# The Design of a Bloom Filter Hardware Accelerator for Ultra Low Power Systems

Michael J. Lyons and David Brooks
School of Engineering and Applied Sciences, Harvard University, Cambridge, MA
{mjlyons,dbrooks}@eecs.harvard.edu

## ABSTRACT

Battery-powered embedded systems require low energy usage to extend system lifetime. These systems must power many components for long periods of time and are particularly sensitive to energy use. Recent techniques for reducing energy consumption in wireless sensor networks, such as aggregation, require additional computation to reduce energy intensive radio transmissions. Larger demands on the processor will require more computational energy, but traditional energy reduction approaches, such as multi-core scaling with reduced frequency and voltage may prove heavy handed and ineffective for motes (sensor network nodes). Alternatively, application-specific hardware design (ASHD) architectures can reduce computational energy consumption by processing operations common to specific applications more efficiently than a general purpose processor. By the nature of their deeply embedded operation, motes support a limited set of applications, and thus the conventional general purpose computing paradigm may not be well-suited to mote operation. This paper examines the design considerations of a hardware accelerator for compressed Bloom filters, a data structure for efficiently storing set membership. We evaluate our ASHD design for three representative wireless sensor network applications and demonstrate that ASHD design reduces network latency by 59% and computational energy by 98%, showing the need for architecting processors for ASHD accelerators.

## Categories & Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

## General Terms

Design, Performance

## Keywords

Hardware Accelerator, Bloom Filter, Wireless Sensor Network

## 1. INTRODUCTION

Battery-powered embedded systems carefully manage energy consumption to maximize system lifetime. Wireless sensor networks (WSNs), made up of many "mote" devices, are often designed to operate for months without intervention. Sensor networks are typically used to monitor an environment and may be deployed in remote and hazardous locations. WSNs can consist of a hundred motes or more, and cover wide areas. As a result, mote software and hardware must consider energy consumption at every level.

Motes are simple, pocket-sized computers. Each mote contains a small battery that powers a radio for wireless networking, a limited amount of memory, and a constrained processor. Aggregation, a widely researched field for reducing data transmissions by combining data on motes, reduces energy use by spending additional energy on computation to save a greater amount of energy on the power-hungry radio [12]. Increasing on-mote processing complexity will require additional computational hardware, demanding more energy. As sensor networks grow and generate larger data sets, these energy costs will continue rising.

Unlike PCs, embedded systems often execute a limited set of applications and have less need for general purpose functionality. Simple bit manipulations poorly utilize a general purpose processor. Complex operations, such as multiplication, require several cycles on a general purpose processor. Many embedded applications require support for these simple and complex operations and most existing systems must poorly utilize a general purpose microcontroller. In contrast, application-specific hardware design (ASHD) tailors hardware to the application. We refer to these ASHD constructs as *hardware accelerators*. If any of these hardware accelerators are unused, they can be Vdd-gated to almost eliminate energy wasted on unused features.

This paper explores ASHD considerations during the design of one such hardware accelerator. The accelerator implements several operations for compressed Bloom filters, a data structure for efficiently storing set membership. These operations include support for inserting items, compression and decompression, and querying. We demonstrate significant performance, power, and energy results over a general-purpose hardware solution for each custom operation. In addition, we explore the benefits of ASHD in the context of three WSN applications: mote health monitoring, object tracking, and duplicate packet removal. For these benchmarks, Bloom filters improve network reliability and reduce radio transmissions by up to 70%. We demonstrate that the ASHD hardware accelerator implementation provides significant gains in network latency (59%), computational delay (85-88%), and computational energy (98%) compared to executing the Bloom filter code on general purpose microcontrollers. Given these improvements, we show the benefits of architecting processors for ASHD hardware accelerators. The Bloom filter accelerator can be Vdd-gated when not in use so that the Bloom filter accelerator only uses energy when it will reduce total system energy consumption by an even greater amount.

This paper is organized as follows. Section 2 discusses related approaches to increase energy-efficiency and motivates the ASHD paradigm. Section 3 describes the algorithms needed for the Bloom filter hardware accelerator, and Section 4 discusses the architectural blocks needed to implement the approach. We then quantify power and performance advantages for the Bloom filter accelerator for specific operations (Section 5) and larger applications (Section 6). Finally, we conclude the paper.

## 2. RELATED WORK

Parallel processing is well known for increasing energy-efficiency in general purpose computing. Designers distribute computation across several low-power cores rather than a single high-power core [14]. The low-power cores operate at a lower frequency, reducing voltage and power requirements. Several cores can be combined in one processor to meet computational goals. Assuming the lowest possible voltage is used, dynamic power is roughly proportional to $nf^3$, where $n$ is the number of cores and $f$ is the operating frequency; potential processing capacity is proportional to $nf$.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ISLPED'09,* August 19–21, 2009, San Francisco, California, USA.
Copyright 2009 ACM 978-1-60558-684-7/09/08 ...$10.00.

