WISENEP: A Network Processor for Wireless Sensor Networks

André Mota[1], Leonardo B. Oliveira[2], Geórgia P. Safe[1], Felipe F. Rocha[1], Ramon Riserio[1], Antonio A. F. Loureiro[1], Claudionor J.N. Coelho Jr.[1], Hao Chi Wong[1] and Eduardo Nakamura[1]

[1] Federal University of Minas Gerais (UFMG), Brazil

[2] Supported by FAPESP grant 2005/00557-9, University of Campinas (UNICAMP), Brazil

Email:{andmota,georgiap,felipefr,rrdl,loureiro,coelho,hcwong}@dcc.ufmg.br, leob@ic.unicamp.br

***Abstract*— Wireless sensor networks are ad hoc networks comprised mainly of small sensor nodes with limited resources and one or more base stations, which are much more powerful laptop-class nodes that connect the sensor nodes to the rest of the world. The advent of this tecnology over the last decade enables large-scale deployment of such sensors. On the other hand, this poses the challenging of how the great amount of information generated by these sensor networks will be handled at the base station. In this paper, we propose a network processor architecture tailored specifically to handling information at sensor network base stations. Our approach optimize information processing by implementing tasks in hardware. We show that the novel architecture is one order of magnitude faster than an architecture based on traditional RISC processor.**

I. Introduction

Wireless sensor networks (WSNs) consist of a large number of distributed communicating resource-constrained devices deployed to accomplishing monitoring and control goal [1], [2], [3]. These networks have the potential to be applied in a variety of areas such as industry, agriculture, environmental control, scientific, and consumer systems. An important point in applications foreseen for WSNs is the availability of relevant data for monitoring system connected to these networks.

When compared with other types of networks, WSNs present some distinguishing characteristics in terms of scale, communication pattern, resource level and mobility. E.g, they are typically orders of magnitude larger [4], [5] than other netwoks and the network traffic flow is asymmetric – mainly from sensor nodes to base stations. As a consequence, base stations are responsible for handling a large amount of data. Based on recent advances in WSN tecnology, we argue that, in near future, base stations will must have built-in computational abilities to be capable of dealing with the collected data in large-scale WSNs.

In this work we present WISENEP, a **WI**reless **SE**nsor **NE**twork **P**rocessor, whose architecture is tailored specifically to handling information at base stations. WISENEP is intended to replace conventional processors in base stations and it can bring new levels of performance in information processing. Even though much work has been done in the context of information processing in WSNs , to our knowlege, this is the first attempt at optmizing this task at the base station. Results show that the novel architecture is at least 10 times faster than an architecture based on traditional RISC processor.

This paper is organized as follows. Section II presents some of the related work. Section III describes the architecture of the network processor proposed in this work. Section IV presents evaluation analytically and by means of simulation. Finally, Section V draws some conclusions and discusses future directions.

II. Related Work

WSNs are used for monitoring purposes, providing information about the area being monitored to the rest of the system and thus most of the work are related, in some way, to information processing.

Among the studies specifically targeted to processing information, Chu *et al.* [6] introduce a general information architecture for designing distributed inference algorithms in WSNs. The architecture comprises a graphical information representation with processing mechanisms guided by sensor evidence and provides a global view of the set of computations occurring in the system

In [7], Ganesan *et al.* describe DIMENSIONS, a system that provides a unified view of data handling in sensor networks. DIMENSIONS incorporates long-term storage, multi-resolution data access and exploits spatio-temporal correlations in sensor data.

To the best of our knowledge, our solution is the first attempt to design a specific hardware architecture to optimize information processing at base stations.

III. Proposed Architecture

The processor proposed in this work is supposed to be the main processing unit of a WSN base station. It continuously receives data packets from the network and processes their content. The result can be either sent to the WSN client or used to manage the network nodes. We imagine the sensed area mapped into a grid, where each grid square represents the measured value of that respective region and the entire grid give us an 'image' of the sensed area. In this case, the base station should perform the following tasks: maintain bi-dimensional matrices representing network measured variables - one matrix for each variable; decode the bit stream received from the network interface into the proper packet's fields; update the matrices on each received packet data; and perform computations on these matrices and define proper actions.

