Computation and Energy Efficient Image Processing in Wireless Sensor Networks Based on Reconfigurable Computing

Tyrone Tai-On Kwok and Yu-Kwong Kwok
Department of Electrical and Electronic Engineering
The University of Hong Kong, Pokfulam Road, Hong Kong
Corresponding Author: Yu-Kwong Kwok (email: *ykwok@hku.hk*)

Abstract

In a wireless sensor network, each node is power-constrained and may need to acquire some raw data of large size (e.g., image data), on which some computation-intensive tasks (e.g., edge detection) will be done. On the other hand, in wireless communication, significant power will be consumed on transferring a sequence of data of large size. Thus, it is of high interest to carry out the sensor nodes' computation-intensive tasks efficiently while reducing the data size for wireless transfer. In this paper, we propose a new design methodology for batch processing of image data in a wireless sensor network, by employing reconfigurable computing using FPGAs.

1. Introduction

Field Programmable Gate Arrays (FPGAs) have long been used in implementing various image processing algorithms not only because of the computation-intensive nature of such algorithms, but also because of the well-matched between data type of image data and computation type of FPGAs [9]. Specifically, image data usually consist of a two-dimensional array of pixels, and FPGAs consist of a two-dimensional array of Configurable Logic Blocks (CLBs). If a set of CLBs (e.g., 4 CLBs) is responsible for handling a pixel, and each set of CLBs directly communicates with its neighbor sets, then there is a high parallelism in processing the image data, where the operations are repetitive in nature. In this paper, we are not trying to present a new FPGA implementation of image processing algorithms. Rather, we would like to present a new design methodology for batch processing of image data. We target on image processing in wireless sensor networks using an embedded reconfigurable computing system.

This research was supported by a HKU CRCG Small Project Research Grant under project number 10205130.

A typical embedded reconfigurable computing system is composed of an instruction set processor and some FPGA fabric. By utilizing massively parallel circuit design in FPGAs, computation-intensive tasks in software applications, which originally are executed in the instruction set processor, can be carried out by hardware and hence increase the overall system efficiency. Moreover, the FPGA fabric can be dynamically reconfigured for different tasks at run-time. Just like a processor switches different executables at different times, different configurations are downloaded onto an FPGA to reconfigure portions of the FPGA to implement different hardware tasks. The idea is illustrated in Figure 1. As can be seen, a complex task is partitioned into five subtasks. Any three (but not all) of the five subtasks can be implemented and executed on the FPGA chip. In order to execute the complex task, firstly the partial configurations (for configuring the corresponding portions of the FPGA) of the five subtasks are stored in some memory device. Then, according to the execution order (assumed that it is defined beforehand), different partial configurations are downloaded onto the FPGA for executing the subtasks.

Since executing computation-intensive tasks in hardware can be more energy-efficient than executing in the processor [7], the above-mentioned platform is good for a mobile sensor system which is battery-operated (i.e., having limited energy) and needs to adapt to the changing environment and user needs, so as to carry out some on-demand tasks (e.g., surveillance by taking some pictures with feature extraction). In essence, we need a reconfigurable computing system that can be operated in a multitasking manner, with the considerations of adaptability, computation efficiency and energy efficiency. With reference to the illustrative example shown in Figure 1 again, we would require the system to finish executing the five subtasks as early as possible, yet without consuming more power. In this paper, we present a hardware task scheduling algorithm for this kind of FPGA resource allocation problem.