Ideally, power demands are minimized when many low-frequency cores are used. However, several factors limit the power reduction:

- Threshold voltage places a lower bound on voltage scaling. Subthreshold operation is possible but adds significant design challenges [20].
- Leakage current increases the power consumption of each additional core.
- Interconnect logic for communication between cores and shared memory requires additional power and may introduce bottlenecks.
- Software must be parallelized to run on all cores simultaneously

Application-specific hardware design (ASHD) provides an alternative approach that may be more appropriate for embedded systems due to the more specialized nature of the workloads. Motes do not require the same amount of general purpose functionality as a conventional computing system and can be customized for better performance and lower power. Furthermore, ASHD leverages the same power-saving properties of parallelism by operating at low voltage-frequency combinations with high performance, while also capturing the benefits of explicit hardware support for simple operations such as bit-manipulations that are inefficient on general purpose cores. Additionally, ASHD systems do not require core interconnect logic or the software challenges of parallelized code.

Tensilica's Xtensa design adds custom instructions to a processor [8]. In this approach, designers integrate custom instructions into a general purpose processor core. This approach is similar to our hardware accelerator architecture, however, no partition exists between general purpose hardware and customized instructions. The hardware accelerator interface used in this paper and described in Section 4 includes support for fine-grain Vdd-gating, powering accelerators down when not in use. Partial Vdd-gating is very difficult in a monolithic general purpose core. Additionally, all operations in the Xtensa design must fit within the processor's instruction set and architecture. In contrast, hardware accelerators have greater architecture autonomy and several accelerators can perform background work in parallel. Clark, et al. [4] automates instruction set generation from application source for Xtensa-like designs. The authors create customizations by merging commonly adjacent instructions. Although this methodology implements hardware automatically, it is unlikely to create highly complex operations, such as those designed in this paper. Additionally, this approach is limited to the same instruction set and architecture confines as the Xtensa approach.

Other research has focused on building a heterogeneous processor from pre-existing general purpose cores [18]. This approach attempts to design custom processors without any custom circuit design. In addition to the limitations of the customized instruction processors, the heterogeneous core is also limited by the cores available. Unusual operations may not have a corresponding core.

# 3. BLOOM FILTER ALGORITHMS

Bloom filters provide a useful case study for an exploration of sensor network mote ASHD. Using Bloom filters, many WSN applications can easily aggregate information and reduce the size of large data sets containing unique identifiers. These factors can reduce costly radio transmissions and lower overall mote energy usage. However, some Bloom filter operations may require several seconds of compute time on general purpose hardware, limiting the applicability of the approach and incurring high energy usage. By implementing hardware support for Bloom filters, WSN applications can achieve significant energy reductions without sluggish performance. The Bloom filter hardware accelerator improves performance and energy use by optimizing several algorithms in hardware. The accelerator natively supports Bloom filters, multiply and shift hashing, and Golomb-Rice coding support for data aggregation, near-random hashing, and data compression, respectively. The following sections describe these algorithms in detail.

## 3.1 Bloom filters

Bloom filters efficiently store set membership of large items by combining data in a large bit array. Using a small number of hash functions, $h_1 \dots h_k$, Bloom filters reduce storage costs up to 70% [1]. Many applications, including spelling checkers and distributed web caches currently use Bloom filters. Other work has also suggested the use of Bloom filters in hardware [16, 15, 6]. However, these works use Bloom filters for internal processor or network management and do not expose Bloom filters to applications. We discuss several wireless sensor network applications utilizing Bloom filters in Section 6.

**Table 1: Bloom filter configurations (16KB bit array, 32-bit elements). Bits per item applies to full Bloom filters**

| Config. | Item Capacity | Bits per Item | Hash Functions ($k$) | False Positive Rate |
|---|---|---|---|---|
| 1 | 13500 | 9.71 | 7 | $< 1\%$ |
| 2 | 9000 | 14.56 | 10 | $< 0.1\%$ |
| 3 | 6500 | 20.16 | 14 | $< 0.01\%$ |

Our hardware accelerator implements a specific range of Bloom filter configurations: the bit array is 16KB, up to 16 hash functions are available, and 32-bit items are supported. Initially, we set every bit in the array to 0, to create an empty Bloom filter. We insert items by hashing the item $x_i$ with every hash function $h_1 \dots h_k$. The results of these hash functions $h_1(x_i) \dots h_k(x_i)$ are addresses to bits in the array, which we set to 1. As we insert more items, the number of 1's in the Bloom filter increases. When inserting items, we may find some bits already set to 1 due to previous item insertions writing to the same bit address.

Querying to check if an item $x_i$ is in the Bloom filter is similar to insertion. We hash the item with every hash function $h_1 \dots h_k$ and check each bit's value at addresses $h_1(x_i) \dots h_k(x_i)$. If any hash function points to a 0 bit, we know with certainty the item is not in the Bloom filter. The item is in the Bloom filter with high probability if all hash functions point to 1 bits, but we cannot know with certainty. These "false positive" errors, although rare, occur when other inserted items hash to the same bits as the queried item. The false positive rate can be pre-configured as required by the application, typically from 1% to 0.01%.

