Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 615824, 10 pages
doi:10.1155/2012/615824

Research Article

Hardware Middleware for Person Tracking on Embedded Distributed Smart Cameras

Ali Akbar Zarezadeh[1] and Christophe Bobda[2]

[1] *Department of Computer Science, University of Potsdam, 14482 Potsdam, Germany*
[2] *Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR 72701, USA*

Correspondence should be addressed to Christophe Bobda, bobda@acm.org

Received 29 April 2011; Accepted 6 December 2011

Academic Editor: Ron Sass

Tracking individuals is a prominent application in such domains like surveillance or smart environments. This paper provides a development of a multiple camera setup with jointed view that observes moving persons in a site. It focuses on a geometry-based approach to establish correspondence among different views. The expensive computational parts of the tracker are hardware accelerated via a novel system-on-chip (SoC) design. In conjunction with this vision application, a hardware object request broker (ORB) middleware is presented as the underlying communication system. The hardware ORB provides a hardware/software architecture to achieve real-time intercommunication among multiple smart cameras. Via a probing mechanism, a performance analysis is performed to measure network latencies, that is, time traversing the TCP/IP stack, in both software and hardware ORB approaches on the same smart camera platform. The empirical results show that using the proposed hardware ORB as client and server in separate smart camera nodes will considerably reduce the network latency up to 100 times compared to the software ORB.

1. Introduction

Smart cameras are embedded systems for performing image analysis directly at the sensor, and thus delivering description of scene in an abstract manner [1]. Networks of multiple smart cameras can be used to efficiently implement features like detection accuracy, fault tolerance, and robustness that would not have been possible with a single camera in such applications like surveillance or smart environments. In such networks, smart camera nodes are usually embedded systems with very tight design and operation requirements. Besides complex computations that must take place in real time, communication among the cameras has to be handled as well. In embedded environment, the complex computation can efficiently be handled using a combination of hardware and software. While the hardware handles the most computational demanding tasks, the software takes care of the control parts. By using FPGA as the main computational component, complex computation can be directly implemented in hardware, while the control part can be carried out by an embedded processor. We used this approach in the design of an FPGA-based embedded smart camera platform [2]. Also, in our previous works [3, 4], we proposed a software/hardware codesign middleware approach, called hardware ORB. The motivation for the hardware ORB was to provide a low-latency intercommunication with deterministic behavior among the smart camera nodes. The middleware enables developers to program distributed applications seamlessly, without having to focus on the details of the implementation, thus reducing the development time [5].

The middleware provides designers to transparently describe distributed systems. A lightweight agent-oriented middleware for distributed smart cameras was introduced by Quaritsch in his PhD thesis [1]. His middleware implements basic networking and messaging functionality for collaborative image processing. There are some existing hardware implementations of ORB which are exploited in the area of software-defined radio (SDR) systems. The integrated circuit ORB (ICO) engine [6, 7] implements a hardware CORBA ORB. PrismTech's ICO is such an ORB in SDR applications.

The ICO engine is responsible for implementing the transfer syntax used in CORBA messages.

According to the network performance analysis, the Linux kernel must be manipulated to collect timing measurements on internals. The data streams kernel interface (DSKI) instrumentation software was made for Linux [8]. It supports gathering of system behavioral data from instrumentation points inserted in kernel and application source. Demter et al., in [9], demonstrated the performance analysis of the TCP/IP stack for the Linux kernel, and then the time traversing of packets for each layer of the Linux kernel TCP/IP stack was analyzed. Their idea to achieve network performance analysis inspired us to follow the same methodology with some modification for extraction of the network latencies.

Adequate coverage of the critical areas is not feasible with only one camera. Inevitably, multiple cameras must observe an *overlapping* scene from different views. It provides robustness against occlusion and improves accuracy [10]. The observations of an object on different captured video streams must be identified as the same object. It is of crucial importance to determine the correspondence between different views. Khan and Shah [10] introduced a pure geometry-based approach without need for camera calibration parameters. Calderara et al. [11] proposed another class of geometry-based approach called homography and epipolar-based consistent labeling for outdoor park surveillance (HECOL). As the highlighted benefit, their method identifies people in groups by leveraging the probabilistic Bayesian theory.

In [3], we proposed Common object Request Broker Architecture (CORBA) in the compact profile as a software middleware. Additionally, the hardware ORB for server side was proposed to overcome the limitations of the software ORB running on an embedded platform in real-time applications. The concept of the hardware ORB was extended in two steps in [4]: (i) a new client hardware ORB as a pair of the sever hardware ORB and (ii) performance analysis of both hardware and software approaches for systematic measurement of the latencies.

This paper adds a distributed multicamera tracker layer with jointed field of view (FoV) on top of our hardware ORB middleware. Tracking being a key part of high-level visual algorithms, a wide variety of applications, like surveillance, human motion analysis, and traffic monitoring, can benefit from the results provided in this paper. The simultaneous association process in an overlapped camera setup demands twofold constraints: (i) a real-time stand-alone tracker at each camera node and (ii) tight data exchange mechanism among multiple cameras with deterministic behavioral. The existing works mostly concentrate on the only software implementations of vision algorithms and communication modules, leaving the optimal hardware/software implementation, in particular in the embedded system open. By presenting such typical vision application, first our novel system-on-chip (SoC) architecture of a real-time stand-alone tracker based on an FPGA-based smart camera is demonstrated. It functions via segmentation process (object detection) and object association (identification) steps. Segmentation algorithm demands an impressive amount of computational power. By hardware acceleration of the segmentation, we achieve an efficient implementation which guaranties a real-time performance. Second, we prove the viability and versatility of our proposed communication infrastructure based on the hardware ORB. Considering the parallelized internal nature of our hardware ORB, it appropriately satisfies such constraints dictated by the upper application layer.