The rest of this paper is organized as follows. In the next section, we present the related work. A sensor network

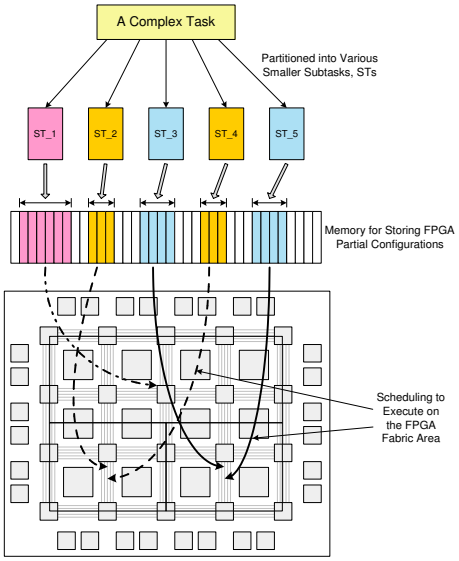


Figure 1. Dynamically partial reconfiguration.

application scenario is presented in Section 3. Section 4 describes the scheduling model. In Section 5, we present our proposed hardware task scheduling algorithm. Evaluation of the proposed algorithm is given in Section 6. Then, we talk about a prototype design of a sensor node hardware platform in Section 7. Finally, we conclude in Section 8.

2. Related Work

Figure 2 [12] shows the four commonly-used FPGA resource placement models, which define how the two-dimensional CLB (Configurable Logic Block) array of an FPGA (i.e., the FPGA fabric area) can be partitioned to implement different hardware tasks. Based on the four models, a number of hardware task scheduling algorithms have been designed [1, 4, 5, 11, 12].

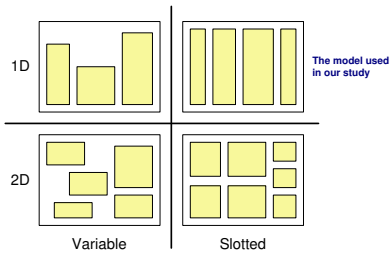


Figure 2. FPGA resource placement models.

However, most of the work in the literature considers scheduling algorithms which are unable or difficult to be implemented using the design flows in current development platform [14], such as 2D resource models [11] and merging of partitioned blocks [5]. The reason is that current development platform only allows the 1D-slotted resource model for implementing different reconfigurable modules [14]. On the other hand, little of the work takes energy consumption into consideration. Khan *et al.* [4] designed a battery-aware task scheduling algorithm for a battery-operated system. However, they partitioned the FPGA area into a set of fixed tiles, which cannot adapt efficiently to the tasks of having different FPGA area requirements. In this paper, we present the design of a hardware task scheduler that can utilize the flexibility of FPGAs, provide a balance between computation efficiency and energy efficiency, and more importantly, can be readily implemented using current design flows (based on the 1D-slotted resource model as shown in Figure 2).

3. Image Processing in Wireless Sensor Networks

3.1. Target Application Scenario

Suppose in a wireless sensor network as shown in Figure 3, the leaf nodes continuously capture some images and send to upper layers, and finally to the base station to the user side. Suppose the application is simply a pattern recognition application where the user wants to track the shape of objects in the monitoring area. If all the images captured are sent to the base station, then a large traffic volume and hence a large transmission delay will be induced, because the size of a typical image is in the order of 100KBytes. However, in a wireless sensor node, energy consumed by the communication part dominates that by the computation part, and more importantly a sensor node is energy-constrained. For example, in a wireless sensor node produced by Rockwell Inc. the energy expended in transmitting 1 bit is around 2000 times of that for executing one instruction [8, 13]. Therefore, it is beneficial to reduce the size of data sent from the leaf nodes to the base station, so as to reduce bandwidth requirement and hence energy consumption. Consequently, it is advantageous to process data locally to extract interested information and then send to the base station.

To process image data locally, we can apply reconfigurable computing using FPGAs at a layer higher than the leaf nodes, that is, the square nodes in Figure 3. Then, the images captured from the leaf nodes can be processed locally using FPGA hardware acceleration at the square nodes. In this way, from the square nodes to the base station, the traffic volume for an image is only in the order of 1KBytes or 10KBytes. This greatly reduces the bandwidth burden on the sensor network, and reduces the computation burden at the user side. In this application scenario, one major task that a square node needs to handle is—the

