Smart Sensor Architecture Customized for Image Processing Applications

Yulei Weng and Alex Doboli
Department of Electrical and Computer Engineering
State University of New York at Stony Brook, Stony Brook, NY 11794
Email: {wengy, adoboli}@ece.sunysb.edu

Abstract

A system level design methodology is applied to the embedded system design for a typical sensor network application: face detection for security purpose. The tradeoff analysis is performed for hardware and software implementations of the tasks in this application. The best system design is achieved with limited hardware resources.

1 Introduction

Embedded sensor networks are emerging as a key technology for applications like environmental monitoring, infrastructure security, smart buildings, manufacturing automation, and many more. These applications introduce new research challenges due to their functionality, which involves sensing the physical properties in a multitude of places (like temperature, light, sound etc), local processing of data, and global control. The main characteristics of sensor networks - that differentiate them from regular wireless networks - are: (a) large number of identical sensors densely distributed in a changing environment, (b) dynamic positioning of the sensors, (c) scarce energy resources, (d) decentralized control with a global information processing goal, and (e) simple processing and communication capabilities. Given the stringent cost and power consumption constraints, sensor nodes have limited processing power, operate at low clock frequencies, and have limited memory. However, they must operate under real-time constraints. This poses serious difficulties for the design of the node architecture, as well as the related digital, RF, and analog blocks.

This paper proposes a sensor node architecture for image processing applications, like infrastructure monitoring and security. The sensor network monitors the infrastructure with cameras, and detects human faces (or other objects of interest) from the sampled images. The detected images are preprocessed, and their characteristics are transmitted through a wireless network to a central server. Once received, the preprocessed images are searched in a global database located at the central server, such as to identify if a certain person belongs to a group of interest or not. The functionality of the sensor node includes (1) image data collection from the camera, (2) data pre-processing by the embedded system processing unit, and (3) communication and coordination with other sensor node for data transmission to the network base station.

We argue that the architecture and software of the sensor node need to be aggressively co-optimized, so that real-time constraints are met using processors operating at low frequencies, thus having low power consumption. The necessary speed increase is obtained by having special-purpose hardware accelerators in the architecture. These hardware accelerators are characteristic to image processing algorithms. As shown in Section 2, these algorithms have significant operation-level parallelism, and high uniformity of computations. This can be exploited by using vector processing, as well as customized ASIC. To identify the best sensor node architecture, we used a hardware-software co-design methodology [10]. The methodology consists of system specification, performance profiling, and hardware-software partitioning. We explored different data path resources for the accelerators, as well as different vector lengths for the vector processor. The resulting execution speedup was observed.

The proposed sensor architecture differs from other sensor nodes, which employ general purpose microprocessors or microcontrollers, like SmartDust [17], motes [15], PicoNet [13], WINS [12, 16]. We propose a customized architecture with higher execution speed than a general purpose architecture. The silicon cost (area) does not increase. Our experiments also showed that systematic exploration is needed, because the best architecture is far from being intuitive.

Section 2 details the face detection application, and the image processing algorithm. Section 3 presents the behavioral model for the vector processor. Section 4 introduces the co-design methodology that was used for the sensor unit. Section 5 provide experiments, and finally, our conclusions are offered.

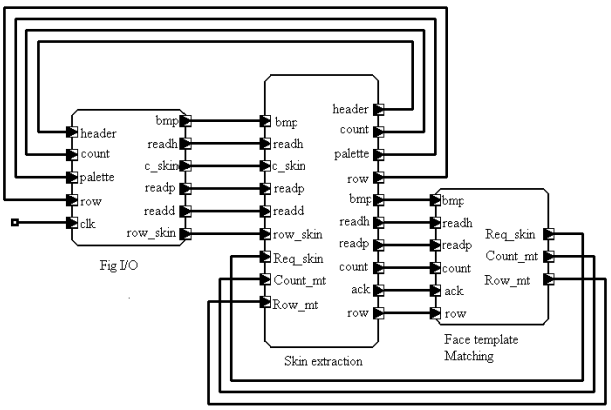

Figure 1. SystemC module for face detection

2 The Application: Face Detection

Identifying human faces in an arbitrary image is a fundamental step in video monitoring, which is an important security application. Face detection algorithms have been widely studied in several disciplines, such as image processing, computer vision, and neural network. The existing face detection techniques include knowledge-based methods, feature invariant approaches, template matching methods, and appearance-based methods [5]. Knowledge-based methods use general rules to capture the relationship between facial features, for example that the nose is between the two eyes. One disadvantage of the approach is the difficulty in generating general rules from human knowledge. Feature invariant approaches define constant features of a face, and use them as a measure for face detection. For color images, the skin color is also an effective feature, since the human skin chrominance doesn't vary a lot. To detect a skin region, we can use a YCrCb color space, and define a region for the skin tone pixels using the Cr, Cb values [6]. The template matching approach uses either a predefined template or a deformable template. The predefined template is a face or a face feature (like eyes, nose, etc). The deformable templates is an a priori elastic model that describes facial features in parameters. Face detection algorithms for deformable templates are more complex than predefined face template matching algorithms. The last category of face detection methods is appearance-based. Well known approaches include eigenfaces [7], and neural networks [8].