The rest of the paper is organized as follows: first, an explanation of the methodology for finding the correspondence of an object which appears simultaneously on multiple overlapping cameras is given in Section 2. In Section 3, an architectural view of the proposed system inside the smart camera is presented. Afterwards, Section 4 is dedicated to explain the protocol stack performance analysis on the basis of probing strategy implementation. Section 5 is devoted to the experimental results. It reveals the obtained empirical results. Finally, the paper is briefly concluded in Section 6.

2. Tracker System

The focus of this section is the multiperson tracking in an overlapping multi-camera setup. The tracking system classifies all the observations of the same person, grabbed from different cameras, into trajectories. In the case of failure in a camera node, the tracking process can uninterruptedly be continued with other cameras, and trajectories of persons from various views can be merged [12]. Regarding the occlusion of the target person, the trajectories still can be kept updated in all cameras via exchanging the extracted abstract features in multiple cameras. The usage of multiple cameras requires keeping consistent association among the different views of single persons. Put simply, it assigns the same label to different instances of the same person acquired by different cameras. This task is known as consistent labeling [10, 11]. We divide the whole procedure into two off-line and on-line phases. At the off-line phase by reidentification of a known person who moves in the camera FoVs, we extract the FoV lines and homography matrix by means of multiview geometry computations. Using the determined hypothesis from the off-line phase, we find the correspondence between different simultaneous observations of the same person in on-line phase.

2.1. Off-Line Phase. The off-line process aims at computing overlapping constraints among camera views to create the homography. This process applies for each pair of overlapped cameras. It performs the computation of the entry edges of field of view (EEoFoV). As the upper-left side of Figure 1 shows, two adjacent cameras C^1 and C^2 have partially overlapping view scenes. Our goal is to extract the FoV lines associated to the C^2 in C^1 and vice versa. The lower-left side of Figure 1 depicts the defined overlapping zones $Z^{1,2}$ and $Z^{2,1}$ between the cameras C^1 and C^2, respectively, assuming a single person walks through the FoVs of the C^1 and C^2 and crosses the EEoFoVs of each individual camera. Once the person appears in the FoV of a camera, an stand-alone tracker is allocated to him/her.

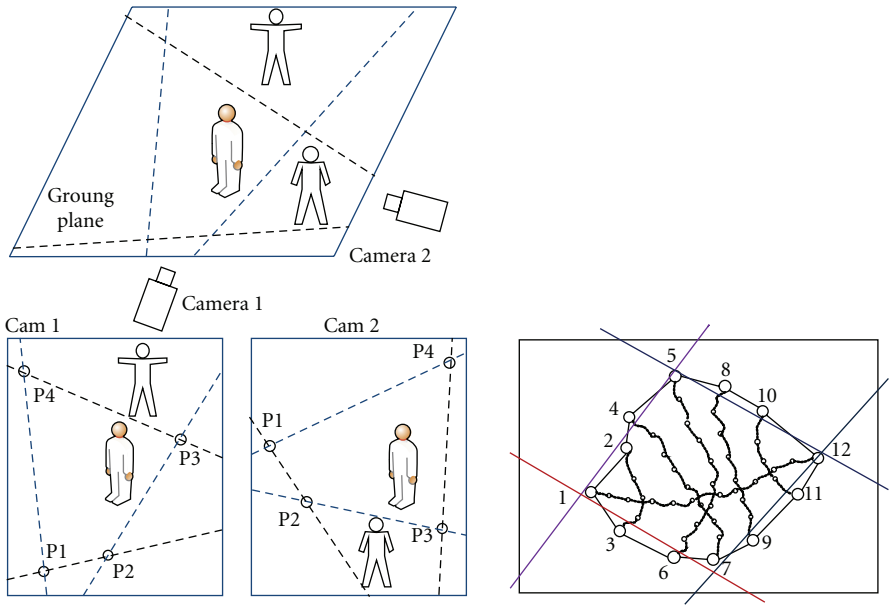

Figure 1: Detecting the overlapping zones based on FoV lines (left), and gathered person trajectories in off-line phase (right).

In each new scan of the scene, the tracker keeps the other neighbor camera updated about new position (lower support point) of the person. Integration of all the positions represents the person's trajectory.

As illustrated in the right side of Figure 1, after gathering adequate number of trajectories, we collect all points which lay on the EEoFoVs and establish a set as $\Gamma^{1,2} = \{p_1^{1,2}, p_2^{1,2}, \ldots, p_{12}^{1,2}\}$, where the subscripts indicate corresponding points in the two cameras. Afterwards, we elicit a polygon, which encompasses all points $p_i^{1,2} \in \Gamma^{1,2}$ with the biggest covered area. In order to compute this polygon, we need to find out if two points in the set $\Gamma^{1,2}$ are adjacent. Our strategy consists of sorting out this set of points based on the $p_i^{1,2}$ of the X and Y coordinates. First, we sort out all the $p_i^{1,2}$ points based on the X values and then construct a new set of points called $\Omega^{1,2}$. Now, to each element of the $\Omega^{1,2}$ a label equal to the element's position at the set is assigned. Afterwards, we sort out the $\Omega^{1,2}$ considering the elements' Y values and then form a new set of points referred to as $\Theta^{1,2}$. Figure 2 shows a 2D representation of the *EEoFoV* points where each column is the $\Theta^{1,2}$ (sorting based on Y), and each element of the column is the label assigned from sorting based on $X(\Omega^{1,2})$. Let us start from the label 1 (point with the lowest X value). At the first column from the position of label 1, we look for its nearest adjacent point based on the minimum Euclidean distance in the direction toward the bigger Y values. The point with label 3 has the minimum Euclidean distance. We continue this procedure up to the last element of the list with the highest Y value. Afterwards, we change the direction backward from the highest to the lowest Y values. After several iterations, all edges are computed.