WISENEP is a processor architecture designed to perform these functions in a very efficient way.

WISENEP design can be better understood if seen as three main blocks. Figure 1 helps visualizing them. They are:

1) A Packet Classifier, responsible for decoding the bit stream received from the network. This packet classifier is no different from packet classifiers proposed for common network processors. It receives a bit stream from the network interface on its input ports and provides the decoded packet fields on the output ports.
2) The Packet Processor, a parallel processor designed specifically to compute WSN data. It maintains the bi-dimensional matrices, each matrix representing one measured variable grid view. It has hardware blocks to determine the location on the matrix of a WSN node and to store the received data on that location. It also has hardware blocks to process these matrices data efficiently (e.g., interpolation and edge detection algorithms). Due to its parallel design, the processing of all different measurements on the same packet can be done at the same time - for example, if a packet has measurements about wind speed, temperature and relative humidity the storage of each one on the respective matrices can be done in parallel.
3) A Host Processor, a general RISC processor responsible for common processing tasks. It has a direct connection with the Packet Processor memory. It is intended to take decisions based on the summarized data provided by the Packet Processor, to send commands to the Packet Processor asking for specific tasks, to send commands to the WSN nodes (like shutting down some nodes to preserve network energy) and to handle commands from the WSN client.

The packet classifier handles only bit streams and does that very fast. The Packet Processor handles WSN data very efficiently, since it is hardware optimized for that task, but the tradeoff is that it will not handle general task as efficiently as a state of the art RISC processor. So, the Host Processor complements the first two blocks, allowing them to be very specific and specialized but still letting our architecture handle general tasks well.

From now on, we will detail the architecture of the Packet Processor, since the architecture of a Packet Classifier and a RISC Host Processor are well know in the literature.

Figure 2 shows the Packet Processor architecture. There are two main elements to be understood: Microengines and ID Lookup.

- A microengine is a specific purpose processor with an instruction set and architecture designed for WSN data processing. Microengines are responsible for storing the packet's data on matrices structures and for performing computations on theses matrices. Each one has one independent memory to store its data. There can be one or many of them on our architecture; using more than one provides parallelism on data handling. Each should run a different code, allowing them to handle different measurement data of the same packet in parallel. On our examples and illustrations we are assuming a project with 3 microengines.
- The ID Lookup is simply a hardware lookup table responsible for converting the node's ID information into X and Y coordinates.

