ENERGY REDUCTION AND DELAY MINIMIZATION IN WIRELESS SENSOR NETWORKS THROUGH ASIPS

Manoj Kumar Jain

Mohanlal Sukhadia University, Udaipur, Rjasthan, India.
email: manoj@cse.iitd.ernet.in

ABSTRACT

Low energy consumption is a major design constraint for battery operated embedded systems such as wireless sensor networks or WSN. Low energy is more important compared to low power for such systems as it will increase lifetime of the system. The major component which can reduce energy is to reduce delay. WSN motes must power sensors, a processor, and a radio for wireless communication over long periods of time, and are therefore particularly sensitive to energy use. Recent techniques for reducing WSN energy consumption, such as aggregation, require additional computation to reduce the cost of sending data by minimizing radio data 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. Instead, application-specific instruction set processor (ASIP) 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. Both simple and complex operations can improve performance and use orders of magnitude less energy with ASIPs. This paper examines the design considerations of a ASIP for compressed Bloom filters, a data structure for efficiently storing set membership.

Index Terms—Wireless Sensor Networks, Application Specific Instruction Set Processor (ASIP), Low Energy, Delay Minimization.

I. 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 or hazardous locations. WSNs can consist of thousands of motes, 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 [1]. 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. Some simple operations, such as bit manipulations, poorly utilize a general purpose processor. Large multiplications and other complex operations may require several cycles on a general purpose processor. Many embedded applications require support for simple and complex operations. As a result, the system must use a power-hungry processor for simple operations or spend many cycles using a simple processor for complex operations.

Application specific instruction set processor (ASIP) tailors hardware to the application, efficiently executing simple and complex operations. We refer to these ASIP constructs as hardware accelerators. If any of these hardware accelerators are unused, they can be Vdd-gated so that they do not waste energy on unused features.

This paper explores ASIP considerations during the design of one such hardware accelerator. The accelerator implements several operations for Bloom filters, a data structure for efficiently storing set membership. These operations include support for

inserting items, compression and decompression, and querying.

We propose a scheduler based ASIP design methodology which is able to explore a large design space. It is validated for three different popular standard processors with significant architectural differences. Use of ASIPs reduces execution time and energy consumptions significantly. Results show that by changing number of registers by just one saves execution time by 57.5% whereas energy consumption is reduced by 62.9%.

This paper is organized as follows. Section II discusses related approaches to increasing energy-efficiency and motivates the ASIP paradigm. Section III describes the algorithms needed for the Bloom filter hardware accelerator. The proposed scheduler based ASIP design methodology ASSIST with its exploration and validation results are presented in Section IV. Section V discusses the architectural blocks needed to implement the approach. Finally, we discuss future work and conclude the paper.

II. RELATED WORK

A well-known paradigm for increasing energy-efficiency in general purpose computing is to utilize parallel processing. For example, in lieu of a single power-hungry core, designers can distribute computation across several low-power cores [2]. 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. 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 [3].
- 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 Instruction Set Processors (ASIP) A typical ASIP design flow includes key steps as application analysis, design space exploration, instruction set generation, code generation for software and hardware synthesis [4]. Design space exploration is driven by performance estimations. These estimates are generated using a simulator based [5] or scheduler based framework [6]. Simulator based technique needs a retargetable compiler to generate code for different processor configurations to be explored. Simulating the generated code is slow. Further, there is a well known trade off between retargetability and code quality in terms of performance and code size compared to hand optimized code. Therefore, in our opinion, simulation based approach is not suitable for early design space exploration.

III. BLOOM FILTER ALGORITHMS

Bloom filters provide a useful case study for an exploration of wireless sensor device ASIP. 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.

A. 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, $h_1 \ldots h_k$, Bloom filters reduce storage costs up to 70% [7]. 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 [8, 9, 10].

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, as illustrated in Figure 1, by hashing the item x_i with every hash function $h_1 \ldots h_k$. The results of these hash functions $h_1(x_i) \ldots h_k(x_i)$ 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 $h_1 \ldots h_k$ and check each bit's value at addresses $h_1(x_i) \ldots h_k(x_i)$. If any hash function points to a 0 bit, we know with certainty the item is not in the Bloom filter.