Figure 3 shows the final computed graph where all vertexes are the EEoFoV points sorted based on the X values and

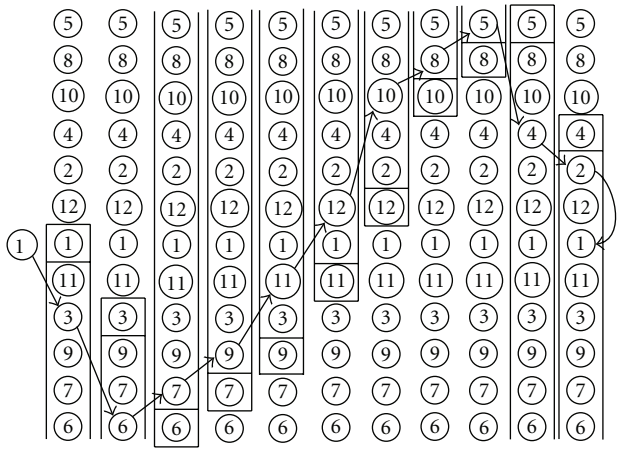

Figure 2: Arrangement of gathered *EEoFoV* points based on X and Y coordinate sorting.

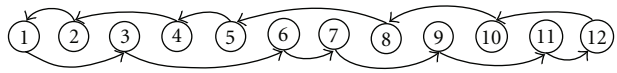

Figure 3: Final computed graph representing the extracted polygon.

all edges are results of previous iterations. Hence, the polygon is extracted.

According to the sign alteration of the slope of lines, connecting the adjacent points (the right side of Figure 1), we classify all the points into four groups. Next, by means of linear regression, the best line on each group is fitted, and thus, all FoV lines are computed. Intersection points between

the four lines in the cameras C^1 and C^2 (the four corners of the $Z^{1,2}$ and $Z^{2,1}$) are the best candidates for computation of homography matrix [11]. The homography matrix $H^{1,2}$ from camera C^1 to camera C^2 is formulated as

$$\begin{bmatrix} x_k^{1,2} \\ y_k^{1,2} \\ 1 \end{bmatrix} = \begin{bmatrix} h_{1,1}^{1,2} & h_{1,2}^{1,2} & h_{1,3}^{1,2} \\ h_{2,1}^{1,2} & h_{2,2}^{1,2} & h_{2,3}^{1,2} \\ h_{3,1}^{1,2} & h_{3,2}^{1,2} & h_{3,3}^{1,2} \end{bmatrix} \times \begin{bmatrix} x_k^{2,1} \\ y_k^{2,1} \\ 1 \end{bmatrix}. \quad (1)$$

2.2. On-Line Phase. At the on-line process, once a camera node detects a new object, it sends the person's position together with an accompanying wall clock time stamp to all overlapped neighboring nodes. Hence, each node has its own local knowledge of the detected person and also the external information received from its neighbors. The correspondence between the local and external information must be accomplished. Based on the computed FoV lines at the off-line phase, we determine whether the detected person is in the overlapping zone or not. If this condition is asserted, with using the computed ground-plane homography matrix, we convert the person's locally extracted lower support point from the local coordinate to the external coordinate corresponding to the neighbor camera node. By using the temporal information (time stamp) and minimum Euclidean distance between the spatial data (e.g., distance between position of the person in local and external observations), we infer the correspondence. Provided that the person is only in the FoV of one camera, the history of that individual is applied to determine whether he is a new person or the one who has been observed previously.

The on-line process must be performed repeatedly for every acquired new video frame at all cameras. Thus, it imposes two key constraints on the entire design: (i) a real-time stand-alone tracker at each camera node and (ii) a real-time data exchange mechanism among multiple cameras. Next Section 3 covers the architectural aspect of our implemented system inside the smart camera to tackle processing and communication tasks.

3. Smart Camera Internal Architecture

The focus of this section is the structure of hardware/software inside the FPGA, which is the computing element inside our smart cameras. To have an efficient partitioning of the hardware/software, we use different processes for implementation of the communication and computer vision tasks. Our architecture consists of two processes: *computer vision module* (CVM) and *communication module* (CM).

3.1. Computer Vision Module. The *CVM* is a framework to launch the upper intelligence layer, which contains all the required computer vision algorithms for processing the captured raw video stream. As Figure 4 shows, it consists of two threads: *serverPeer_Thread* (SPT) and *ClientPeer_Thread* (CPT).

The *CPT* listens to the receiving events from the *CM* process inside the node. Based on the initial configuration, it

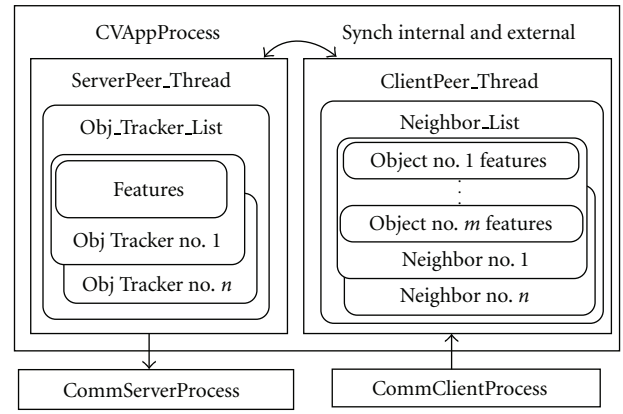

Figure 4: *Computer vision processor* internal architecture.

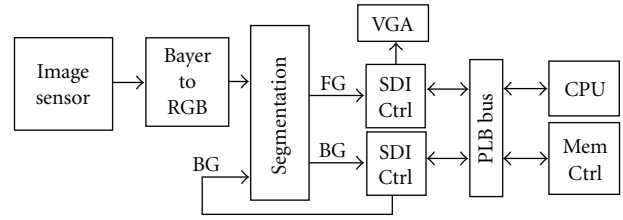

Figure 5: The hardware accelerated segmentation module.

generates a neighbor list, which includes all the node's neighbors inside the cluster. At run time, this list keeps track of persons for a particular neighbor node. The *SPT* is designed to get query from the image sensor and subsequently running upper layer computer vision algorithms cyclically. In each cycle, it runs a locally stand-alone multiperson tracker for detection and tracking purposes. Segmentation plays explicitly a basic role for labeling the pixels as foreground (FG) or background (BG) pixels. The segmentation is performed by the method proposed by Horprasert et al. [13]. To gain real-time performance in such a low level pixel processing unit, the segmentation process is totally offloaded into the hardware. Figure 5 shows the SoC for the segmentation process.