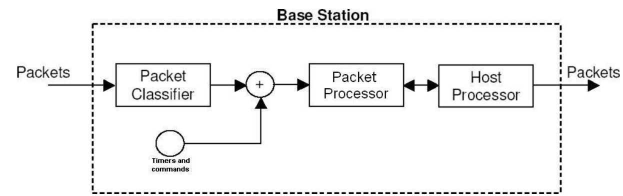


Fig. 1. Architecture Overview

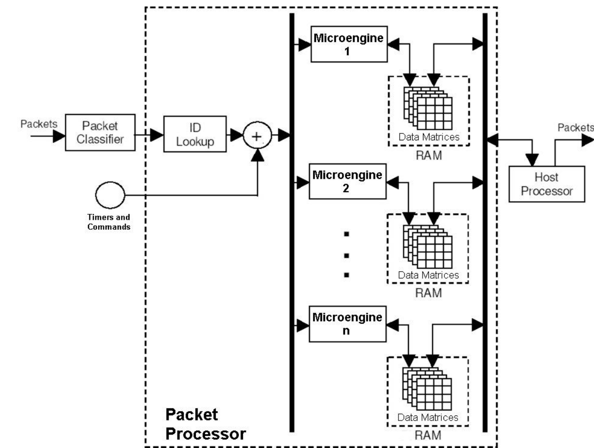


Fig. 2. Packet Processor Architecture

The principles of our archictecture will be made clear next, when we explain the flow of information inside WISENEP. The reader can follow this better by looking at figure 2. Suppose that a packet from network node 1 containing three different measurements (e.g., wind speed, temperature and relative humidity) arrives at the base station. First, the Packet Classifier will decompose the bit stream in four different fields (node ID and the three measurements). Then, these four fields will go into the Packet Processor, entering first the ID Lookup block. The ID Lookup will lookup node 1 position and will provide the four original fields plus the node's X and Y position. Then, this information will be passed to the three microengines. The code on each microengine instructs it to store one of the measurements (e.g., microengine 1 handles wind speed, microengine 2 handles temperature and microengine 3 handles relative humidity). So, as packets are received, the Packet Processor will maintain a grid view of the three measurements on three different matrices in real-time. Occasionally, the Host Processor may want those informations summarized. For instance it may want to find the regions on the grid view where there is high contrast. So, it sends a specific request to the Packet Processor. This request may be an Edge Detection command, for example. Each microengine will then perform edge detection on its matrix. Finally, the Host Processor can read the resulting matrices representing the edges and take apropriate action. Note that the Packet

Processor handles events. An event may be a Host Processor command, a Timer Event generated by an external clock or a network packet.

We will now detail the microengine internals. This explanation can be followed referring to figure 3. A microengine has a ROM (or FLASH memory) with specific code to handle each kind of event (network packet reception, clock timer events, host processor commands). Each microengine is also attached to an independent RAM memory. The RAM is used to store the grid view of the network measurements and to process them. A simple circuit between the microengine and the RAM allows the first to access the memory as a bi-dimensional matrix automatically. On the input of each microengine there is an Event Queue. It serves as a buffer of the received events, allowing the event incoming rate to became superior than the event processing rate for a short period of time. Note that, on the long run, the overall event incoming rate must be equal or less than the processing rate, or else the buffer will run out of space. The Event Queue assigns a different number to each kind of event so that, when the Event Processor handles each event, its respective number specifies which ROM code will be executed.

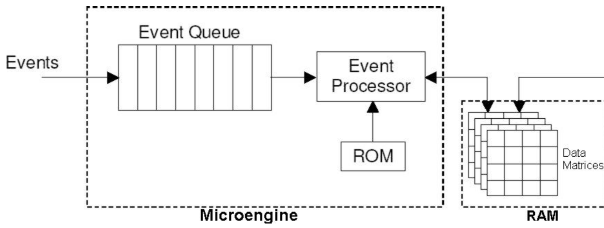


Fig. 3. Microengine Details

The Event Processor is the core of a microengine, so we will explain it more fully. The Event Processor will continuously dequeue an event, execute the associated code and look for the next event. Its datapath is specifically optimized to handle measurements coming from the network[1] and store them on the grid view. It also has hardware blocks implementing matrices operations. Note that, once the grid view provides an 'image' of the network measurements, it is interesting to use image processing algorithms on it. So, we envisioned hardware execution blocks performing Interpolation, Edge Detection and Thickening algorithms on the grid view matrices. We opted to implement those functions in hardware because hardware implementation of those algorithms can be one order of magnitude more efficient that software implementation. The Event Processor fetches from ROM the instructions specifying what it must do. The programmer of the microengine has a set of data processing instructions to use to construct an event handling logic. All memory instructions[2] specifies a memory location using X and Y coordinates, instead of a one-dimensional continuous addressing. There are also specific instructions to do image processing and there are general purpose instructions to do arithmetic, logical and branch operations. To specify the end of the event handling, there is a Next intruction, to instruct the Event Processor to unqueue the next event. The event number of the next event fetched from the queue will specify which code on ROM will handle the event data - after a Next instruction, the Program Counter of the Event Processor automatically points to the proper ROM address. Figure 4 shows the datapath designed to perform all those functions.

[1]temporally stored on the Event Queue

[2]Load and Store instructions

IV. Performance Evaluation

Performance evaluation of WISENEP was conducted according to two methodologies: analytical model study and processor simulation. In both we compare the performance of WISENEP against a RISC processor, since a base station would be equiped with a RISC processor. We believe that using two different approaches to evaluate performance gains, both getting to similar conclusions, strongly validates our results. In that way, the two methodologies are complementary.

In the following, we first present the analytical model study, its methodology, benchmarks and results. Then we present the WISENEP functional simulation, its benchmarks and results.

A. Analytical Model

1) Methodology: Our goal is to quantitatively compare the relative execution time that each architecture takes to perform the WSN network management operations. To achieve this goal, we have defined a set of WSN applications benchmarks and created execution models for these benchmarks. Relying upon these execution models, we counted the number of memory and ALU operations [3] needed in each architecture and used these results as a proxy to the total execution time.