Items cannot be removed from a Bloom filter. Hypothetically, an item could be removed by setting any of the item's corresponding array bits to 0. However, many inserted items may hash to the same bit, and removing one item may inadvertently remove several other items. If a Bloom filter becomes full, all elements can be cleared by setting all bits in the array to 0.

The false positive rate, item capacity, and energy requirements to insert or query an item are determined by $k$, the number of hashes used by the Bloom filter. When $k$ is larger, the false positive rate decreases. However, smaller values of $k$ result in Bloom filters with a larger item capacity and lower energy cost per item insertion or query. This trade-off is illustrated in Table 1. A detailed analysis of Bloom filter configuration is available in [1].

Bloom filters merge by bitwise ORing bit arrays, assuming both Bloom filters use the same bit array lengths and hash functions. This property makes aggregating data in a WSN spanning tree a trivial task: parents can merge Bloom filters from child motes quickly, insert their own items, and transmit the aggregate Bloom filter to its own parent.

The Bloom filter is considered full when half of the array's bits are 1. At this point, further insertions will dramatically increase the false positive rate. Bloom filter storage is most efficient when full, as the bit array is always a constant length. For example, configuration 1 in Table 1 can store 32-bit elements using less than 10 bits when full.

## 3.2 Multiply and Shift Hashing

Multiply and shift hashing, described by Dietzfelbinger et al. [7], is simple, yet effective. Each hash function $h_1 \dots h_k$ requires a hash key $HashKey_1 \dots HashKey_k$. Hash keys are odd integers randomly chosen before the Bloom filter is used. The accelerator represents hash keys as 32-bit integers.

To perform a hash $h_i$ of element $x_j$, we calculate

$$h_i(x_j) = \frac{(HashKey_i \times x_j) \bmod 2^{32}}{2^{32-b}} \quad (1)$$

where $b$ is the number of bits in the Bloom filter bit array address. For the 16KB bit array used by the accelerator, $b = 17$. The modulo and divide are powers of two and can be efficiently implemented with a bit mask and shift.

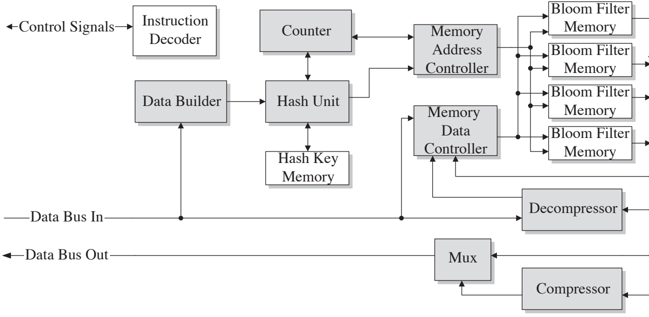


**Figure 1: Bloom filter hardware accelerator hardware flow: arrows indicate the direction of information, shaded blocks indicate modules controlled by the Instruction Decoder.**

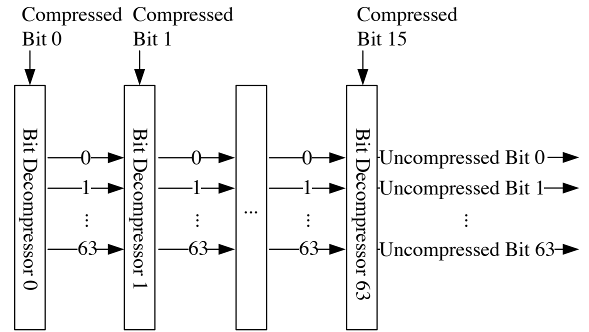


**Figure 2: Decompressor information flow**

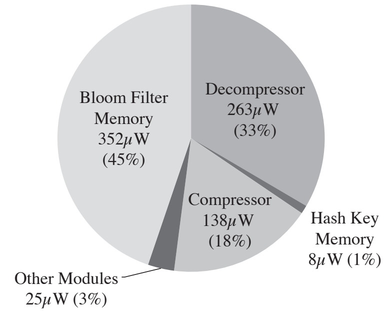


**Figure 3: Average power usage of Bloom filter hardware accelerator modules**

## 3.3 Golomb-Rice Coding

The accelerator implements Golomb-Rice coding, a popular compression and decompression method used in Apple's Lossless Audio Codec (ALAC) and Lossless JPEG (JPEG-LS) [13, 17]. As noted in Section 3.1, a Bloom filter contains more 0s than 1s until filled. Therefore, sparsely filled Bloom filters (under 70% full) can reduce Bloom filter size through Golomb-Rice coding. The algorithm, a form of run length encoding, is simple to implement, and therefore power efficient.

The number of 1s in the bit array are first counted to determine the "remainder part" length $l$. The relation between 1s in the bit array and $l$ is precomputed; only a quick lookup is needed to determine the remainder part length.