The SoC design presents a pipelined chain of IP cores from grabbing the raw image, conversion from the Bayer to RGB patterns, segmenting, and streaming data flow. The integrated algorithm inside the segmentation module compares an initially trained BG image to the current image. The decision for each pixel, whether an FG or BG, is made based on the color and brightness distortions properties. Two streaming data interface (SDI) controllers are employed for handling the streaming data flow between a given individual IP core and memory [14].

The output FG image of the hardware segmentation module is readily accessible from the software tracker in a preassigned block of memory. The tracker exploits the FG image to detect the moving persons in the scene. A new instance of the tracker is instantiated for each new person in the scene. Each instance holds the extracted features for an individual person. The *SPT* participates on finding

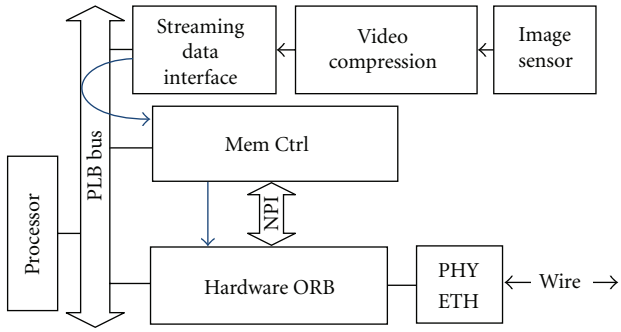

Figure 6: The usage of the hardware ORB for video streaming.

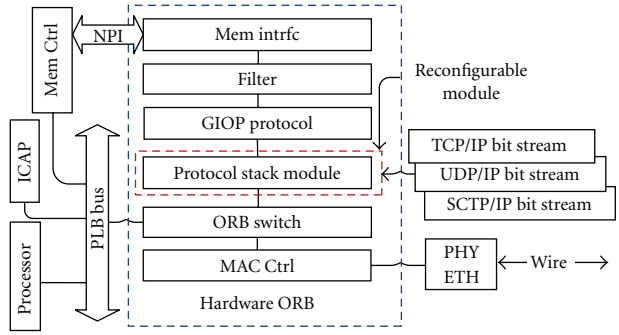

Figure 7: The general architecture of the hardware ORB IP core.

the association between its local person's features and the external person's features kept by the *CPT*. The mechanism for the discovery of such a correspondence is based on the tracking algorithm explained in Section 2.

3.2. Communication Module. The *CM* acts as the output/input gates for pushing/pulling the locally generated events by the *CVM* to the cluster of cameras. It has an interconnection with the *CVM* through some of interprocess facilities, for example, shared memory. The hardware ORB is the core engine of the *CM*. As a CORBA-based middleware IP core, the hardware ORB allows clients in a real-time manner to invoke operations on distributed objects without concern about communication protocols. The hardware ORB server is responsible for replying determined requests of the CORBA client hosted in a GPP or another smart camera platform. Hence, a predefined block of memory can be used as a CORBA servant object. Figure 6 shows the usage of the hardware ORB as a video streaming solution for the acquired video footage. In this case, it is supposed that the compressed information of video stream resides in a determined location of memory. Therefore, upon the request of a client node in another smart camera node, the hardware ORB fetches such processed data from memory and then directly serializes and transmits it through the protocol stack towards the client node. Figure 7 shows the overall anatomy of the hardware ORB IP core. The most considerable feature of such architecture is to have a redundant offloaded protocol stack in parallel operation with the embedded processor. Its main advantage is to have the possibility of using the network services on the operating system and also utilizing another software ORB running on the embedded processor. Besides the usage of the software ORB, the hardware ORB plays the role of an auxiliary deterministic and real-time performance communication engine. Considering Figure 7, the *ORB switch* shares the MAC layer between the processor and the hardware ORB. Basically, the hardware ORB can exchange the contents of blocks of the memory through general inter-ORB protocol (GIOP), without the usage of the embedded processor. It implements the lower layers of the standard TCP/IP stack and also supports the direct access to DDR memory with the native port interface (NPI) of multiport memory controller (MPMC). Figure 8 shows the internal architecture of the IP core as client/server node in more details. To initiate a new transaction in the client side, the embedded processor must activate the transmission path by writing in a specific register of the IP core. Then, based on the defined responsibilities of GIOP such as common data representation (CDR) and interoperable object reference (IOR), it constructs the *request* message format and sends it down through the protocol stack. Afterwards, there are two separate transmission buffers, one for the embedded processor and another for the hardware ORB, called *Tx buffer* and *hard ORB Tx buffer*, respectively. The *ORB switch* connects one of the two buffers to the *MAC Tx controller* for transmission.

The hardware ORB's server mechanism is semantically similar to the client's one. Once the server receives a request packet from the Rx path of the protocol stack, if the packet satisfies the filter criteria (i.e., the predefined TCP port number and CORBA interoperable object reference), the server fetches the relevant data from memory by a module, called *reader controller*. Then the hardware ORB constructs the TCP/IP packet and finally stores it inside the *hard ORB Tx buffer* for transmission via the *MAC Tx controller*.

Once the client receives the *reply* from the server, the replied packet comes up through the receiving path of the hardware ORB and finally is stored in the memory by means of the *writer controller*. Moreover, the embedded processor is connected to the TX and RX dual port buffers of the MAC layer (*Tx buffer* and *Rx buffer*) by means of the PLB slave interface, and it enables the protocol stack on Linux. Referring to Figure 8, once a new packet arrives from the *MAC Rx controller*, one copy is placed inside the *Rx buffer* of the embedded processor and routed at the same time to the Rx path of the protocol stack of the hardware ORB. The hardware ORB listens to the incoming packets and looks for the specific ones with a predefined TCP port number and CORBA IOR. If the incoming packet satisfies the filter criteria, first, with the masking of the Rx interrupt to the embedded processor, the hardware ORB blocks the interface to the embedded processor. It avoids racing between the hardware ORB and the embedded processor. Afterwards, the hardware ORB copies the received data to DDR memory by a module called *writer controller*.