Our execution models considers each basic operation taking a time unit. An basic operation is a memory access or a ALU operation - because they are the critical paths on a regular processor. A execution model is a directed graph, where the nodes represent basic operations and the edges show the processing flow. Sequential operations are represented as a sequence of nodes and parallel operations are represented with many paths starting from a node. The longest path from the beginning of the graph to the latest node denotes the number of basic operations executed sequentially, and, therefore, the number of time units required. Figure 5 illustrates these concepts.

One basic assumption about the RISC performance model is that the instruction scheduling is optimal and instruction cache hit rate is 100%. This implies that the processor is capable of dispatching one instruction per cycle. As far as the proposed applications are concerned, the current state of the art in RISC technology [8], [9], [10] allows current processors to approximate these assumptions. Operating System overhead is not considered in our formulation. Nevertheless, our model establishes an upper limit to the performance of the RISC architecture.

2) Benchmarks: We designed a set of a typical tasks that together define a WSN application. We are assuming a application similar to the one described in the previous section. Table I shows the tasks needed to provide that application.

[3]arithmetic and logic operations

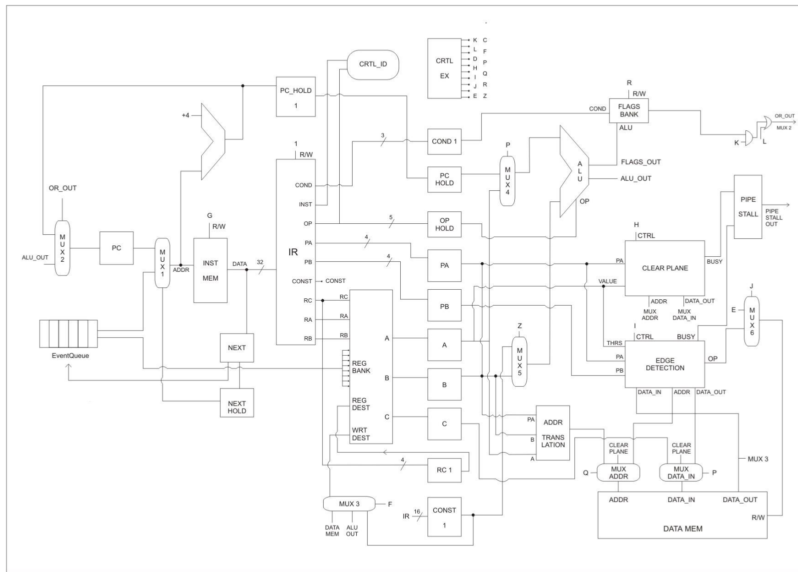


Fig. 4. Datapath design

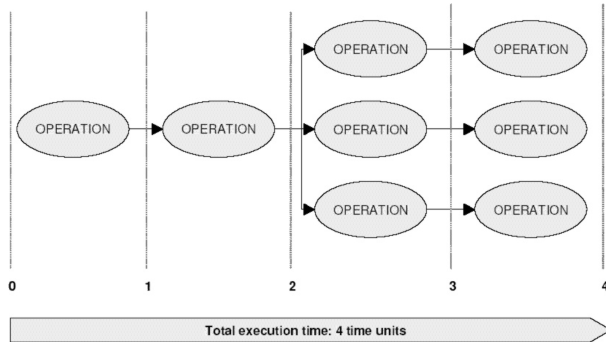


Fig. 5. Example of an execution model graph

Task	Description
Data Packet Processing	Receives packet and writes node's measurements to each plane
Grid Processing	
Conditional Interpolation	Interpolation (3×3 convolution) of unmeasured grid regions
Edge Detection	Sobel Edge Detection (3×3 convolution)
Binary Thickening	Threshold followed by thickening (3×3 convolution)

TABLE I

DESCRIPTION OF TASKS USED FOR BENCHMARKING

Incoming data packet processing is the simplest and the most common operation. At any given time, data packets may arrive at the base station. Grid processing tasks, however, are performed from time to time, in order to summarize the collected data. The Host Processor uses the summarized data to take appropriate actions.