Since our research was focused on sensor node architecture design, selecting the best face detection algorithm became a secondary issue. Considering its popularity, we used the template matching method. The template is a full human face, and it was obtained from Michigan State University [9]. To improve the face detection result, we also incorporated a skin feature extraction method, which converts the RGB color space into the YCbCr space, and defines the skin tone pixel by selecting

skin extraction

```
for(i=0; i<image height; i++)
{
 for(j=0; j<image width; j++)
 {
       R[i][j]=ImageR[i][j];
       G[i][j]=ImageG[i][j];
       B[i][j]=ImageB[i][j];

       Cr[i][j]=0.5*R[i][j]-0.41869*G[i][j]+0.5*B[i][j];
       Cb[i][j]=-0.16874*R[i][j]-0.33126*G[i][j]+0.5*B[i][j]

       If ((-35<Cr[i][j]<40)&(Cb[i][j]<-68))
       skin[i][j]=1;     #skin region is true
       else
       skin[i][j]=0;     #not skin
 }
}
```

template matching

```
for (i=0; i<template height; i++)
{
 for (j=0; j<template width; j++)
 {
       B_T[i][j]=TemplateB[i][j];
 }
}

for(i=0; i<height; i++)
{
   for (j=0; j<width; j++)
       {
    for (ii=-template_length/2; ii<-template_length/2; ii++)
    {
     for (jj=-template_length/2; jj<-template_length/2; jj++)
     {
       correlation_coefficient=correlation_coefficient
          +(B_T[ii+template_length/2][jj+template_length/2]-B_T_mean)
          * (B[ii+template_length/2][jj+template_length/2]-B_mean)
     }
    }
   }
}
```

Figure 2. Skin extraction and template matching

Cb, Cr values within a certain region.

The face detection algorithm was modeled in SystemC. SystemC is a recent C++ based system specification language, which contains constructs for expressing both hardware and software modules. The application as well as the sensor node architecture were described in SystemC. Besides, as compared to languages like UML, StateCharts, or SpecCharts, there is an abundance of hardware simulation and synthesis tools from SystemC programs, which made the architecture development process much easier for us. Figure 1 presents the structure of the SystemC module for face detection. The SystemC specification contains two major module, one for skin extraction and the other for template matching. In order to observe the algorithm effectiveness, an image I/O module was also specified in SystemC to load and store testing images.

Figure 2 shows the pseudocode for the skin extraction and face template matching algorithms. The color space YCbCr was used for skin feature extraction. The RGB to YCbCr color space conversions were defined as

$$Y = 0.29891 \times R + 0.58661 \times G + 0.11448 \times B \quad (1)$$

$$Cb(U) = -0.16874 \times R - 0.33126 \times G + 0.50000 \times B \quad (2)$$

$$Cr(V) = 0.50000 \times R - 0.41869 \times G - 0.08131 \times B \quad (3)$$

The face template matching algorithm was implemented by correlating the face template with the image pixels. The absolute correlation coefficient is:

$$r_{ij} = \sum_{jj=-N_j/2}^{jj<N_j/2} \sum_{ii=-N_i/2}^{ii<N_i/2} (T - T_{mean})(I[i+ii][j+jj] - I_{mean}) \quad (4)$$

A face template and two experimental images were retrieved from [9] for validating the template matching specification. Figure 3 shows the face template, and the two original images. The skin extraction result, and the face template matching result are depicted in Figure 4. This motivates the correctness of our SystemC specification.

Figure 3. Face template

Figure 4. Original images and after face detection

3 Sensor Node Architecture

The smart sensor node is composed of a sensing element, a communication block, and a processing unit. The architecture is shown in Figure 5. The *sensing element* collects data (image, sound, temperature, etc) from the physical environment, and then sends data to the preprocessing unit. In our application the sensing element is the digital camera. The camera controller in the system controls data access to the digital camera. The *communication block* includes the coprocessor for the wireless communication protocol, and the RF transceiver, which incorporates analog and RF circuits, like low noise amplifiers, high-frequency filters, sample and hold, multiplexer, oscillators, PLL, and data converters. Communication in a sensor network is different from traditional wireless networks in that it requires cooperation of the sensor nodes to aggregate information and reduce redundancy. The IEEE 802.11 standard, defined for LANs, is not very effective for sensor networks. Existing communication approaches for sensor networks are (a) localized algorithms, such as directed diffusion [2], in which packets are forwarded between neighboring nodes with direction control, (b) distributed tracking algorithm IDSQ enabling sensor node collaboration based on the transmitting cost and resource constraints [3], and (c) mobile-agent-based sensor network that uses mobile agent with integrated data to reduce the communication bandwidth required for the network [4].