4. Protocol Stack Performance Analysis

In this section, the performance analysis methodology on the basis of probing strategy is explained. We use the terminology of [15] to characterize the performance parameters

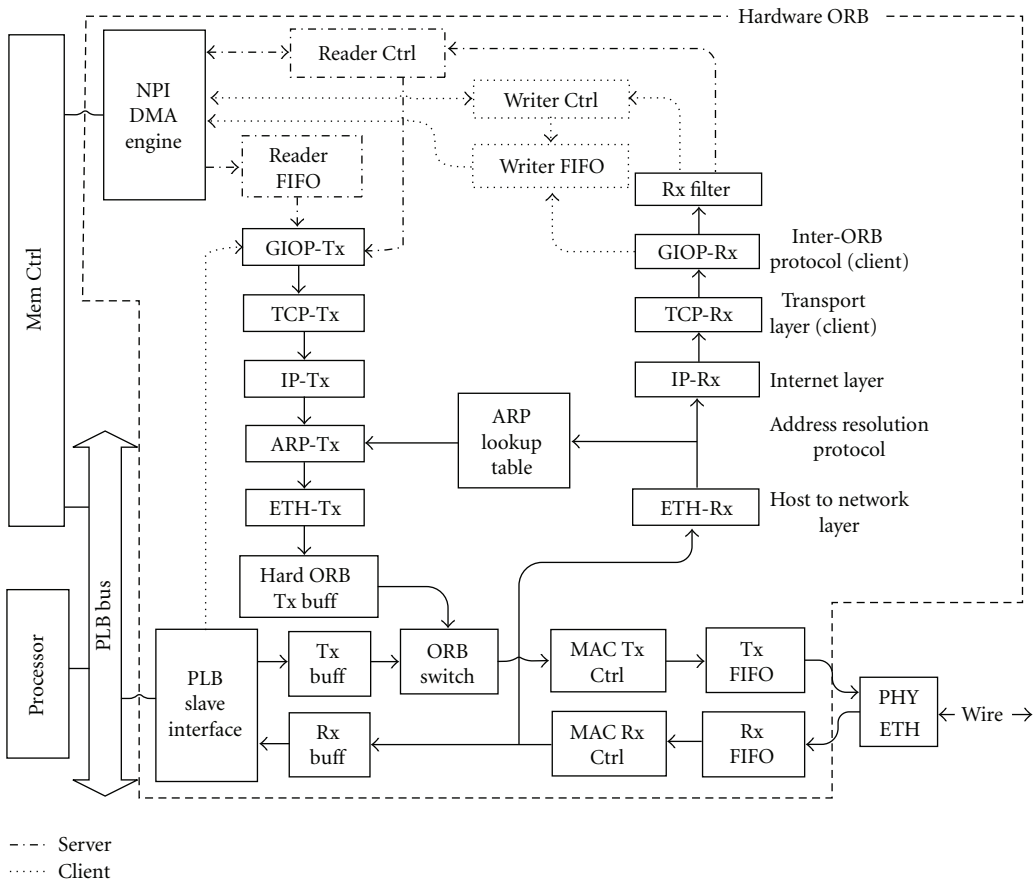

Figure 8: The architecture of the hardware ORB client/server IP core.

(i.e., time of flight, sending/receiving overheads, injection/reception, transmission time, and bandwidth). For a network with only two devices, the bandwidth for *injection* (sending) is yielded by dividing the *Packet size* upon the maximum of the *sending overhead* and *transmission time*. The similar calculation can be done for the bandwidth of *reception* by substitution of the *receiving overhead* instead of the *sending overhead*. Total data bandwidth supplied by network is defined as *aggregate bandwidth*. The fraction of aggregate bandwidth that gets delivered to the application is *effective bandwidth* or *throughput* [15],

$$\text{Eff.bandwidth} = \min\left(2 \times BW_{\text{Link Injection}}, 2 \times BW_{\text{Link Reception}}\right). \quad (2)$$

The goal of performance analysis is to determine the underlying metrics such as *sending* and *receiving overheads* and the *time of flight*. It is of particular interest to investigate the protocol stack traversal times (GIOP/TCP/IP/ETH). In Sections 4.1 and 4.2, we explain our methodology for probing the software and hardware ORB approaches, respectively.

4.1. Software ORB Probing Strategy. Figure 9 illustrates the internal mechanism for extraction of the protocol stack traversal times where two FPGA-based smart cameras are using

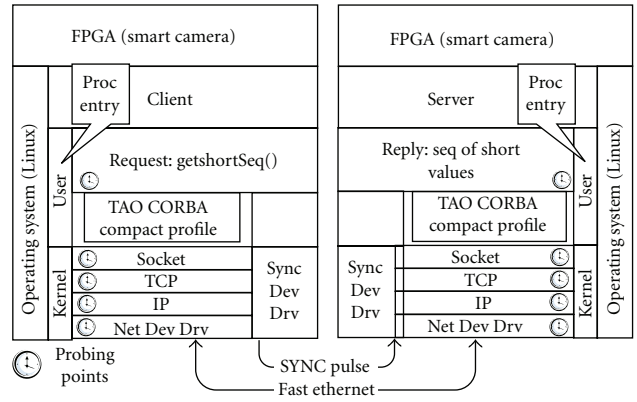

Figure 9: Software ORB experiment setup.