3) Execution Models:

a) Data Packet Processing:: There are three basic steps any implementation must perform on incoming packets. First, the packet bit stream must be analyzed and decoded. Then, the processor needs to lookup the geographic position of any given node. Usually, it has an internal lookup table associating each node ID with their known (X,Y) coordinates. Finally, all data in the packet must be written on the apropriate matrix.

Note that we focused our analysis on packet content processing, not network communication processing. The reason is that solutions for pure network processing are well known [11]. We are assuming that network communication processing does not became a bottleneck.

We consider here that each packet contains the node ID plus an arbitrary number of measurements (for example, a packet could contain node ID plus measurements about wind speed, temperature and relative humidity - summing three different kinds of measurements). From now on, we well refer to the number of different kinds of measurements simply as n.

Figure 6 shows the basic units of execution of the WISENEP architecture. In the proposed processor, each matrix store operation (MEM_WRITE cycle) can occur in parallel, up to the limit of the number of microengines avaliable. Also, execution of the three necessary steps for processing incoming packets can be overlapped - while one packet content is begin processed on MEM_WRITE cycle, the next one when is already on the MEM_READ cycle. Thus, WISENET can deliver a rate of one new packet per unit of time, assuming that there is at least one engine for each plane. If there are not enough microengines (e.g., the incoming packet may contain five types of measured information but the processor has only three microengines), the number of units of time to process each packet is the ratio between the number of planes and the number of engines, rounded up to the nearest integer.

In Figure 7, we can see that the same tasks are all performed sequentially on RISC processors. A RISC processor needs to read each packet field, translate node ID to X and Y coordinates and finally write each kind of measurement on its matrix. So, a it would take $2n + 4$ unit of time to execute this operation.

Note that ID Lookup phase takes 3 memory reads in a RISC processor, instead of just one as in WISENEP. This is because WISENEP ID Lookup hardware is especially designed for this

task, so 3 independent memory banks can be used. On the other hand, a general propose RISC processor can access only one memory position at a time.

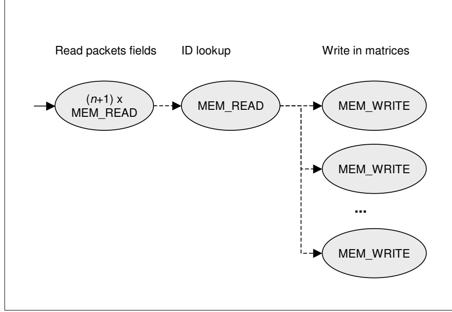


Fig. 6. Execution model for receiving data packets on WISENEP

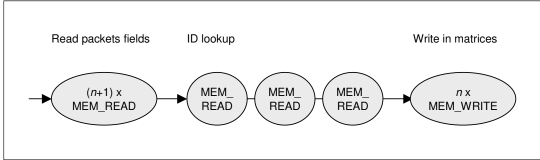


Fig. 7. Execution model for receiving data packets on RISC

Since RISC handles one packet in $2n+4$ units of time and WISENEP handles an average of one packet per unit of time, we can see that WISENEP is in general $2n+4$ times faster than a RISC processor in Data Packet Processing. For example, assuming a packet with 3 different kinds of measurements, we could have a speedup factor of 10 on the most common operation.

b) Grid Processing:: Figures 8 and 9 shows the execution model for edge detection on both architectures. They represent the necessary steps to perform the Sobel Edge Detection algorithm on each architecture. The x and y on the Figures, as well on the equations ahead, denotes the horizontal and vertical size of the grid view, in terms of number of grid points. The lozenges on these diagrams represent a loop of x, y and n steps, respectively. We will not explain how the execution model graphs were derived from Sobel algorithm in order to not make the argument too long. Readers who wants more information about Sobel algorithm can refer to [12].

Analyzing those execution models, we can see that a RISC processor must work on each grid view sequentially, whereas WISENEP will process each grid simultaneously using its parallel microengines. There is also another benefit of using a customized architecture for grid processing: faster matrix operations. Processing a grid view usually consists of many independent operations on a $m \times m$ matrix. Sobel edge detection, in this particular case, performs 7 different multiplications on 3×3 matrices. Especially designed hardware can calculate these 7 multiplications in parallel[4].