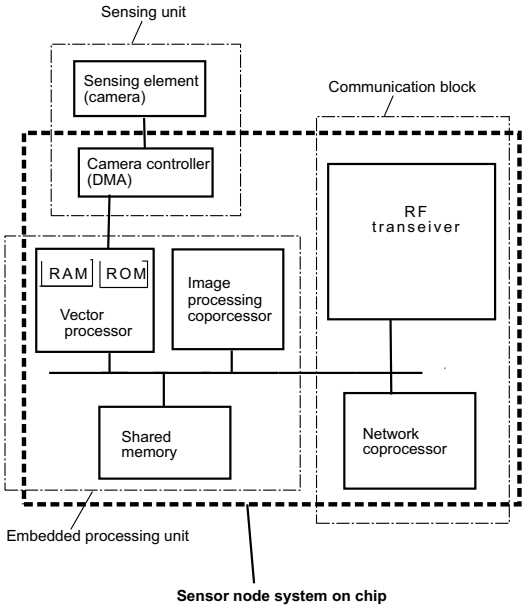

Figure 5. Sensor node architecture

The *processing unit* is composed of a vector processor, ASIC coprocessor, and shared memory. The processing unit performs the face detection algorithm. The template of the architecture follows a typical embedded system processor-coprocessor architecture [10], as shown in the left part of Figure 6. However, the template does not instantiate the attributes of the architectural blocks, like the length of the vector registers, the number of operations executed in parallel by the vector processor, the size of the RAM memory, and the resource set of the ASIC.

As explained in Section 4, we suggest using a hardware-software co-design methodology for systematic exploration of the architectural attributes, so that the speed-up of the architecture is maximized, and the silicon area constraint for the architecture is met. The co-design method was applied to partition the face detection algorithm onto the vector processor and ASIC, hence to identify the architectural resources of the vector processor and ASIC. The experiments in Section 5 prove that this process is not trivial or intuitive. Since many other sensor network application, such as environmental and sound monitoring, have similar functional attributes and per-

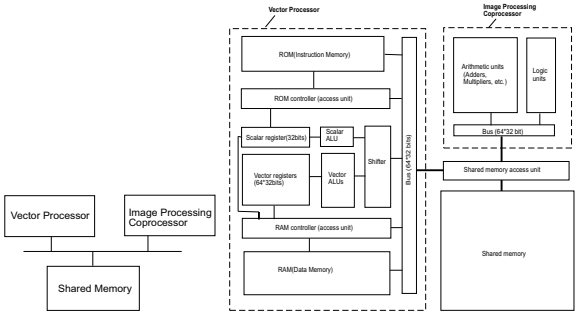

Figure 6. Processing unit architecture template

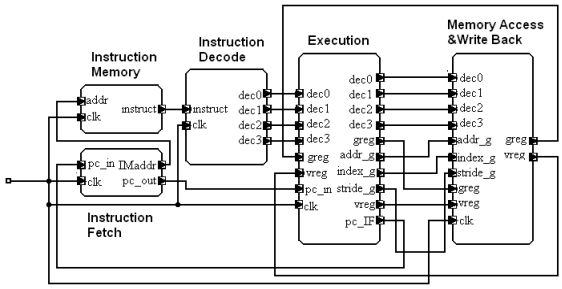

Figure 7. SystemC module of the vector processor

Logical	AND	OR	NAND	NOR
	ANDI	ORI	NANDI	NORI
Arithmetic	ADD	SUB	MULT	DIV
	ADDI	SUBI	MULTI	DIVI
	ADDV	SUBV	MULTV	DIVV
	ADDSV	SUBSV	MULTSV	DIVSV
	SUBVS	DIVVS		
Load/store	LB	SB	LH	SH
	LW	SW	LV	SV
	LVWS	SVWS	LVI	SVI
	CVI			
Control	JSR	RET	BEQZ	BNEZ

Table 1. Instruction set of the vector processor

formance requirements (like timing constraints, massive data preprocessing before transmission to the base station etc), the above embedded system design methodology can be generalized to other applications, too.

The sensor node must have sufficient processing capability, so that the input image data (set as 128×128 pixels per frame) can be processed within the time limit set for this real-time system. To fulfill the performance requirement, we used a vector processor as the core processor for the system. The vector processor's major difference from a general processor is its vector registers. For example, a typical vector register contains 64 elements, and each vector operation performs 64 computations. Since the application for our system does not require very complex control operations, the vector processor was implemented as a simplified RISC processor. The vector processor has its own data memory. The co-processor is an ASIC, and its size is limited by the available area on the silicon chip. The shared memory was included for the processor and coprocessor to exchange and store processing data.

For system level design, the register transfer level (RTL) description is too detailed. Instead, we specified the vector processor at the behavioral level in SystemC. The Synopsis Cocentric development tool was used for the SystemC module design. Figure 7 shows the resulting SystemC modules.