the software ORB. The big challenge here is how to accurately track packets traversing the Linux kernel. To measure packet traversal, an additional time stamp probing code is added to the Linux kernel code, similar to the approach in [9]. As depicted in Figure 9, a variety of probing points at several positions of the kernel's protocol stack (including network device driver, IP layer, TCP layer, and socket layer) are inserted. A unique identification (ID) code is assigned to each probing point position. Later at the postprocessing phase of

performance analysis, those ID codes are used for packet tracking across network protocol stack. The acquired time stamps relevant to each unique ID must be stored and made later accessible in the user space for further analysis. For storage and access at initialization phase, a virtual memory segment is allocated. Anytime the time stamp code inside the Kernel is called, the acquired time stamp together with ID code is written into this segment. To make the measurement results transparent in the user space, a *proc* file entry as a bridge between the kernel and user space is created. The *proc* file is a virtual file residing in the *proc* directory of Linux. This gives the possibility of reading the aforementioned kernel memory segment into the user space. Therefore, based on our definition, each line of this *proc* entry file contains an ID, which determines unique probing point and also a time stamp. Thus, after completing the test, the contents of this *proc* entry are used for further analysis. However, for time synchronization between two nodes (for measurement of *time of flight*), two particular IO pins of FPGAs on both cameras are connected by an external wire. On a certain predetermined point of transaction, client node first records a time stamp and simultaneously asserts a pulse signal to the server node to generate an interrupt there. At that point, the server records a time stamp in its relevant interrupt service routine (*ISR*). In the postprocessing analysis, on the basis of those time reference events, we reach to the same time reference in the client and server measured records. To have an access to this physical IO, another new Linux device driver on both client and server nodes is developed for handling ISR. Also, to have a seamless measurement results in the user and kernel spaces for later postprocessing; again the mentioned device driver is used. In fact, by calling the *ioctl* in the user space, some particular probing points in the kernel space are activated.

4.2. Hardware ORB Probing Strategy. The second probing strategy (Figure 10) presents a realistic interconnection of two smart camera systems with fully utilization of the hardware ORB's client and server. We perform measurements on the FPGA by monitoring some hardware probing points in the state diagram of each protocol stack layer and then starting and stopping a timer which gives us the best results. As Figure 10 shows, inside both client and server nodes, an auxiliary IP core called *measurement tool* is used. It counts precisely the number of clock cycles between the start and stop commands coming from probing points placed in several layers of TCP/IP inside the hardware ORB. Therefore, the consumed time is exactly determined. Also, for measurement of *time of flight* in the same manner like the software ORB probing, particular IO pins of FPGAs on both cameras are connected by an external wire. Once the server node sends the first bit of packet, it commands for starting of a counter inside. When the client receives this first bit, it notifies the server node to stop counting via a pulse on the wire.

5. Experiments

We mounted three video cameras apart in our lab. Two sets of our developed embedded FPGA-based smart camera systems

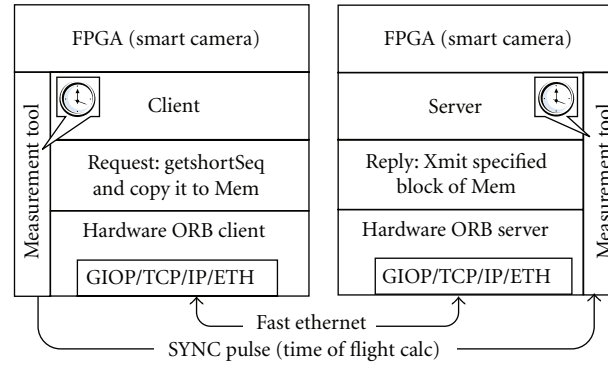

Figure 10: Hardware ORB client and server experiment setup.

Table 1: The FPGA resource usage by the SoC design.

IP Core	Flip-flop	LUT	BRAM
Segmentation	140	479	8
VGA out	73	248	
Bayer 2 RGB	64	75	2
SDI Ctrl 0	1658	1867	2
SDI Ctrl 1	1522	1731	2
Mem Ctrl	8262	7172	19
Hard ORB	3787	4265	2

(PICSy) [2] were interconnected to a desktop PC with logitech Webcam C500. PICSy features a *Virtex4 (Xilinx)* FPGA device with a *PowerPC 405* clocked at 300 MHz. OpenCV was used for implementation of computer vision algorithms. TAO [16] was applied as the middleware for exchanging the abstract information between cameras. Table 1 shows the FPGA resource usage by each IP core in the SoC design. We first program the cameras to identify multiple persons randomly walking on the site. The following sequence was conducted between the adjacent neighboring nodes. When a walking person appeared in the FOV of a camera, the camera generated a new event at each captured video frame. The payload of event was the person's extracted lower support point and the wall clock. The camera cyclically supplied its other neighboring consumer nodes with those features. It was of particular interest to measure the performance of the segmentation module on both smart camera and desktop PC. A PC with Intel Core2Duo running at 3.16 GHz needed an approximately 23 ms to compute the FG for a 640×480 sized image. In comparison to the PC, the superiority of the hardware-implemented module was achieved by a factor of ≈ 7 (≈ 3.8 ms). Furthermore, the entire procedure from the acquiring one video frame to putting the FG into the memory took ≈ 34 ms (the pixel clock of the image sensor was 48 MHz). Next, we investigated intercamera communication as follows:

(i) software ORBs running on both cameras (Figure 9);

(ii) hardware ORBs hosted on both cameras (Figure 10).

CORBA TAO client and server ran on both smart cameras having Linux with kernel 2.6.31 and TAO version 1.7.3 as