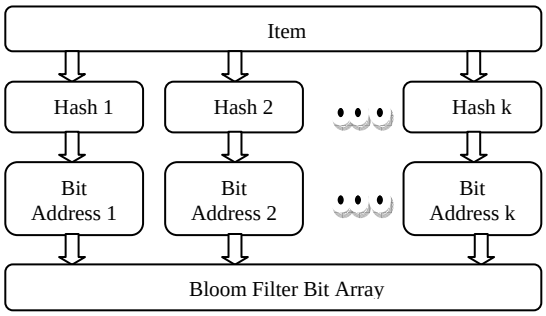

Figure 1: Inserting an item into a Bloom filter

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

Configuration	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%

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 [7].

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.

B. Multiply and Shift Hashing

Multiply and shift hashing, described by Dietzfelbinger et al. [11], is simple, yet effective. Each hash function $h_1 \ldots h_k$ requires a hash key $HashKey_1 \ldots HashKey_k$. 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

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

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.

C. 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 (JPEGLS) [12, 13]. As noted in Section III.A, 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 efficent.

1) Compression

First, the number of 1s in the bit array are 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 and quotient part q must be calculated:

$$r = \left\lfloor \frac{n}{2^l} \right\rfloor \quad (2)$$

$$q = n \bmod 2^l \quad (3)$$

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 counter's value using l bits. Therefore, Golomb-Rice compression can be reduced to many simple bit operations.

2) Decompression

Decompression is the inverse of compression. We read in one run length at a time from the compressed bit stream, knowing the quotient part ends at the first 1 and the quotient follows for the next r. The run length is calculated:

$$n = q \times 2^l + r \quad (4)$$

Once the run length n is calculated, n 0s are written to the uncompressed bit stream, followed by a 1. This process continues until the final run length is decompressed. Implementation is simpler: when reading the quotient part, 2^l bits are written to the uncompressed bit stream for every 0 in the compressed bit stream. When a 1 is read in the compressed bit stream, we switch to remainder part mode. In remainder part mode, we write 2^d 0s to the uncompressed bit stream for every 1 in the compressed bit stream, where d is the binary digit in the remainder. After the last, or 0^{th} digit, is reached in the compressed bit stream, a 1 is written to the compressed bit stream and we switch back to quotient part mode. Therefore, Golomb-Rice decompression consists of several simple bit operations.

IV. ASIP DESIGN METHODOLOGY

A. ASIP Design Methodology

Gloria et al [6] defined some main requirements of the design of application-specific architectures. Important among these are as follows:

1 Design starts with the application behavior.

2 Evaluate several architectural options.

3 Identify hardware functionalities for speed up

4 Introduce hardware resources for frequently used operations only if it can be supported during compilation.

ASIP fits in between these two and provides flexibility at lower cost than general programmable processors. According to MK Jain et al [4] design of ASIP can be typically divided in five steps which is shown in Figure 2:

a) Application Analysis
b) Architecture design space Exploration.
c) Instruction-set generation
d) Code synthesis
e) Hardware synthesis

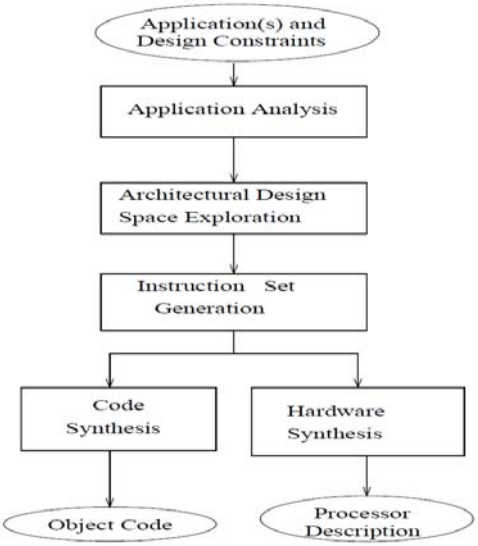

Figure 2: Flow Diagram of ASIP design Methodology

1) Application Analysis

ASIP design starts with analysis of application, analysis of test-data and design constraints. An application written in any high level language is analyzed both statically and dynamically which is then stored in some suitable intermediate format, which is then used in the subsequent steps.