Following general architecture concepts, the behavioral SystemC specification divides the vector processor unit into several modules, including instruction fetch, instruction decode, execution, and memory access/write back. The specification does not include pipelining, therefore the vector operation is assumed to be finished in one instruction cycle. Pipelining results in high power consumption, thus was discarded. An instruction memory module was included in the SystemC specification to test the correctness of the vector processor specification. The SystemC model of the vector processor has a complete list of operations shown in Table 1.

4 Hardware-software Co-Design Methodology for Processing Unit Design

The sensor processing unit for the face detection application was designed using a top-down hardware software co-design methodology. The major advantage of such an approach is the ability of getting intricate knowledge of the application-specific system, and the capability of designing a cost effective co-processor, such that the implementation is able to reach the needed speed performance with minimum area cost.

The hardware-software co-design methodology was adapted after the COSYMA methodology [10]. Figure 8 depicts the co-design flow. The top-down system design methodology can be divided into the following steps. The first one is co-specification, which is the specification of both hardware and software components of the sensor node. In Section 2 and Section 3 we have introduced the SystemC specification for our vector processor and the face detection application. Following this, the data profiling step extracts the hardware and software timing and area estimation data for the specification. The next design step is co-synthesis, in which the interdependencies between hardware and software components are explored, and the application is partitioned into hardware and software to meet performance requirement (timing and area). In the co-synthesis process, the system architecture is developed based on hardware-software partitioning, resource allocation, func-

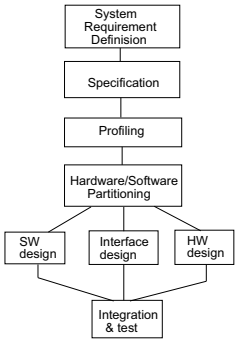

Figure 8. Hardware-software co-design flow

tionality mapping, and scheduling. Hardware-software partitioning is the central concept. It is composed of several steps: (1) functional partitioning, in which functional specifications are partitioned in to execution processes, (2) estimation of the performance for each function, and (3) allocation of processes onto different processing elements. The primary goal for partitioning is to meet a performance requirement. The secondary goal is to minimize hardware cost.

Next, we introduced the profiling step for the face detection application. Hardware software partitioning using the profiling is also illustrated.

4.1 Application data profiling

The general data profiling procedure starts from the specification of the application. By observing the specification code the whole application can be splitted to smaller task blocks. Each block may contain several lines of instructions or an instruction loop. The general rules in guiding the software timing estimation are: (1) Execution time for a task block equals the number of basic operations this task block contains. (2) Execution time for a loop in the specification equals the number of basic operations inside the loop times the iteration number. The basic operations refer to the instruction set of the core processor shown in Table 1.

The hardware timing estimation steps are: (1) Given a set of hardware resource, a hardware resource block graph was generated. (2) Based on the hardware block graph and proper operation scheduling, the time steps needed for the task blocks can be estimated.

The communication timing estimation steps are: (1) Determine the protocol setup time by selecting the proper communication protocol between processing elements. (2) Determine the bus width, then estimate the data transmission time needed for each execution block, and the time needed for data multiplexing.

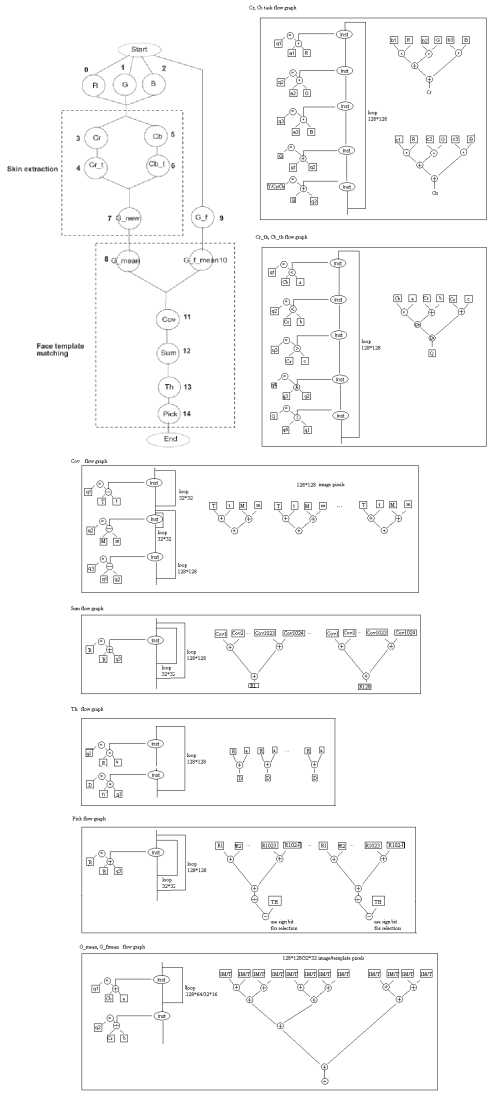

Figure 9. Task graphs for face detection

In this section, the data profiling procedure was detailed for the face detection application as an example.