Second, the bit array is iterated from start to finish, scanning for run lengths of 0s between 1s. For each run length of $n$ 0s, the remainder part $r = \lfloor \frac{n}{2^l} \rfloor$ and quotient part $q = n \bmod 2^l$ must be calculated.

After calculating $r$ and $q$, we write $r$ 0s to the compressed bit stream, followed by a 1. $q$ is then written directly, using $l$ bits. This process is used to write all run lengths in the uncompressed bit stream until the end is reached. The second step's implementation does not require any expensive divisions or modulos; a counter is kept of the current 0 run length. If the next bit is a 0, the counter is incremented. If the counter reaches $2^l$, a 0 is written to the compressed stream and the counter is reset. If the next bit is a 1, a 1 is written to the compressed stream, followed by the the counter's value using $l$ bits. Therefore, Golomb-Rice compression can be reduced to many simple bit operations.

# 4. ACCELERATOR ARCHITECTURE

This paper leverages the mote architecture described by Hempstead, et al. [10] which provides a framework for custom hardware accelerators. The architecture proposes a lightweight event processor for managing power and offloading tasks to hardware accelerators. High-level events and tasks are decoded on the event processor and deployed to accelerators via memory mapped operations. A simple processor executes any operations not explicitly handled by accelerators. We anticipate implementing hardware accelerators as synthesized standard cells (e.g. ASIC flow) or through a shared on-chip programmable FPGA substrate.

We designed the Bloom filter hardware accelerator to work within the processor architecture of [10] and support a 16-bit data bus. The accelerator consists of several modules, illustrated in Figure 1. In the following sections, we will examine each major module in the Bloom filter accelerator and discuss design decisions for reducing energy and delay.

## 4.1 Bloom Filter Memory

The Bloom filter bit array is stored in four 2K x 16-bit modules. The bit array is stored sequentially by address, so that bits are stored in the following order: Module1[0], Module2[0], Module3[0], Module4[0], Module1[1], and so on. We chose a four-module configuration to provide access to all four memory modules simultaneously, boosting performance by up to 4x. Only one memory access is possible per cycle, so increasing the amount of memory available at a given cycle can greatly improve performance. We also decided to use four modules with a 16-bit data bus rather than one module with a 64-bit data bus because some Bloom filter operations only use one block per cycle. In this case, the unused three blocks can be disabled to reduce dynamic power consumption. We decided not to support an even larger data bus because significant additional logic would be required and wider bus lengths would rarely be fully utilized.

## 4.2 Memory Data Controller

The Memory Data Controller manages data stored in the Bloom filter memory. The accelerator supports several Bloom filter operations, each writing to memory in a distinct style. Insertions and queries only modify one bit at a time, while other operations may modify one block or four blocks per cycle. The Memory Data Controller is responsible for ensuring each operation can write as many or as few bits as is required.

The Memory Data Controller also counts the number of 1s inserted into the Bloom Filter at every cycle. We use this counter during compression operations to eliminate the need for an additional full memory iteration as described in Section 3.3. As previously noted, memory access can be a bottleneck, so this optimization is critical for performance.

## 4.3 Memory Address Controller

The Memory Address Controller coordinates with other blocks to correctly set the addresses of each of the four Bloom filter blocks. Although item insertion and query operations randomly jump from bit to bit in memory, some operations may sequentially read one module at a time, and others read sequentially from all blocks simultaneously. During sequential operations, the Memory Address Controller remembers where processing ended in the last cycle so that the operation can be easily resumed.

## 4.4 Decompressor

The Decompressor reads 16-bit Golomb-Rice encoded Bloom filter blocks from the data bus and unpacks up to 64 bits of uncompressed Bloom filter. The Decompressor guarantees the entire compressed block will be processed, or 64 bits of uncompressed Bloom filter will be unpacked. These derive from the 16-bit data bus and

the 64 bit width of Bloom filter memory. Although these limits require significant additional logic, we support this higher performance design to avoid elevated computation delays when processing Bloom filters containing many elements.

The Decompressor is composed of 16 serially-connected bit decompressors, illustrated in Figure 2. This design allows each compressed bit to be decompressed serially. Although each bit could be decompressed in parallel and reassembled, the serial design allows bit compressors to be disabled when the uncompressed stream is full, thus reducing dynamic power. A dynamic style would increase the speed of decompression, but is unnecessary due to the slow 100 KHz clock frequency used by the Hempstead processor.

## 4.5 Compressor

The Compressor design is similar to the decompressor design and is composed of 64 serially connected single-bit compressors. The Compressor reads 64 bits of uncompressed data from the Bloom filter bit array, producing up to 16 bits of compressed data per cycle. These bit limitations are due to memory access and data bus limitations respectively. As a result, compressed Bloom filters can be produced 4x faster than uncompressed Bloom filters. Supporting these guarantees requires additional logic, but gains in performance make this addition worthwhile.

# 5. ACCELERATOR EVALUATION

In this section, we evaluate the design decisions discussed in Section 4 by comparing power, energy, and performance of the application-specific hardware design against a general purpose hardware design paradigm.