2) Architecture Design Space Exploration

It involves identifying the broad architectural features of the ASIP. First of all, the architectural space to be explored is defined, keeping in view the parameters extracted during application analysis and the input constraints. Architecture is defined using some standard Architecture Definition Language (ADL).

3) *Instruction set generation*

Instruction set is to be generated for that particular application and for the architecture selected. This instruction set is used during the code synthesis and hardware synthesis steps.

4) *Code synthesis*

Compiler generator or retargetable code generator i used to synthesize code for the particular application or for set of application.

5) *Hardware synthesis*

In this step the hardware is synthesized using the ASIP architecture template and instruction set architecture starting from a description in VHDL/VERILOG using standard tools.

B. *ASSIST: A Scheduler based ASIP Design Methodology*

The overall flow diagram of ASSIST methodology is shown in figure 3. The inputs include application behavior in C, performance, power and area constraints, basic processor configuration, pipeline templates and memory access models, power models for various components, area and clock period models. The application is analyzed with the help of a profiler to extract application parameters. Design space exploration is an iterative process and it starts with a basic configuration (or minimal) that would be synthesized. The performance estimator estimates performance based on present processor and memory configuration, application parameters and input models. The configuration selector compares estimates to the user specified constraints to generate the next potential configuration. This process is iterated until a satisfactory configuration is generated which is used by a retargetable compiler generator in generating a customized compiler and by a VHDL synthesizer to generate a synthesizable VHDL code for the customized processor.

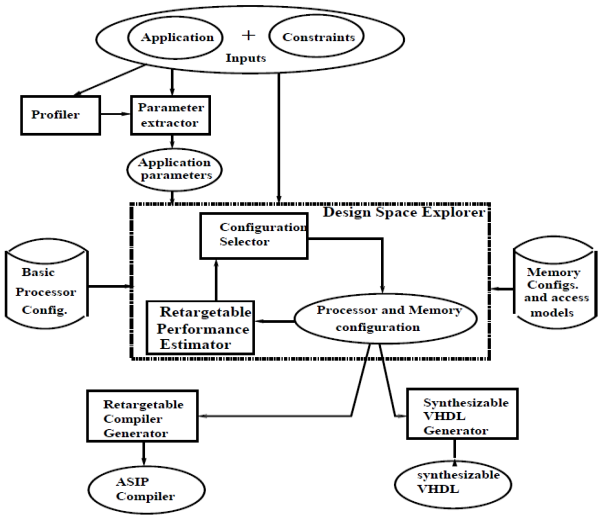

Figure 3: ASSIST (A Scheduler based ASIP Design Methodology)

C. *Integrated On-Chip Storage Exploration Technique*

Storage exploration is a part of the design space exploration phase of overall methodology. Proposed technique for storage space exploration is shown in figure 4. Cycle count for application execution on the chosen processor and memory configuration is estimated. A parameterized model for processor as well as memory is considered. Parameters of data cache include size, line size, associativity, replacement policy and access time. Processor configuration specification includes register file and windows organization along with pipeline information and functional unit (FU) operation capability and latency.

Register allocation is done on unscheduled code using reuse chains. We have defined cost of merging of reuse chains considering spills. We have also developed systematic way of merging these reuse chains. A priority based resource constrained list scheduler is used for performance estimation. Global register need estimation is done using variable usage analysis. Further, we estimate overheads due to limited register windows and data cache memory. We have integrated this technique to explore register file size, windows and cache configurations.

Overall execution time estimate (*ET*) for an application for the specified memory and processor configuration can be expressed as follows.

$$ET = et_R + oh_W + oh_C \quad (5)$$

Where

et_R : Execution time when register file contains R registers.

oh_W : Additional schedule overhead due to limited windows.

oh_C : Additional schedule overhead due to cache misses.

et_R can be further expressed by the following equation.

$$et_R = bet + oh_{dep} + spill_R * t_R \quad (6)$$

Where

bet : Base execution time considering constraints of resources

other than storage.

oh_{dep} : The overhead due to additional dependencies inserted during register allocation.