4.1.1 Task graph

In order to precisely estimate the hardware area and software timing of the system implementation, the face detection algorithm was broken into small execution tasks, and a task graph was extracted to capture the data dependencies between the tasks. Figure 9 shows the task graph of face detection. Each task has its software execution representation and its ASIC execution representation.

The fifteen tasks involved in the face detection algorithm can be illustrated as:

R,G,B: Get the red, green and blue pixel color information from the image data.

Cr,Cb: Convert the RGB color space into YCbCr color space, and gets the corresponding value of Cr,Cb for each image pixel.

Cr_{th}, Cb_{th}: Compare each pixel's Cr, Cb with their thresholds to determine the face and skin region in the figure.

G_{new}: Leaves the face and skin region green color value unchanged, and sets other regions in the image to black.

G_{mean}: Calculates the means of the green color index among all the pixels in the image.

G_f: Gets the green color index in the face template.

G_{fmean}: Calculates the means of the green color index among all the pixels in the face template.

Cov: Calculates the covariance between each pixel in the face template and each pixel in the image.

Sum: Sums up the covariance in one region as large as the face template.

Th: Compares the correlation coefficient with a threshold to filter out the non face region in the image.

Pick: Selects the face region, and marks it with square.

4.1.2 Software and hardware timing and area estimation

During co-design, each of the tasks in Figure 9 can be assigned either to the vector processor, or the ASIC for the co-processor. To explore the quality of different hardware-software partitionings, the software and hardware timing and hardware area estimations were performed for each task. Table 2 shows the vector processor timing estimation in terms of instructions cycles. Table 3 shows the estimated hardware area of the vector processor. Table 4 shows the ASIC performance estimation. All the hardware area estimation in this paper were based on a standard macro cell library by SAMSUNG [11]. The library uses 0.25 micron technology.

4.1.3 Communication overhead estimation

The handshaking protocol was used for communication among the processor, co-processor, and shared memory. When an application task is assigned to the coprocessor, a corresponding communication overhead is added to the system timing.

No.	0	1	2	3
Name	R	G	B	Cr
cycle	256	256	256	1280
No.	4	5	6	7
Name	Cr_{th}	Cb	Cb_{th}	G_{new}
cycle	768	1280	768	256
No.	8	9	10	11
Name	G_{mean}	G_f	G_{fmean}	Cov
cycle	256	16	16	$16 * 128^2$
No.	12	13	14	
Name	Sum	Th	Pick	
cycle	$15 * 128^2$	$2 * 128^2$	$19 * 128^2$	

Table 2. Timing estimation for vector processor

Component	Register	Vregister	ALU	Shifter
Num	32	8	64	64
Area(mm^2)	1.3	21	0.03	0.06
Component	Instruction mem	Data mem	controller	
Num	1(0.97Mb)	1(1.6Mb)	1	
Area(mm^2)	11	18	1	

Table 3. Area estimation for vector processor

	Adder	Multiplier	Divider	Nand	Time(cycle)
get Cr	512	1024	-	-	96
Cr_{th}	1024	-	-	-	16
Cb	512	1024	-	-	96
Cb_{th}	1024	-	-	512	96
G_{mean}	1024	-	1	–	26
G_{fmean}	1024	1	-	-	11
Cov	1024	1024	-	-	32768
Sum	1024	-	-	-	163840
Th	1024	-	-	-	16
Pick	1024	-	1	-	196608

Table 4. Performance estimation for ASIC

The communication overhead was defined as the time for data transfer through the shared memory. Since the vector processor has it's internal memory, if two execution blocks are mapped to software, no communication overhead needs to be added.

The data bus capacity was set to 64×32 bit, which is the same as the vector register size. It was assumed that the data transfer speed from the processor to the shared memory is same as the internal data transfer speed. The time unit for data transmission equals an instruction cycle. The handshaking time is also added to the communication overhead.

The estimation of the total communication overhead is:

$$t_{com(in/out)} = N * (t_{unit} + t_{multiplex} + t_{prot}) \quad (5)$$

$$N = \frac{N_{result}}{N_{bus}} \quad (6)$$

$$t_{multiplex} = \frac{N_{bus}}{N_{data}} \quad (7)$$

where *N* is the number of transmission required, t_{prot} is the handshaking protocol time to setup the transmission, $t_{multiplex}$ is the time required to multiplex data bits onto the data bus, and t_{unit} is the time required to transmit one multiplexed unit (64×32 bit). N_{result} denotes the total number of bits resulted from the execution, and N_{bus} denotes the bus capacity in bits. N_{data} is the number of bits in a single execution result. For each execution blocks, two communication direction are defined, $t_{com(in)}$ refers to the communication overhead from the previous block to the current block, and $t_{com(out)}$ refers to the communication overhead from the current block to the next block.