The Bloom filter hardware accelerator, with the exception of Bloom filter memory, was implemented in Verilog and synthesized for the UMC 130nm process using Synopsis Design Compiler, Encounter, and Cadence. The accelerator area is $792,850\mu m^2$ and uses $1.217M$ transistors. Power calculations are based on Design Compiler estimates, using the check_power command on high effort. Bloom filter memory was generated by the Faraday Memaker tool and power estimates were profiled using Synopsis HSIM simulations. The accelerator implements two-phase clocking, operates at 1.2V and supports a 100 KHz clock frequency. Figure 3 shows the distribution of power between the larger elements of the accelerator. When synthesized individually, each module requires roughly 10% additional energy than shown, however, Design Compiler is able to reuse logic between modules to reduce total system energy. Therefore, we assume shared logic savings is proportional to the total energy cost of each module, and scale each module's power equally to match Design Compiler's system estimate.

The general purpose design uses the Bloom filter software implementation for motes in Chang, et. al [2]. This software implements Bloom filter operations exactly as described in Section 3. The software was written in nesC for TinyOS 1.1.15 [11] and tested directly on the TMote Sky mote. The TMote Sky features a relatively powerful 16-bit, 8 MHz TI MSP430 processor with 10KB of memory. We chose to compare our hardware accelerator with the MSP430 processor due to its wide popularity in the sensor network community. For our comparisons, note that the Hempstead mote processor is completely distinct from the TI MSP430 processor.

Due to memory limitations of the TMote Sky, only 8KB of memory is available for the Bloom filter bit array in the general purpose implementation. Since the ASHD implementation uses a 16KB Bloom filter, the ASHD will use additional power supporting the additional memory, as well as extra cycles to work on a Bloom filter twice as large. Yet, the ASHD logic demonstrates significant performance and energy savings despite this handicap.

Timing figures for the ASHD implementation are generated by counting the number of cycles used. We assume total system power for the ASHD design is 886$\mu$W, of which 786$\mu$W is expended by the Bloom filter accelerator. The remaining 100$\mu$W is consumed by the Hempstead event processor and other infrastructure logic.

Timing figures for the general purpose implementation are obtained through experimentation. Operations are executed on the TMote Sky and timed using internal microsecond and millisecond clocks for high accuracy. We assume average TMote power is 4.86mW, derived from the TMote Sky datasheet [5]. Therefore, the general purpose implementation's processor power requirements are almost 450% higher than the ASHD implementation.

For both implementations, we calculate energy as $Power_{avg} \times Time$.

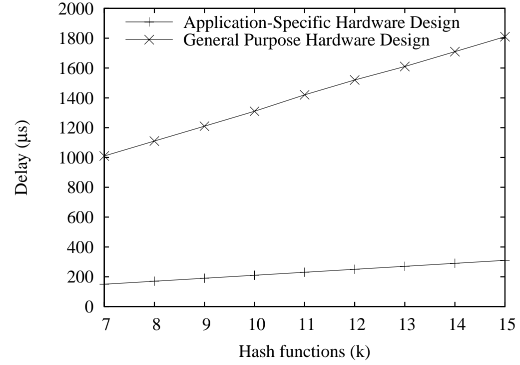


**Figure 4: Item insertion times of application-specific hardware design logic and general purpose design logic**

## 5.1 Item Insertion and Querying

Item insertion is highly efficient in ASHD logic due to native support for the complex multiply and shift hashing operation. ASHD logic requires significantly less time for insertions in all cases, as illustrated in Figure 4. Furthermore, insertion uses 97% less energy per insertion than general purpose logic, regardless of the number of hash functions used.

Several factors contribute to the ASHD advantage. First, the ASHD logic can hash during one cycle due to its native support for multiply and shift hashing. On the contrary, the MSP 430 processor used in the general purpose implementation only supports 16-bit math and must spend many cycles to complete the multiplication operation.

Further, the ASHD implementation does not require additional logic to perform the required bit shift. Instead, the ASHD logic simply uses bits 31 through bits 15 (bit 0 is the least significant bit). The MSP 430 does not natively support this bit selection and must spend additional cycles on a 32-bit shift to obtain the corresponding bit address.

Memory operations limit the speed of as the ASHD implementation, as only one memory read or write can be performed per cycle. Therefore, the ASHD insertion implementation requires 2 cycles per hash (one to read the block, another to write the modified block back). Parallelizing item insertion would require significant additional logic due to the seemingly random bit addressing caused by the hash function. Although support for processing up to four hashes could theoretically be possible due to the four Bloom filter memory modules, the block location of each hash is unknown until calculated. Furthermore, all hashes could theoretically point to the same block, making simultaneous bit insertions impossible.

Querying items contained in the Bloom filter is similar to item insertion: the item is hashed $k$ times and the bit at address $h_i(x_j)$ is verified to be 1. This process takes roughly half the time of insertion on ASHD logic because only one memory read is required for each hash. The general purpose implementation is time-bound by the hash function, however, so performance is largely equal to insertion.