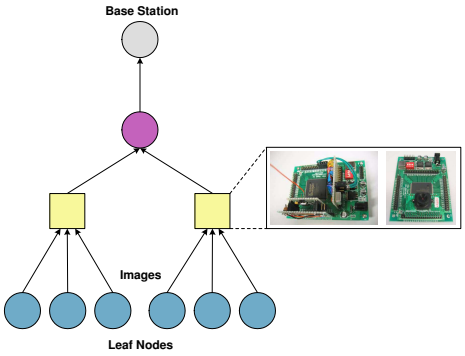


Figure 3. Application scenario.

square node receives a batch of jobs (e.g., edge detection, AES/DES encryption, etc.) from the leaf nodes and it needs to decide when and how to process the jobs, since different jobs have different execution times and need different resources. Thus, a hardware task scheduler is needed to handle the resource management problem. Figure 3 also shows the picture of a prototype design of the square sensor node, which is described in more detail in Section 7.

3.2. The Runtime System

Figure 4 depicts the block diagram of our target runtime system. Specifically, a user's application program is partitioned into two types of tasks, namely software task to be executed by the instruction set processor and hardware task to be executed by the reconfigurable device. After the application program is submitted for execution, the corresponding software and hardware tasks are diverted to the respective queues of the software task and hardware task schedulers. In our study, we focus on developing a scheduling algorithm for the hardware task scheduler. The hardware/software partitioning of an application is not the concern in this paper. The long-term goals of the runtime system are to minimize the total energy consumption and execution time of the set of hardware tasks.

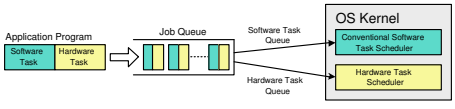


Figure 4. Runtime system.

4. Scheduling Model

4.1. Reconfigurable Architecture

In our study, we consider only reconfigurable architecture that is feasible for implementation using current technology. Figure 5 shows the target reconfigurable architecture. As can be seen from the figure, a host CPU is connected to a reconfigurable device. The host CPU is used for configuring the reconfigurable device and transferring data to/from the device. The reconfigurable device is divided into two areas, static area and dynamic area. The dynamic area implements some reconfigurable modules (RMs) for executing some hardware tasks, while the static area implements a static module which servers as a bridge between the host CPU and the RMs. The function of the static area is fixed after the whole reconfigurable device is configured. On the other hand, each RM can be partially reconfigured, without affecting the rest of the device.

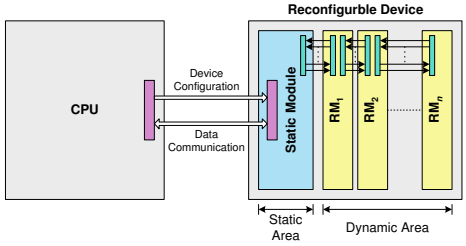


Figure 5. Reconfigurable architecture.

Logic circuits in an FPGA device are implemented in the CLBs, which are arranged as a two-dimensional array inside the FPGA. We suppose that the dynamic area comprises $W \times H$ CLBs, where H is the height of the reconfigurable device and W, the width, is variant to the size of the dynamic area. On the other hand, we assume that the size of the smallest RM is of $W_{MIN} \times H$. In practical implementation, the height of each RM spans the full height of the device [14], and W_{MIN} depends on the hardware task which requires the smallest number of CLBs. Then, the maximum number of RMs is:

$$N_{MAX} = \frac{W}{W_{MIN}} \quad (1)$$

Here, we assume that W is a multiple of W_{MIN}. Figure 6 shows some of the possible partitioning strategies of the dynamic area for the case where $N_{MAX} = 8$. As mentioned earlier, because we consider practical implementation of the reconfigurable architecture, we do not allow merging/splitting of RMs, which is the reason why the partitioning of the dynamic area needs to be explicitly defined. This also follows that, to change from one partitioning strategy to another, the whole dynamic area needs to be reconfigured.