For fine grain partitioning the communication overhead becomes significant, which affects the partitioning result drastically. Table 5 shows the estimated $t_{com(in/out)}$.

Task No.	0	1	2	3
Task	R	G	B	Cr
$N(in)$	256	256	256	256*3
$t_{com(in)}$	17152	17152	17152	17152*3
N_{result}	$128^2 * 32$	$128^2 * 32$	$128^2 * 32$	$128^2 * 32$
$N(out)$	256	256	256	256
$t_{com(out)}$	17152	17152	17152	17152
Task No.	4	5	6	7
Task	Cr_{th}	Cb	Cb_{th}	G_{new}
$N(in)$	256	256*3	256	256
$t_{com(in)}$	17152	17152*3	17152	17152
N_{result}	524288	524288	524288	524288
$N(out)$	256	256	256	256
$t_{com(out)}$	17152	17152	17152	17152
Task No.	8	9	10	11
Task	G_{mean}	G_f	G_{fmean}	Cov
$N(in)$	256	16	16	
$t_{com(in)}$	17152	1072	1072	18228
N_{result}	524320	32768	32800	524288
$N(out)$	257	16	17	256
$t_{com(out)}$	17155	1072	1075	17152
Task No.	12	13	14	-
Task	Sum	Th	Pick	-
$N(in)$	256	256	256	-
$t_{com(in)}$	17152	17152	17152	-
N_{result}	524288	524288	32	-
$N(out)$	256	256	1	-
$t_{com(out)}$	17152	17152	3	-

Table 5. Communication overhead estimation

4.2 Hardware-software partitioning

The hardware-software partitioning algorithm uses the simulated annealing heuristics under the guidance of following cost function [10]:

$$dc(B) = [t_{HW}(B) + t_{com}(B) - t_{ov}(B) - t_{SW}(B)]It(B) \quad (8)$$

$$t_{com}(B) = \Sigma t_{com(in)}(B) \bigcup t_{com(out)}(B) \quad (9)$$

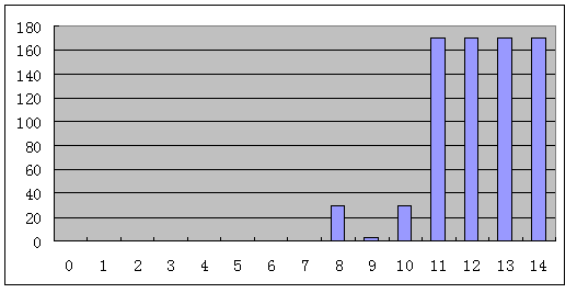

Figure 10. Hardware extraction statistics

where *B* is the execution block listed above. $dc(B)$ is the decrement of the cost. $t_{HW}(B), t_{SW}(B), t_{com}(B)$ are the hardware timing, software timing, and the communication overhead. t_{ov} is the hardware and software overlap time. Each execution block *B* is used only once in our application, therefore the iteration factor *It(B)=0*. The partitioning cost function is similar to the COSYMA cost function [10]. However, we implemented the partitioning control parameters external to the cost function to avoid over-design.

4.3 Partitioning results

Applying the partitioning algorithm, we mapped the tasks in the face detection application into software and hardware, and achieved a speed-up of about 2.0 using an additional ASIC co-processor that contains 1024 adders, 1024 multipliers, 1 divider and 512 NAND gates. The software timing is 839,952 cycles, while the hardware and software timing result is t_{SWHW}=407,734 cycles. The best partitioning solution the annealing process found was to move tasks 11, 12, 13, and 14 to hardware. The hardware extraction statistics in Figure 10 shows that the above task combination is the most often extracted partitioning solution.

4.4 Silicon area estimation

The silicon area of the vector processor was estimated to be $54mm^2$. The co-processor area was estimated by summing up all the required hardware area for the extracted execution blocks, which was shown in Table 6. The final hardware extraction solution to the face detection application was to move tasks 11, 12, 13, and 14 to hardware. Therefore, the co-processor area plus the shared memory is about $268mm^2$ (Ignoring NAND and the divider area). The total silicon area is $323mm^2$. Hence, the silicon implementation of the sensor node requires the largest MOSIS package (PGA391L), which has a die area of $18^2 = 324mm^2$.

Task combination	adder	muliplier	shared memory
11,12,13,14	1024	1024	1Mb
area(mm^2)	25	240	3

Table 6. Silicon area for the partitioned blocks

5 Discussion of Co-design Tradeoffs

5.1 Reducing ASIC area

The previous implementation (Subsections 4.3 and 4.4) has a much larger ASIC area than the vector processor. To further explore possibilities for hardware area reduction, we conducted more experiments for different ASIC area:

Case 1: 1024 Adders, 1024 Multipliers, 512 NAND.

Case 2: 512 Adders, 512 Multipliers, 512 NAND.

Case 3: 512 Adders, 128 Multipliers, 512 NAND.

Case 4: 128 Adders, 512 Multipliers, 512 NAND.