[4]These are simple multiplications (factors of 0, 1, 2, -1 and -2) that can be done with simple combinational logic. We are not proposing 7 full multiplicators in parallel.

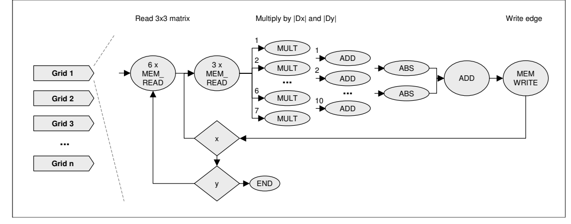


Fig. 8. Execution model for edge detection on WISENEP

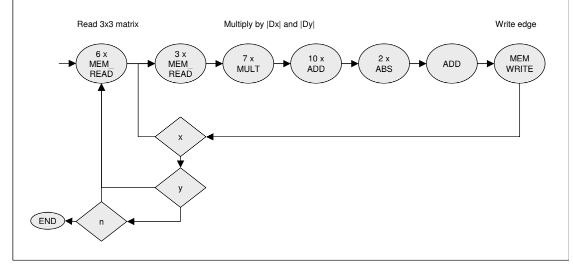


Fig. 9. Execution model for edge detection on RISC

Considering that all grid views can be processed in parallel, the time needed to process Sobel edge detection on WISENEP is (in units of time):

$$y(6+10x)$$

In a RISC processor, on the other hand, the same computation is performed in:

$$n \times y(6+24x)$$

Since x should be much greater than 10 in a normal grid view, $10x$ would be much greater than 6 (so $\approx 24x$). Therefore, the speedup can be summarized as:

$$\frac{n \times y(6+24x)}{y(6+10x)} \approx \frac{n \times 24xy}{10xy} = n \times 2.4$$

For example, assuming three different grid views, the performance gain of WISENEP would be 7 times.

Analytical modeling of other grid operations is done similarly. Table II shows the results for each grid operation.

Grid Operation	**Execution time in WISENEP**	**Execution time in RISC**	**Speedup (approximately)**
Conditional Interpolation	$y(6+5x)$	$n \times y(6+30x)$	$n \times 6$
Edge Detection	$y(6+10x)$	$n \times y(6+24x)$	$n \times 2.4$
Binary Thickening	$y(6+6x)$	$n \times y(6+11x)$	$n \times 1.8$

TABLE II
EXECUTION TIME AND SPEEDUP FOR EACH GRID OPERATION

4) Analysis: In comparison with a RISC processor, WISENEP speedups data packet processing, the most common task, by a factor of 10. Grid processing, a not so common task but very time-consuming, is speeded up about 9 times. Grid processing speedup is the combined speedup of Edge

Detection, Binary Thickening and Conditional Interpolation operations. Figure 10 shows the execution time for each benchmark on both architectures. To calculate these numbers, we are considering $n = 3$ (three different kinds of measurements). In this particular case, these results mean that for data packet processing and grid processing, WISENEP is about 9 to 10 times more efficient than a RISC processor.

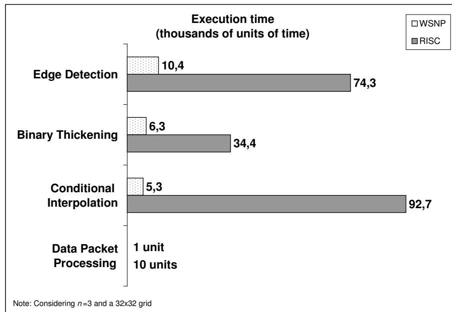


Fig. 10. Benchmarks' execution time for each processor

B. Processor Simulation

In order to simulate WISENEP architecture, a functional simulator was implemented in C++ language. A functional simulator divides the processor into blocks that will exist in a real implementation. Therefore, the simulation must produce results similar to results of the program running on the actual processor. The total number of clocks necessary to execute the benchmarks is a natural output of a functional simulator.

WISENEP's performance was compared against the performance of a RISC processor. The RISC application was implemented in the C language and executed in a SPARC ULTRA-5 (360MHz) machine running in a single user mode, i.e., only a small set of essential kernel processes are left running. This application was instrumented in order to log the amount of CPU time consumed by the application what, in turn, was used for estimating the number of clock cycles elapsed.