$spill_R$: The number of register spills.

t_R : The delay associated with each register spill.

Computation of et_R is described in the next Section. oh_W can be

further expressed by the following equation.

$$oh_W = spill_w * t_W \quad (7)$$

Where

$spill_w$: Number of window spills and

t_W : Delay associated with each register window spill.

oh_C can be further expressed by the following equation.

$$oh_C = miss_C * t_C \quad (8)$$

Where

$miss_C$: Number of cache misses and

t_C : The cache miss penalty.

t_W is computed by knowing register window size and the latency of 'store' instruction. t_C is computed using block size and the delays associated in each data transfer. Storage configuration selector selects suitable processor and memory configuration to meet the desired performance by knowing all the execution time estimates.

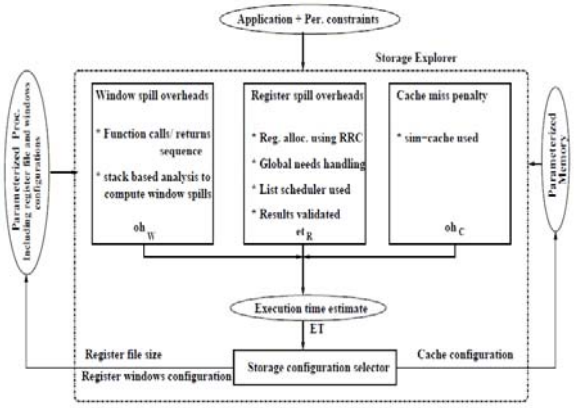

Figure 4: Integrated On-Chip Storage Exploration Technique

D. Exploration Results

1) Trade-off between Number of Register Windows and Window Size

We are interested in trade-off between the number of registers and window sizes. For each total number of registers, window size would be different for different number of windows. While generating results (figure 5), we assumed that register file will be distributed in windows of equal sizes. We also assume that within a context, number of registers available for register allocation is equal to window size. Depending on the performance requirement, suitable register file size can be chosen and for the chosen register file size, number of windows and hence the window size (number of registers in a window) can be decided. On one end, when the number of windows is small, the time overhead due to context switches dominates the cycle count. At the other extreme, when the number of windows is large for the same total number of registers, the individual window size becomes small and the overhead due to load and stores dominates the cycle count.

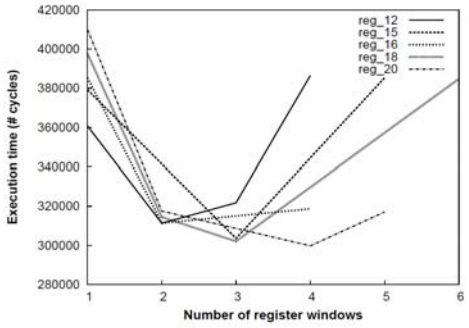

Figure 5: Trade-off between number of windows and their sizes

2) Trade-off between Register File Size and on-chip Data Cache

Execution time estimates for various benchmark applications for different register file sizes and different data cache sizes were generated. We have not considered the impact of cache size variation on memory latency, but it can be considered by choosing appropriate values of a1 and a2. Consider the results produced for *matrixmult* program for different register file size and memory configurations (figure 6). Some interesting trade-offs can be observed. Based on the generated execution time estimates and the input performance constraint, suitable configurations can be suggested.