Case 5: 128 Adders, 128 Multipliers, 512 NAND.

A data profiling process was performed for each of the cases. The software timing and area estimation remain the same. The ASIC time profiling for each task in the application was shown in Table 7.

	Case1	Case2	Case3	Case4	Case5
Cr	96	192	416	320	512
Cr_{th}	16	32	97	386	386
Cb	96	192	416	320	512
Cb_{th}	16	32	97	386	386
G_{mean}	26	41	41	135	135
G_{fmean}	11	11	11	15	15
Cov	32768	65536	163840	163840	262144
Sum	163840	163840	163840	212992	212992
Th	16	32	32	128	128
Pick	196608	196608	196608	245760	245760

Table 7. Timing for different ASIC areas

As the ASIC silicon area decreases, the execution time doesn't necessarily increase in the same scale. Therefore, it is possible to obtain a design with smaller hardware area, but similar execution time. The hardware-software partitioning algorithm is applied to these cases, and Table 8 shows the tasks extracted to hardware, the speed-up, and ASIC area for each case.

From the experimental result it is shown that Case 2 only takes around half of the hardware area of Case 1, whereas the timing remains almost the same. Hence, it can be concluded that an upper limit exist for the speed-up as the hardware area increases. Above the limit, the hardware area increment stops

	HW tasks	Speed up	Adder	Multiplier	Memory	Area (mm^2)
1	11-14	2	1024	1024	1Mb	268
2	11-14	1.9	512	512	1Mb	136
3	11-14	1.56	512	128	1Mb	76
4	11-14	1.33	128	512	1Mb	129
5	12-14	1.15	128	128	1Mb	70

Table 8. Partitioning results for different ASIC compositions

to offer performance enhancement. Comparing Case 3 and Case 4, it can been seen that the number of adders contributes more to speed up than the number of multipliers. This is due to the fact that the tasks in the face detection application have more addition steps than multiplication steps. Thus, allocating more adders results in higher speed-up. The other advantages of adders is the much smaller area it has over the multipliers. From the area comparison of Case 3 and Case 4, it can been seen that Case 3 takes much lesser silicon area. When the ASIC area is strictly limited, assigning tasks to the ASIC co-processor may not be able to achieve significant speed-up. This situation is illustrated by Case 5.

This experiment shows that the allocation of hardware resources is an important decision that needs to be made in system level design. The partitioning results for different allocation possibilities is very helpful in making the appropriate architectural choices. For instance, for the face detection application, if the designer is limited with the ASIC area but needs a high speed-up, then the best sensor architecture will correspond to Case 3, which gives a speed-up of 1.56 at the expense of a small ASIC area ($76mm^2$), which is compatible to the vector processor area ($54mm^2$).

5.2 Vector processor design

The vector processor's computational capability depends on the number of elements inside the vector register. The more elements are contained, the more concurrent computations the vector processor is able to support. Since the vector register takes up a large area on the chip, it is important to explore the relationship between the vector processor speed and the silicon area it takes.

The face detection application is a real time application that requires a high processing speed. Thus, the vector processor design is limited by the execution time. Table 9 shows the execution time that the vector processor can achieve under different vector register sizes and ALU numbers, and the speed-up of the four ASIC design options. In these experiments, the communication overhead is assumed to remain the same, because the bus width is set to 64×32 bits. It can be seen that the

execution time doesn't very largely as the number of ALU and the number of elements in the vector register is reduced. This counterintuitive result is due to the partitioning of the application tasks, which moves most of the data intensive tasks to the ASIC coporcessor. As we mentioned before, further increasing the ASIC co-processor area could not gain more speed-up because of the data dependencies between operations. Therefore, a upper limit for the system performance exists. Depending on the hardware area and system timing requirements set by the application, an optimal design solution can be chosen from the candidates in Table 9

ASIC (Add,Mult)	Vector Register	ALU num	SW timing	SW+HW timing	speed up
(1024,1024)	64	64	839952	407734	2
(512,512)	64	64	839952	432076	1.9
(512,128)	64	64	839952	538432	1.56
(128,512)	64	64	839952	631542	1.33
(128,128)	64	64	839952	730394	1.15
(1024,1024)	32	32	1679904	409732	4.1
(512,512)	32	32	1679904	430734	3.9
(512,128)	32	32	1679904	480124	3.5
(128,512)	32	32	1679904	671962	2.5
(128,128)	32	32	1679904	763584	2.2
(1024,1024)	8	8	6719656	447970	15
(512,512)	8	8	6719656	448012	15
(512,128)	8	8	6719656	516893	13
(128,512)	8	8	6719656	678752	9.9
(128,128)	8	8	6719656	781354	8.6

Table 9. Vector register size and performance

5.3 Vector processor vs. ASIC

The vector processor is a good candidate for data intensive application compared to general RISC processor, because it is capable of parallel computation. The experiment shows that when the co-processor ASIC area is strictly limited (for example, in the case of 128 adders and 128 multipliers), the execution speed-up is also reduced, and moving tasks to the co-processor does not offer much execution speed. When hardware area is not the limiting factor, applying the tasks to the ASIC co-processor can achieve significant speed-up, as shown in our experimental results.