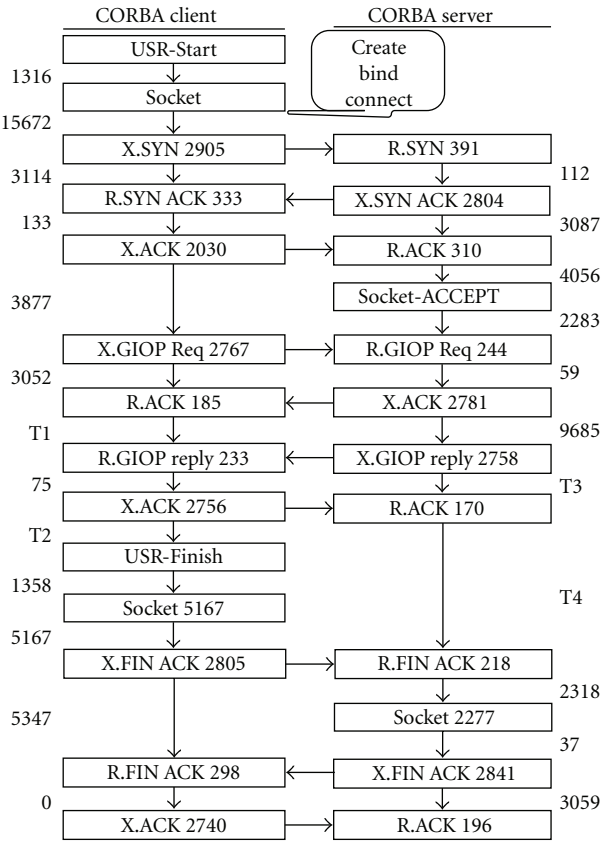

Figure 11: Measured latency values (in μs) from CORBA IIOP transaction state diagram for software ORB.

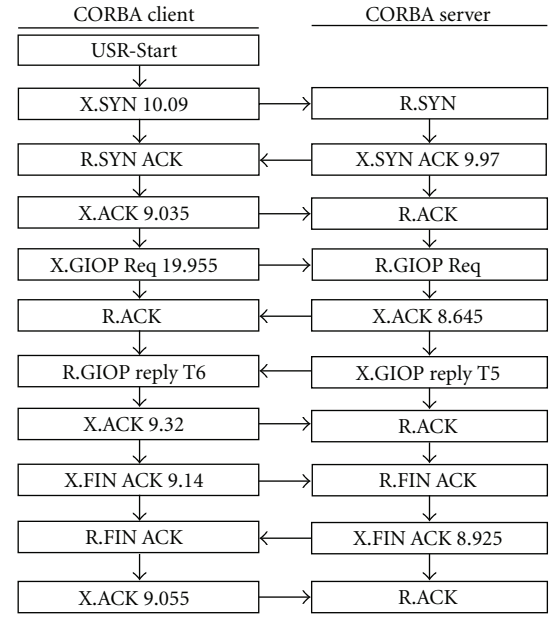

Figure 12: Measured latency values (in μs) from CORBA IIOP transaction state diagram for hardware ORB.

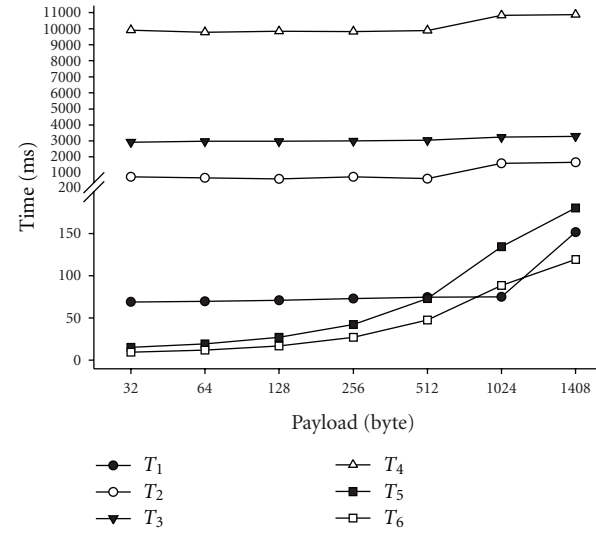

Figure 13: Timing parameters for Figures 11 and 12.

ORB. In both test scenarios, the measurements were achieved with a CORBA client invoking on a CORBA servant the operation *getshortSeq()*, returning a short sequence value as a result of operation. The network nodes were connected to each other directly without any other interfaces. In both benchmarks, the whole progress is repeated for different values of the *ShortSequence* parameter length, doubling the payload size starting from 32 bytes up to 1408 bytes. The tests were performed with the network interface speed of 100 Mbps.

Figures 11 and 12 show the transaction state diagrams in both client and server nodes based on the on-the-wire protocol of CORBA (internet inter-ORB protocol (IIOP)), corresponding to the software ORB and hardware ORB approaches.

Here, the major focus was on the demonstration of the packet processing time when traversing the protocol stack layers. In both Figures 11 and 12, *X.* stands for transmit and *R.* for receive, respectively. There is one number inside of each state, which represents measured processing time (μs) for that particular state. There is another number in Figure 11, related to each two adjacent states, which shows the waiting time for either receiving a special event from the other node or interprocessing time between two consequent states in the same node. Depending on the payload size, there are some variable timing values in both figures, called T_1, T_2, T_3, T_4, T_5, and T_6 which are shown in Figure 13, as well. Considering Figure 11, the whole transaction starts with invoking the operation *getshortSeq()* in the client node in the user space application. It corresponds to *USR-start* state. Afterwards, there are some interactions between the client and server across the TCP state diagram. Finally, *USR-finish* is the point where the application receives the request results. Figure 12 demonstrates the same story for the hardware ORB implementation. But there are some remarkable differences. First, in the software ORB approach, a very costly procedure is placed between *USR-start* state and *X.SYN* in the client side. It consists of the CORBA internal processing

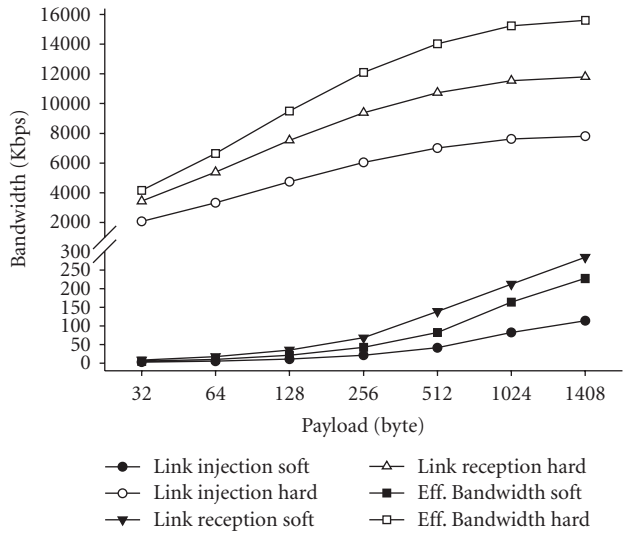

Figure 14: Comparison of latencies between software and hardware ORBs.