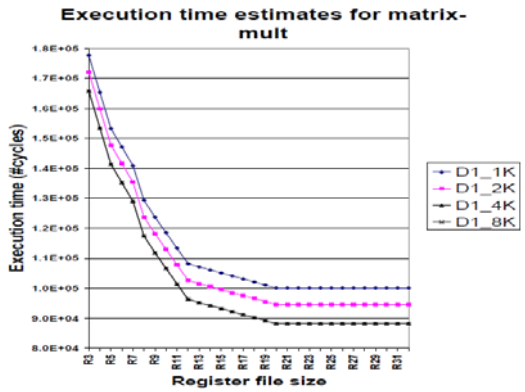

Figure 6: Results for matrix-mult

3) Execution Time Validation

Performance estimations with varying on-chip storage configurations for selected benchmarks applications were done. Three processors namely ARM (*ARM7TDMI* a RISC) [14], Trimedia (*TM-1000* a VLIW) [15] and *LEON* (a processor with register windows) [16] were chosen for experimentation and validation. *TM-1000*'s five-issue-slot instruction length enables up to five operations to be scheduled in parallel into a single VLIW instruction. To know correctness of our techniques, we chose to validate our result against the numbers produced by standard tool sets. Validation shows that our estimates are within 10% compared to the actual performance numbers produced by standard tool sets. The actual figures were 9.6%, 3.3% and 9.7% for *ARM7TDMI*, *TM- 1000* and *LEON* respectively. Further, this technique was nearly 77 times faster compared to the simulator based technique. Results generated were also validated against VHDL level simulation for collision detection application. The execution time estimates produced by our estimator (443278 cycles) are within 10.33% compared to the estimates produced by tsim (494375 cycles) and within 5.26% compared to the estimates produced by VHDL simulation

V. ASIP ARCHITECTURE

This paper leverages the mote architecture described by Hempstead, et al. [17] 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 [17] and support a 16-bit bus. The accelerator consists of several modules, illustrated in Figure 7. In the following sections, we will examine each major module in the Bloom filter accelerator and discuss design decisions for reducing energy and delay.

A. Instruction Decoder

The Instruction Decoder is the command center for the Bloom filter accelerator. In contrast to general purpose processors, the accelerator's instruction decoder is simple because it only handles a small set of 32 Bloom filter instructions. The Instruction Decoder receives control signals from the event processor to determine the current operation, and sends control signals to other modules in the accelerator (shaded in Figure 7). As the cycle completes, the Instruction Decoder monitors progress and notifies the Event Processor using acknowledgment signals and interrupts when appropriate.

B. Data Builder

As noted earlier, all hardware accelerators must support a 16-bit bus. However, several Bloom filter operations require 32-bit hash keys and 32-bit items. The Data Builder is used to combine 16-bit segments from the data bus over two cycles into a 32-bit integer. If future designs require larger items, the Data Builder can be easily modified to combine segments over more cycles. We chose to design the data builder as a distinct module for reuse in future accelerators.

C. Hash Unit

The Hash Unit is responsible for managing hash keys and performing multiply and shift hashing. When initially powered, the mote's code must save the correct hash keys. These hash keys are rarely changed and could be statically programmed. However, regularly changing hash keys could deter snooping on gathered data. Once the hash keys are saved, the Hash Unit uses multiply and shift hashing to generate bit addresses for item insertion and querying. As noted in Section III.B, we simply take the 31st through 15th bits from the hash key-item multiplication, and do not require a full 32-bit multiplication. We implemented this complicated operation in hardware so that the calculation could be performed within one cycle. These hash operations are extremely fast in hardware and we are able to use the hash result to access memory in the same cycle.

D. Hash Key Memory

The Hash Unit uses the Hash Key Memory to store hash keys. The memory stores up to 16 hash functions. As only 14 hash functions are required for a false positive rate below 0.01%, no need exists for more hash

key capacity. We separated the Hash Key Memory from the Hash Unit for applications which never change hash keys. In this case, the Hash Key Memory could be replaced by a lower power ROM.

E. Counter

The counter is used when iterating through hash keys for item insertion or querying, and for several operations iterating through the entire Bloom filter memory. Because these operations require several cycles, the counter is used to remember the next hash key or address in Bloom filter memory at the next cycle. We chose to create a distinct counter module because it is used by the Memory Access Controller and Hash Unit, although never simultaneously. By sharing the counter, we reduce power consumption by eliminating counting-related memory by half.

F. 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.

G. 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 III.B.I. As previously noted, memory access can be a bottleneck, so this optimization is critical for performance.