5.4 Reducing shared memory size

The shared memory in the embedded processing unit stores the intermediate results for the image processing tasks that are mapped to the co-processor. In many image processing cases the information stored in the memory cells are highly repetitive. By eliminating the redundant information, the shared memory size can be reduced. However, this approach sometimes requires a repetitive computation for the intermediate result, and may increase the processing period. Careful exploration into the memory size and processing speed tradeoffs needs to be performed to get the optimal implementation.

6 Conclusion

This paper proposes a sensor node architecture for image processing applications, like infrastructure monitoring and security. The processing unit is customized for face detection applications. A top-down hardware-software co-design methodology was used to design the embedded sensor processing unit. The application data profiling and the hardware/software partitioning are two most important steps in the co-design process. Experiments with the face detection application show that an appropriate choice of the resource set of the co-processor ASIC results in optimal silicon area and speed-up combinations. The best architecture found by co-design extracted four tasks to the ASIC co-processor, which had 512 adders and 128 multipliers. The co-processor area ($76mm^2$) was compatible to the vector processor area ($54mm^2$). A speed-up of 1.56 compared to pure software implementation was achieved with this design. Our experiments also concluded that systematic exploration is needed, because the best architecture is far from being intuitive.

References

[1] J. Stankovic, T. Abdelzaher *et al*, "Real-Time Communication and Coordination in Embedded Sensor Networks", *Proc. of the IEEE*, Vol. 91, No.7, July 2003, pp. 1002-1022.

[2] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next Century Challenges: Scalable Coordination in Sensor Networks", *Proc. Int. Conf. Mobile Computing and Networking (MOBICOM)*, 1999, pp. 263-270.

[3] F. Zhao, J. Shin, and J. Reich, "Information-driven Dynamic Sensor Collaboration for Tracking Applications", *IEEE Signal Processing Magazine*, Vol. 19, pp. 61-72, March 2002.

[4] H. Qi, S. S. Iyengar, and K. Chakrabarty, "Multi-resolution Data Integration using Mobile Agents in Distributed Sensor Networks", *IEEE Trans. Syst., Man, Cybern.*, Vol. 31, pp. 383-391, Aug. 2001.

[5] M.-H. Yang et. al., "Detecting Faces in Images: A Survey", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.24, No. 1, Jan. 2002.

[6] D. Chai and K.N. Ngan, "Locating Facial Region of a Head-and-Shoulders Color Image", *Proc. Third Int'l Conf. Automatic Face and Gesture Recognition*, pp. 124-129, 1998.

[7] M. Kirby and L. Sirovich, "Application of the Karhunen-Loe've Procedure for the Characterization of Human Faces", *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 12, No. 1, pp. 103-108, Jan. 1990.

[8] J.-E. Villet, and M. Collobert, "A Fast and Accuracte Face Detector Based on Neural Networks", *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 22, No. 1, pp. 42-53, Jan. 2001.

[9] *http://www.cogsci.msu.edu/fasp/*

[10] R. Ernst *et al*, "Hardware-software cosynthesis for microcontrollers", *IEEE Design and Test of Computers*, 10, No.4, Dec.1993, pp. 64-75.

[11] *http://www.samsung.com/Products/Semiconductor/ASIC/IPCoreLibrary /IntellectureProperties/MemoryCores/DRAMFlash/*

[12] A. Abidi, G. J. Pottie, W. J. Kaiser, "Power Conscious Design of Wireless Circuits and Systems", *Proc. of the IEEE*, Vol. 88, No. 10, October 2000, pp. 1428-1455.

[13] F. Benett, D. Clarke, J. B. Evans, A. Hopper, A. Jones, D. Leask, "PicoNet: Embedded Mobile Networking", *IEEE Personal Communication Magazine*, 4(5): 8-15, Oct. 1997.

[14] W. R. Heinzelman, A. Chandrakasan, H. Balakrishnan, "Energy-Efficient Communication Protocol for Wireless Microsensor Networks", *Hawaii Int'l Conf. on System Sciences*, 2000, pp. 2-12.

[15] J. Hill, D. Culler, "A Wireless Embedded Sensor Architecture for System-Level Optimization", *Technical Report*, UC Berkeley, 2001.

[16] G. J. Pottie, W. J. Kaiser, "Wireless Integrated Network Sensors", *Communications of the ACM*, Vol. 43, No. 5, 2000, pp. 51-58.

[17] B. Warneke, M. Last, B. Liebowitz, K. Pister, "SmartDust: Communicating with a Cubic Millimeter Computer", *IEEE Computer*, January 2001, pp. 2-9.

[18] T. Grotker, S. Liao, G. Martin, S. Swan, "System Design with SystemC", *Kluwer*, 2002.