tasks in the user space and then coming through the socket operation in the kernel space, and thereafter, starting the TCP state diagram (*X.SYN*). According to our experiments, this segment is very expensive and nondeterministic. The second time-consuming zone is positioned between *X.ACK* and *X.GIOP reply* states in the server side. It constructs the major part of *link injection* time. Finally, the third critical point appears in the client side when delivering the received data to the user space application. On the contrary, the entire client and server transactions in the hardware ORB approach are summarized in the pure TCP state diagram, and it drastically shrinks the overheads. It is interesting to mention that the processing time for the whole transaction is 100 times faster than the software ORB. Furthermore, it should be noted that on the receiving side of each state in the hardware ORB, there are no timing values due to its pipelining nature. Collectively, considering the measurement of the *link injection* and *reception* in both benchmarks, Figure 14 shows the calculated bandwidth based on the different payloads. In terms of the software ORB, the measured time for T_1 on the client side, as a good representation of the server processing time, is used for the *link injection* parameter and summation of T_2 and T_3 for the *link reception*. In the same manner, T_5 and T_6 parameters are used for *link injection* and *link reception* in the hardware ORB approach. As it can be seen in Figure 14, the proposed hardware ORB will greatly increase the network throughput, much better than the software ORB. Additionally, a combination of the hardware ORBs client and server provides a realistic solution for interconnection between two smart cameras, resulting in huge improvement of the network latency, thus guaranteeing a real-time behavior. Judging from the data in this graph, the trend for the software ORB is continuing in the higher payload sizes. It can be easily concluded from Figure 14 that the software ORB is realizable when having higher payload size.

6. Conclusion and Future Research

Through this paper, we demonstrated the concept of a distributed person tracker system as an upper layer on top of the communication layer. The tracker leverages a geometry-based approach to establish correspondence among different views in an overlapped installed camera setup. The SoC design of an FPGA-based smart camera was presented. The design has emerged as a novel hardware/software architecture aimed at enforcing the real-time requirement of image processing applications. Experimental results reveal that the low level pixel processing hardware implemented module outperformed the PC one by the factor of 7. To provide the required communication facilities dictated by upper application layer, that is, distributed person tracker, we provided a real-time hardware/software intercommunication system among smart camera nodes, the hardware ORB which drastically shrinks the network latencies. Moreover, an exhaustive performance analysis was accomplished to accurately track packets traversing inside the Linux kernel. Very outstanding advantage of the hardware ORB in terms of speed was pointed out in favor of using gate-level implementation rather than embedded processor. The outlook of this work includes the model of the person tracker. Because of occurrence of segmentation noise and also instant arrival of group of people to scene, such simple solution is not a stable one. Considering the proposed method by Calderara et al. in [11], such kind of probabilistic models like Bayesian framework is a promising solution to increase the certainty and stability of the tracker system.

References

[1] M. Quaritsch, *A lightweight agent-oriented middleware for distributed smart cameras*, Ph.D. dissertation, Graz University of Technology, 2008.

[2] C. Bobda, A. A. Zarezadeh, F. Mühlbauer, R. Hartmann, and K. Cheng, "Reconfigurable architecture for distributed smart cameras," in *Proceedings of the Engineering of Reconfigurable Systems and Algorithms (ERSA '10)*, pp. 166–178, 2010.

[3] A. A. Zarezadeh and C. Bobda, "Hardware orb middleware for distributed smart camera systems," in *Proceedings of the Engineering of Reconfigurable Systems and Algorithms (ERSA '10)*, pp. 104–116, 2010.

[4] A. A. Zarezadeh and C. Bobda, "Performance analysis of hardware/software middleware in network of smart camera systems," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 196–201, 2010.

[5] The evolution from single to pervasive smart cameras, 2008.

[6] F. Humcke, "Making fpgas first class sca citizens," SDR Forum Technical Conference, 2006.

[7] F. Casalino, G. Middioni, and D. Paniscotti, "Experience report on the use of corba as the sole middleware solution in sca-based sdr environments," SDR Forum Technical Conference, 2008.

[8] "Data streams kernel interface," http://www.ittc.ku.edu/kusp/initial/initial_datastreams.html.

[9] J. Demter, C. Dickmann, H. Peters, N. Steinleitner, and X. Fu, "Performance analyzer of the tcp/ip stack of linux kernel 2.6.9," http://user.informatik.uni-goettingen.de/.

[10] S. Khan and M. Shah, "Consistent labeling of tracked objects in multiple cameras with overlapping fields of view," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 10, pp. 1355–1360, 2003.

[11] S. Calderara, A. Prati, and R. Cucchiara, "HECOL: homography and epipolar-based consistent labeling for outdoor park surveillance," *Computer Vision and Image Understanding*, vol. 111, no. 1, pp. 21–42, 2008.

[12] S. Velipasalar, *Peer-to-peer tracking for distributed smart cameras*, Ph.D. dissertation, Princeton University, 2007.

[13] T. Horprasert, D. Harwood, and L. S. Davis, *A Statistical Approach for Real-Time Robust Background Subtraction and Shadow Detection*, vol. 99, Citeseer, 1999.

[14] F. Mühlbauer, L. O. M. Rech, and C. Bobda, "Hardware Accelerated OpenCV on System on Chip," Reconfigurable Communication-Centric Systems-on-Chip Workshop, 2008.

[15] T. M. Pinkston and J. Duato, *Interconnection Networks—Appendix E of Computer Architecture: A Quantitative Appraoch*, Morgan Kaufmann, 4th edition, 2006.

[16] "The ace orb," http://www.cs.wustl.edu/~schmidt/TAO-status.html.