## 5.2 Compressing Bloom Filters

The ASHD accelerator improves Bloom filter compression performance up to 1800%, as shown in Figure 5, and reduces energy consumption up to 99%. The key to these ASHD improvements is custom support for Golomb-Rice coding. When implemented in software for general purpose systems, each uncompressed bit must be examined to count run lengths of 0 bits. When a 1 is encountered, another lengthy set of bit operations must be performed to determine the correct sequence of compressed bits. As we insert more elements into the filter and the frequency of 1s increases, the quantity of run lengths grow and additional work is required to compress. In general purpose software, this additional work increases the compression delay. The ASHD accelerator also requires additional cycles as well, but provides a fast upper bound on compression delay. The ASHD compressor guarantees a compressed 16-bit block will be produced or 64-bits of uncompressed data will be processed every cycle. Therefore, compression can never exceed 81.92ms and is often faster.

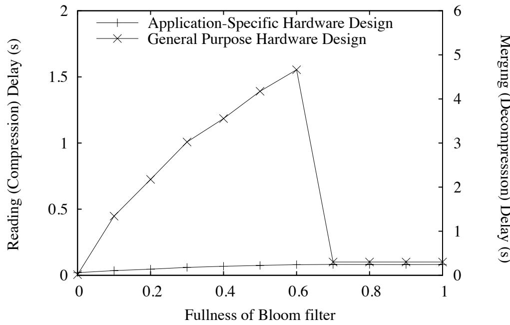


**Figure 5: Bloom filter reading and merging delay at a 1% false positive rate. The ASHD implementation uses a 16KB Bloom filter; the general purpose implementation uses an 8KB Bloom filter. Uncompressed Bloom filters are used after reaching 70% fullness.**

As noted in Section 3.3, the number of 1s in the bit array must be counted to determine $l$, before compressing run lengths. Memory access is slow in the ASHD implementation due to the 100 KHz clock frequency, and iterating through the memory would require an additional 20.48ms. To avoid this penalty, the ASHD implementation counts the number of ones in the filter as they are inserted. Therefore, ASHD requires only one memory pass. The bit tracking technique is not used in the general purpose implementation due to lack of hardware support. Adding support in software would require several cycles per insertion or query, operations frequently used in many applications. Adding bit tracking support for lengthy merge operations would also require significantly larger delays. Performing two passes of memory, a fast process in the 8 MHz general purpose implementation, requires less delay overall in the general purpose system.

At 70% of capacity, run lengths become too small to effectively compress, and Bloom filters are delivered uncompressed. The general purpose implementation no longer compresses and simply reads from memory. Although the ASHD is only 19% faster when the Bloom filter is approaching capacity, recall that the ASHD Bloom filter is twice the size of the general purpose implementation. If both implementations used equivalently sized bit arrays, the ASHD implementation would perform 59% faster. Further, the ASHD could reduce the uncompressed Bloom filter read delay by 75% if a wider data bus is used in a future architecture.

## 5.3 Merging Compressed Bloom Filters

Bloom filter merging uses bitwise ORs to combine a foreign Bloom filter with the filter stored in the bit array. The foreign Bloom filter, delivered over the data bus, is processed over several segments. If compressed, each foreign Bloom filter segment must first be decompressed. Meanwhile, the corresponding Bloom filter segment stored in memory is loaded. The two segments are bitwise ORed and saved back into the bit array memory.

Bloom filter merging performance resembles Bloom filter compression performance, as shown in Figure 5. The ASHD accelerator performs up to 2700% faster and can reduce the energy cost by more than 99%. This performance boost is largely due to the ASHD decompressor design. The decompressor guarantees the entire compressed Bloom filter segment will be decompressed, or four blocks of uncompressed memory will be processed. The first guarantee provides an upper bound of 163.85ms per merge, but the second speeds up the process when the foreign Bloom filter is highly compressed.

The large gains are only obtained when using compressed Bloom filters. Rice-Golomb coding is relatively inefficient in general purpose hardware due to the large number of bitwise operations. However, once uncompressed Bloom filters are used beyond 70% capacity, general purpose performance noticeably improves. The general purpose implementation appears to operate faster beyond 70% capacity due to the memory size disparity between implementations. If both implementations used the same bit array size, the ASHD implementation would require 34% less time and 88% less energy.

When low or no compression is used, the data bus limits performance. Because each Bloom filter segment requires two cycles (once to read from the bit array and once to write), 16,384 cycles are required to perform a merge in the worst case. If a larger data bus were used in a future architecture, merging delay could be reduced by an additional 75%.

# 6. APPLICATION EVALUATIONS

In this section, we examine several distinct Bloom filter-based wireless sensor network applications to demonstrate Bloom filter gains and evaluate ASHD performance and energy improvements. Each application represents a different class of Bloom filter use. The mote status application shares a Bloom filter across a sensor network, so that a central server can check if any motes in a large network require attention. In the object tracking application, each mote in the network individually records the unique identifiers of sensed objects in a private Bloom filter, periodically transmitting the filter to a central server. The duplicate packet removal application uses a Bloom filter to locally store identifiers of each packet received to quickly remove any packets duplicated by routing errors.