1) WISENEP Simulation Methodology: In order to construct an accurate simulator, we designed a entire processor datapath and control logic. To make sure the feasibility of a real world implementation, this design is almost at hardware implementation level.

The simulator was built using a hardware simulation framework that provided simulation core functions, like basic hardware blocks, clock generation, block interconnection features and wire signal propagation. The framework has a library of pre-defined hardware blocks, such as ALU, Registers, Mux's, Logical Gates, RAM and ROM Memory and others. More complex blocks - for example, edge detection logic - were constructed using Finite State Machines in conjunction with simple combinational logic. In this manner, this framework gives a low level idea as how it would be a real hardware implementation of our design.

Each of these blocks has a set of inputs and outputs. To construct the WISENEP simulator, we connected the inputs and outputs of the blocks according to our datapath design. The final result is a cycle accurate hardware simulator that not only shows the feasibility of a real world implementation but gives us precise performance results in terms of clock cycles, as well.

Note, however, that the framework does not simulate gate propagation delay and in turn we cannot determine the minimum clock period. On the other hand, the critical path of our design is no longer than a typical RISC so, considering the same underlining technology, this allows us to assume that clock frequencies for WISENEP and RISC processors will be very similar. Under these circumstances, the number of clock cycles needed for carrying out a task is a very good proxy for comparing WISENEP and RISC processors performance.

2) Benchmarks: An application has been envisioned in order to be implemented on both architectures (RISC and WISENEP). Imagine a large area mapped into a 32×32 grid where each sensor corresponds to a grid point and they are collecting four data types (e.g., temperature, pressure, humidity and luminosity).

The tasks were chosen to be part of the benchmark: data packet processing and edge detection grid operation. Data packets containing the four values measured by each sensor will arrive uninterruptedly and the simulators will just store these values in four different matrices. Besides that, from time to time, a timer event will arrive, so that the edge detection function will be started for each one of the four grid views. A new matrix containing the result of the edge detection function will be generated for each kind of data.

In order to study the performance gains of the proposed network processor, we varied the workload in terms of two parameters: the amount of event timers and number of data types being monitored (i.e., which arrive into the packet, simultaneously).

3) Simulation Results: The initial results will be presented by means of graph curves representing the number of clock ticks elapsed, where we vary the number of different measurements. Each graph contains two curves: one for RISC data and another one for WISENEP data.

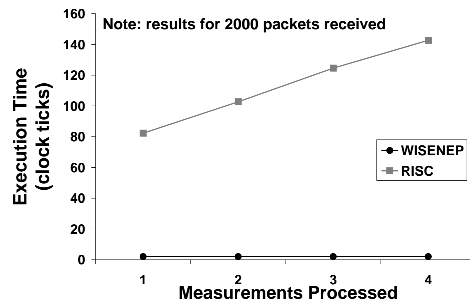


Fig. 11. Execution time comparison varying number of sensed variables, considering only data packets

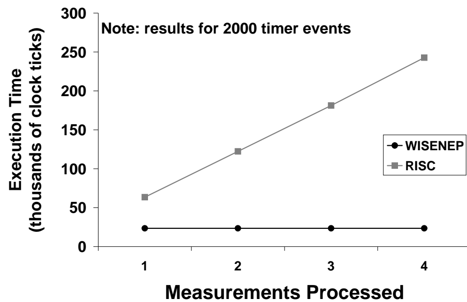


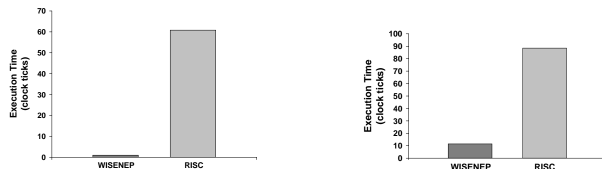
Fig. 12. Execution time comparison varying number of sensed variables, considering only timer events

Note that not only WISENEP is more efficient, but also it scales more gracefully. When we varied the workload in terms of number of different kind of measurements, WISENEP performs better than RISC while receiving both timer event and data packets —Figures 12 and IV-B.3, respectively.

