DAI, H., ROSE, J., SHETH, A., SHUCKER, B., AND HAN, R. MANTIS: System support for MultimodAl NeTworks of In-situ Sensors. In *Proc. Int. Wkshp. Wireless Sensor Networks and Applications* (Sept. 2003), pp. 50–59.

[2] BISWAS, S., SIMPSON, M., AND BARUA, R. Memory overflow protection for embedded systems using run-time checks, reuse and compression. In *Proc. Int. Conf. Compilers, Architecture & Synthesis for Embedded Systems* (Sept. 2004), pp. 280–291.

[3] CHOUDHURI, S., AND GIVARGIS, T. Software virtual memory management for MMU-less embedded systems. Tech. rep., Center for Embedded Computer Systems, University of California, Irvine, Nov. 2005.

[4] DOUGLIS, F. The compression cache: Using on-line compression to extend physical memory. In *Proc. USENIX Conf.* (Jan. 1993), pp. 519–529.

[5] GANESAN, P., VENUGOPALAN, R., PEDDABACHAGARI, P., DEAN, A., MUELLER, F., AND SICHITIU, M. Analyzing and modeling encryption overhead for sensor network nodes. In *Proc. Int. Conf. on Wireless Sensor Networks and Applications* (Sept. 2003), pp. 151–159.

[6] GAY, D., LEVIS, P., AND CULLER, D. Software design patterns for TinyOS. In *Proc. Languages, Compilers, and Tools for Embedded Systems* (June 2005), pp. 40–49.

[7] GAY, D., LEVIS, P., CULLER, D., AND BREWER, E. nesC 1.1 language reference manual, May 2003.

[8] GEHRKE, J., AND MADDEN, S. Query processing in sensor networks. *Pervasive Computing 3*, 1 (Jan. 2004), 46–55.

[9] GUESTRIN, C., BODI, P., THIBAU, R., PASKI, M., AND MADDE, S. Distributed regression: an efficient framework for modeling sensor network data. In *Proc. Int. Symp. on Information Processing in Sensor Networks* (Apr. 2004), pp. 1–10.

[10] HELLERSTEIN, J. M., AND WANG, W. Optimization of in-network data reduction. In *Proc. of Int. Wkshp. on Data Management for Sensor Networks* (Aug. 2004), pp. 40–47.

[11] KARLOF, C., AND WAGNER, D. Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's AdHoc Networks J. 1*, 2–3 (Sept. 2003), 293–315.

[12] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. Code Generation and Optimization* (Mar. 2004), pp. 75–86.

[13] LEKATSAS, H., HENKEL, J., AND WOLF, W. Code compression for low power embedded system design. In *Proc. Design Automation Conf.* (June 2000), pp. 294–299.

[14] LI, D., WONG, K., HU, Y., AND SAYEED, A. Detection, classification, and tracking of targets. *Signal Processing Magazine 19*, 2 (Mar. 2002), 17–29.

[15] LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo`.

[16] MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. TAG: a tiny AGgregation service for ad-hoc sensor networks. In *Proc. Symp. on Operating Systems Design and Implementation* (Dec. 2002), pp. 131–146.

[17] Memory expansion on embedded systems without MMUs. MEMMU link at `http://www.eecs.northwestern.edu/~dickrp/projects.html`.

[18] NATH, S., GIBBONS, P. B., SESHAN, S., AND ANDERSON, Z. R. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. Int. Conf. on Embedded Networked Sensor Systems* (Nov. 2004), pp. 250–262.

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

[20] POLASTRE, J., SZEWCZYK, R., MAINWARING, A., CULLER, D., AND ANDERSON, J. Analysis of wireless sensor networks for habitat monitoring. *Wireless sensor networks* (2004), 399–423.

[21] POTTIE, G. J., AND KAISER, W. J. Wireless integrated network sensors. *Commun. ACM 43*, 5 (May 2000), 51–58.

[22] PRADHAN, S. S., KUSUMA, J., AND RAMCHANDRAN, K. Distributed compression in a dense microsensor network. *IEEE Signal Processing Magazine 19*, 2 (Mar. 2002), 51–60.

[23] RIZZO, L. A very fast algorithm for RAM compression. *Operating Systems Review 31*, 2 (Apr. 1997), 36–45.

[24] TREMAINE, B., FRANASZEK, P. A., ROBINSON, J. T., SCHULZ, C. O., SMITH, T. B., WAZLOWSKI, M., AND BLAND, P. M. IBM memory expansion technology. *IBM J. of Research and Development 45*, 2 (Mar. 2001), 271–285.

[25] TUDUCE, I. C., AND GROSS, T. Adaptive main memory compression. In *Proc. USENIX Conf.* (Apr. 2005), pp. 237–250.

[26] WILSON, P. R., KAPLAN, S. F., AND SMARAGDAKIS, Y. The case for compressed caching in virtual memory systems. In *Proc. USENIX Conf.* (Apr. 1999), pp. 101–116.

[27] YANG, L., DICK, R. P., LEKATSAS, H., AND CHAKRADHAR, S. CRAMES: Compressed RAM for embedded systems. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis* (Sept. 2005).

[28] YANG, L., LEKATSAS, H., AND DICK, R. P. High-performance operating system controlled memory compression. In *Proc. Design Automation Conf.* (July 2006).