Mote networks may contain thousands of motes in the future, and managing mote operation will be critical in maintaining reliability. Mote networks are typically routed in a spanning tree formation and support multi-hop routing. A central server will connect to the root mote to send, analyze, and store information from the sensor network. If a mote is not close enough to directly transmit data to a desired mote, data can hop across several intermediate motes to reach its destination. For example, if a mote wishes to send data to the central server, the mote would send a packet to its parent mote, which sends the packet to its own parent mote, eventually reaching the server by way of the root mote. Extremely large mote networks can easily become saturated if storage and transmissions are not properly managed.

The following examples assume the sensor network uses a two child per parent routing tree structure. We also assume mote radios have a 40kbps effective data rate (does not include transmission overhead) [3]. All calculations are estimations based on the ASHD accelerator analysis from Section 5 and on-mote timing profiling of the general purpose implementation software.

## 6.1 Mote Status

Spanning tree topology can be problematic for the root mote and motes nearby. The root mote must forward every packet sent from the sensor network to the central server; nearby motes must also handle large amounts of network traffic. Recall that Mote radios are slow and power-hungry. In many cases, the root mote will not be able to forward data to the server quickly enough, resulting in dropped packets and poor quality of service. Even if the radio can handle the network traffic, the radio will quickly exhaust the root mote's energy and cut off the sensor network from the server.

Bloom filters can greatly reduce the transmission load on these taxed motes by efficiently aggregating mote status information, such as low battery warnings, within the network. In this example, we will support a sensor network of 128,000 motes to demonstrate this application's ability to monitor extremely large sensor networks by using Bloom filters. We will use a 4-byte unique identifier to represent each mote and assume 10% of the mote batteries are low at any given time. A mote can send a low battery alert by periodically transmitting its unique identifier (UID) to the server via its parent. Leaf motes on the boundaries should individually send these UIDs without Bloom filters. As Figure 6 indicates, Bloom filters are only efficient when at least 15% full, or when about 2000 items are inserted with a 1% false positive rate. As these low battery alerts hop from parent to parent, approaching the root mote, each parent mote will need to forward twice as many alerts as each child. When a mote has received 2000 or more items, it should create an empty Bloom filter and insert these elements. This Bloom filter will then be sent to the following parent mote, which will merge any Bloom filters it receives, insert UIDs sent sequentially without Bloom filters, and insert its own UID if its battery is low. This process of merging Bloom filters and inserting single UID items will continue as Bloom filters approach the root mote. The root mote will finally deliver a single Bloom filter, containing all low battery alerts, to the server. The server can then query the Bloom filter for each UID in the network to discover which motes require attention. Additionally, the server can track alerts over time to reduce errors from false positives: by identifying motes which consistently report low batteries, the server can remove erroneous alerts that sporadically

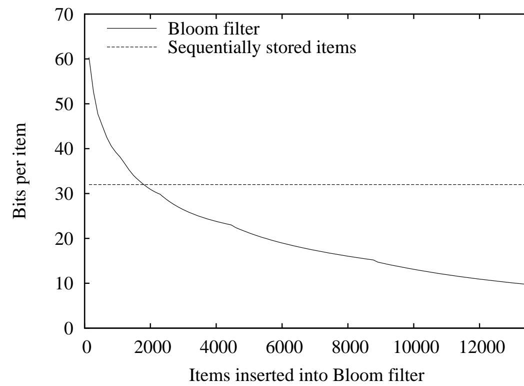


**Figure 6: Storage cost per item for a 16KB Bloom filter and 1% false positive rate. Bloom filters are more efficient than sequentially storing 32-bit items when the bit per item cost is under 32 bits.**

appear. Meanwhile, all motes will clear their Bloom filters and restart the process as needed.

Both ASHD and general purpose approaches implement the same algorithms, so both are capable of reducing transmissions near the root mote up to 70%, thus reducing use of the mote's most power-hungry component. However, the ASHD implementation significantly reduces Bloom filter end-to-end delay: the maximum delay from a mote issuing a low battery alert to the server detecting the alert, is reduced from the general purpose implementation's 46s to 19s, assuming no transmission errors. Furthermore, the ASHD implementation reduces Bloom filter computation energy costs by 98% to 2.73mJ for every network-wide mote status scan.

## 6.2 Object Tracking

Previous work has used mote networks for object tracking [19]. Motes can identify objects by a unique ID using technologies such as RFID. In this application, we use Bloom filters with a sensor network to find packages in a busy package delivery warehouse. We assume each package has an RFID tag, so motes can detect package UIDs when nearby.

As packages move within the vicinity of a mote, the mote will wirelessly read the package's UID and store it in the mote's Bloom filter. When the Bloom filter becomes full, the mote will send the Bloom filter to the server. Note that the object tracking application does not merge Bloom filters with other motes. Instead, its Bloom filter is forwarded by other motes to the server for analysis. When a package is lost, the server looks at the most recently received Bloom filter for each mote and queries each to see if any have seen the package. If a false positive causes the package to appear in multiple places, previous Bloom filters can be examined to correctly identify the package location.