In addition, while RISC architecture presents a linear growth when increasing the number of sensed variables, WISENEP maintains the number of *clock ticks* constant. As mentioned in Section IV-A.3, this is so because RISC processes each measure sequentially. WISENEP, on the other hand, is able to distribute workload among its microengines.

In Figure 13(a), it is presented the execution time for processing one data packet with 3 different kinds of measurements. Note that WISENEP is 60 times more efficient than RISC, approximately. In the same way, Figure 13(b) shows that WISENEP runs Sobel edge detection 8 times faster than RISC. Note that some of this can be attributed to the overhead generated by the execution environment. Nevertheless, this value gives an idea of how much WISENEP is efficient.

The simulation results presented here are coherent with the analytical results. Both showed that performance could be improved by one order of magnitude by using the proposed architecture, especially in large scale networks.



(a) Execution time for processing one data packet with 3 sensed variables

(b) Execution time for processing edge detection for 3 planes

Fig. 13. Efficiency comparison between WISENEP and RISC architectures

V. CONCLUSIONS

In this paper we presented a novel architecture for a WSN base station processor designed to be able to handle large amounts of sensor data very fast. We can summarize the main designs decisions in three points: First, the processor datapath was optimized to handle the most common task - data packet processing. Second, the use of explicit parallelism, by means of multiple microengines and independent RAM memories, allows simultaneous processing of multiple packet fields. Third, the implantation of image processing algorithms in hardware allows the grid view to be processed very efficiently.

By tailoring the processor design to the specific requirements of WSN data processing tasks, we achieved significant performance gains over the reference solution - a base station with a RISC processor. Our results, using two different approaches, show an improvement of one order of magniture in speed over a regular RISC processor. These results will enable new kinds of applications or new levels of functionality in situations where the base station must process a global view of the measurements made by the sensors.

WISENEP combines the flexibility of a RISC processor with the speed of an Application Specific Integrated Circuit (ASIC). It can process general purpose instructions (although less efficiently than a RISC processor) and it has a datapath specifically designed to handle WSN data packets. However, WISENEP will not be efficient for processing tasks other than spatial sensor data mapped into a bi or tri-dimensional grid. WSNs that do not have those characteristics will not benefit from WISENEP.

Further work includes implementing WISENEP in a FPGA and then validating the concept with respect to processor area. A physical implementation will allow us to compare power consumption, cost of the proposed solution and the actual speed in relation to a RISC processor. **.**

REFERENCES

[1] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar, “Next century challenges: Scalable coordination in sensor networks,” in *Mobile Computing and Networking*, Seattle, WA USA, 1999, pp. 263–270. [Online]. Available: citeseer.nj.nec.com/estrin99next.html

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “A Survey on Sensor Networks,” *IEEE Communications*, vol. 40, no. 8, pp. 102–114, 2002.

[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System architecture directions for networked sensors,” in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ACM Press, 2000, pp. 93–104.

[4] V. Mhatre and C. Rosenberg, “Homogeneous vs. heterogeneous clustered sensor networks: A comparative study,” in *IEEE International Conference on Communications (ICC 2004)*, Paris, France, June 2004.

[5] H. Zhang and A. Arora, “Gs3: scalable self-configuration and self-healing in wireless sensor networks,” *Comput. Networks*, vol. 43, no. 4, pp. 459–480, 2003.

[6] M. Chu, S. K. Mitter, and F. Zhao, “An information architecture for distributed inference on ad hoc sensor networks,” in *Forty-first Annual Allerton Conference on Communication, Control, and Computing*, Monticello, USA, 2003, oct.

[7] D. Ganesan, D. Estrin, and J. Heidemann, “Dimensions: why do we need a new data handling architecture for sensor networks?” *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 143–148, 2003.

[8] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 3rd ed. Prentice Hall, 2003.

[9] W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 4th ed. Prentice Hall, 1996.

[10] H. S. Stone, *High-Performance Computer Architecture; third edition*. Addison-Wesley, 1993.

[11] D.E.Comer, *Network Systems Design using Network Processors*. Prentice Hall, 2003.

[12] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Addison-Wesley, 1992.