H. 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.

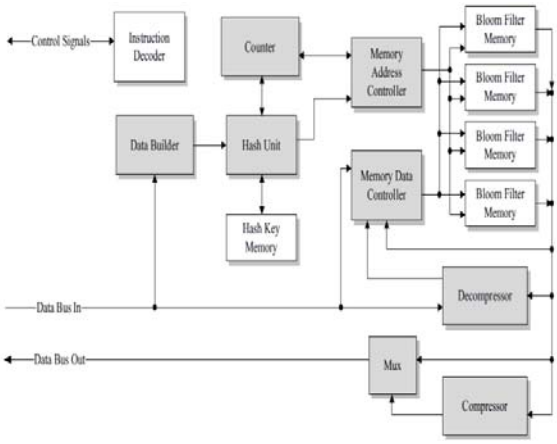

Figure 7: Bloom filter hardware accelerator hardware flow: arrows indicate the direction of information, shaded blocks indicate modules controlled by the Instruction Decoder.

I. 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 limits are solely due to the 16-bit data bus and 64-bits of Bloom filter memory accessible during a given cycle. Although these limits require significant additional logic, we decided to support this higher performance design to avoid elevated computation times when processing Bloom filters containing many elements.

The Decompressor is composed of 16 serially-connected bit decompressors. This design allows each compressed bit to be decompressed serially, as described in the implementation portion of Section III.C.2. 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.

J. Compressor

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. As noted in Section V.I, supporting these guarantees requires additional logic, but gains in performance make this addition worthwhile.

The Compressor design, is composed of 64 serially connected single-bit compressors. Each single-bit compressor performs the implementation discussed in Section III.C.1, adding a single bit to the compressed bit stream as needed. Although the compressors could execute in parallel and reassemble the compressed bit stream, serial execution allows later bit compressors to be disabled if no room is available in the compressed bit stream. Due to the slow operating frequency, parallel processing is not required.

VI. CONCLUSION

In this paper we have proposed an ASIP solution for a low energy and performance efficient wireless sensor networks. Proposed ASIP approach is validated for three versatile standard processors and results are encouraging. Since proposed approach uses scheduler based approach for performance estimation, it is significantly better than traditional time consuming simulator based approaches. In addition to that our approach is able to handle larger design space compared to simulator based approaches. We plan to implement at prototype level for an efficient wireless sensor network in future.

REFERENCES

[1] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[2] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[3] A. Wang, B. H. Calhoun, and A. P. Chandrakasan. *Sub-threshold Design for Ultra Low-Power Systems*. Springer, 2006.

[4] M.K. Jain, M. Balakrishnan, and A. Kumar. ASIP Design Methodologies: Survey and Issues. In *Proceedings of the IEEE / ACM International Conference on VLSI Design. (VLSI 2001)*, pages 76–81, January 2001.

[5] A. D. Gloria and P. Faraboschi. An Evaluation System for Application Specific Architectures. In *Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture. (Micro 23)*, pages 80–89, November 1990.

[6] N. Ghazal, R. Newton, and J. Rabaey. Retargetable Estimation Scheme for DSP Architecture Selection. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 485–489, January 2000.

[7] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Allerton*, 2002.

[8] A. Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *ISCA '05*, pages 458–468. IEEE Computer Society, 2005.

[9] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data and prefetching. In *ICS '02*, pages 189–198. ACM Press, 2002.

[10] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for content filtering. In *ANCS '05: Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pages 183–192, New York, NY, USA, 2005. ACM.

[11] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

[12] J. Meany. Golomb coding notes. http://ese.wustl.edu/class/fl06/ese578/GolombCodingNotes.pdf, 2005.

[13] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, second edition, 2000.

[14] ARM Ltd. Homepage. "http://www.arm.com".

[15] Trimedia Homepage. "http://www.trimedia.com".

[16] LEON Homepage. "http://www.gaisler.com/leon.html".

[17] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and . D. Brooks. An ultra low power system architecture for sensor network applications. In *ISCA '05*, pages 208–219. IEEE Computer Society, 2005.