Note that this merge-free approach requires additional latency: Bloom filters only store data for one mote, so more time is required to fill the Bloom filter. However, this technique also ensures Bloom filters are sent when they are full and store items most efficiently. When latency is not critical, individualizing Bloom filters can improve transmission energy costs at every hop, not just near the root.

To build each Bloom filter, we must clear the Bloom filter, insert enough UIDs to fill the filter, and read the Bloom filter for transmission. With a false positive rate of 1%, the ASHD accelerator is able to reduce Bloom filter computation to 2.13s. This 85% reduction in delay over the general purpose logic design corresponds to a 97% reduction in computation energy consumption.

## 6.3 Duplicate Packet Removal

Bloom filters are well equipped for removing duplicate packets [9]. Wireless sensor networks are particularly susceptible to duplicate packets due to wireless transmission errors. By using Bloom filters to track whether a packet was previously received, these transmission errors can be filtered out. When a mote receives a packet, the mote creates a unique packet identifier from the source packet's UID and the packet's sequence number. We query the Bloom filter using this packet identifier. If the packet is found, we process the packet and send an acknowledgment to the source mote to indicate that the packet was received. We also insert the packet's identifier into the Bloom filter. If found, we likely received the packet and ignore it. However, we send an acknowledgment to the source mote indicating a duplicate packet because false positives may cause us to mistakenly ignore an original packet. However, the source mote will realize the mistake upon receiving the acknowledgment and resend the same packet with a new sequence number. Dropped packets will be detected when no acknowledgment is received by the sending mote. In this case, the packet will be resent with the same sequence number.

Although this application does not transmit Bloom filters, the accelerator improves storage ability and reduces search time. The Bloom filter accelerator stores more than 200% additional packet UIDs and provides extremely fast search times. Individually stored packet identifiers would require significantly more physical memory and additional searching algorithms.

When working with frequent radio transmissions, delays must be minimized. In the worst case, each delivered packet requires one item query and one item insertion to eliminate duplicate packets. For a 1% false positive rate, this process requires 240$\mu$s with the ASHD accelerator. This performance boost corresponds to an 88% delay reduction and 98% computation energy reduction over the general purpose implementation.

# Acknowledgments

The authors wish to thank Adam Kirsch for his invaluable help selecting algorithms. This work is supported by NSF grant CCF-0448313 (CAREER).

# Conclusion

We demonstrated the power, energy, and performance benefits of application-specific hardware design over general purpose design for wireless sensor networks and other embedded systems. Unlike desktop computers which must support an endless number of programs, wireless sensor networks require support for select few applications. Our mote status, object tracker, and duplicate packet removal applications demonstrate significant gains by designing hardware to fit application needs. Through hardware acceleration, WSN applications can aggregate information in-network, improve network reliability, reduce transmission latency, and efficiently store and search data.

# 7. REFERENCES

[1] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Allerton*, 2002.
[2] S. Chang, et al. Energy and storage reduction in data intensive wireless sensor network applications. TR-15-07, Harvard University, 2007.
[3] B. Chen, et al. Ad-hoc multicast routing on resource-limited sensor nodes. In *REALMAN '06*, pages 87–94. ACM, 2006.
[4] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO 36*, page 129. IEEE, 2003.
[5] Moteiv Corp. `http://bit.ly/I0cXD`, 2007.
[6] S. Dharmapurikar, et al. Fast and scalable pattern matching for content filtering. In *ANCS '05*, pages 183–192, 2005. ACM.
[7] M. Dietzfelbinger, et al. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.
[8] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.
[9] P. Hebden, et al. Bloom filters for data aggregation and discovery: a hierarchical clustering approach. In *ICIS '05*, pages 175–180, 2005.
[10] M. Hempstead, et al. An ultra low power system architecture for sensor network applications. In *ISCA '05*, pages 208–219. IEEE, 2005.
[11] J. Hill, et al. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.
[12] S. Madden, et al. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.
[13] J. Meany. Golomb coding notes. `http://bit.ly/3TCfI`, 2005.
[14] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.
[15] J.-K. Peir, et al. Bloom filtering cache misses for accurate data speculation and prefetching. In *ICS '02*, pages 189–198. ACM, 2002.
[16] A. Roth. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *ISCA '05*, pages 458–468. IEEE, 2005.
[17] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, second edition, 2000.
[18] L. Strozek and D. Brooks. Efficient architectures through application clustering and architectural heterogeneity. In *CASES '06*, pages 190–200. ACM, 2006.
[19] C. Taylor, et al. Simultaneous localization, calibration, and tracking in an ad hoc sensor network. In *IPSN '06*, pages 27–33. ACM, 2006.
[20] A. Wang, et al. *Sub-threshold Design for Ultra Low-Power Systems*. Springer, 2006.