Our view on allowing different partitioning strategies at run-time, instead of a particular one after the system starts, is that the system needs to adapt to the changing workloads, which are characterized by different computational complexity of tasks. In general, a more complex task

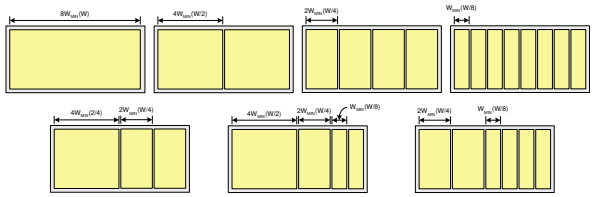


Figure 6. Some possible partitioning strategies of the dynamic area for the case where $N_{MAX} = 8$.

needs more CLBs to implement, and hence needs an RM of wider width. Analogous to a cluster computing environment where a number of parallel programs ask for different number of machines for execution, in our target reconfigurable architecture, a number of tasks ask for execution in the RMs of different widths.

On the other hand, it should be emphasized that Figure 6 only shows a particular subset of all the possible partitioning strategies for the case where $N_{MAX} = 8$. For the sake of practical implementation in an embedded system, only a set of partitioning strategies as in Figure 6 would be implemented. The reason is that it needs space in a memory device (e.g., a flash memory chip) to store the FPGA device configuration files for each of the partitioning strategy, whose size is typically in the order of 100 KBytes. In view of the above reasons, the set of partitioning strategies to be implemented should be designed to be as generic as possible, so as to cope with different workload patterns. For instance, one design criterion is that, for every hardware task, there must be a partitioning strategy implemented for the execution of the hardware task. Another criterion is that, if during a particular period of time the hardware tasks in the system only require one kind of RM to execute on, then there must be a partitioning strategy implemented such that the dynamic area can accommodate the maximum number of such RM.

4.2. Task Definition

We assume that the hardware tasks executing in the RMs are independent to each other and cannot be preempted, i.e., cannot be stopped and resumed later on the same or different RM. A hardware task T_i in our target reconfigurable architecture is characterized by two parameters: 1.) f_i^{MAX}, which specifies the maximum clock frequency, in MHz, the hardware task can be executed at; and 2.) w_i, which specifies the width of the reconfigurable module needed (it should be reminded that all the RMs share the same height).

In addition to the above two parameters, each hardware task has execution time t_i and energy consumption E_i when it is working at f_i^{MAX} MHz for a predefined amount of work. t_i can be found out by considering the amount of work to be carried out by the task. E_i can be estimated by adopting the energy model developed by Choi *et al.* [2]. Moreover, when a hardware task is scheduled to execute in the system, the task has an arrival time A_i and completion time C_i. All the above-mentioned attributes and parameters of a task T_i are stored in a vector $\mathbf{v_i}$.

In our target system, a special type of hardware task called NOP (no-operation) is defined for each RM of different widths. When an RM is configured with an NOP task, we assume that the RM will consume negligible energy. The use of NOP tasks will be elaborated in Section 5.2.

4.3. Execution Time and Energy Consumption of A Task

The following two functions are defined to extract the estimated execution time and energy consumption information, respectively, of each task T_i executing at different frequency f:

$$g^{time}(\mathbf{v_i}, f) = \frac{f_i^{MAX}}{f} \cdot t_i \quad (2)$$

$$g^{energy}(\mathbf{v_i}, f) = \frac{f}{f_i^{MAX}} \cdot E_i \quad (3)$$

When applying the energy model developed by Choi *et al.* [2], it is shown that the energy consumption is directly proportional to the working frequency, which is the reason why Equation 3 is such defined.

5. The Design of a Hardware Task Scheduling Algorithm

5.1. Design Goals

Recall that in Section 3.2 we discussed the long-term goals of the target runtime system. After we have talked about the task definition, execution time and energy consumption of a hardware task, the design goals of our proposed hardware task scheduler (to minimize the total energy consumption and total execution time of the set of hardware tasks) are formulated as follows:

$$Minimize(\sum g^{energy}(\mathbf{v_i}, f)) \quad (4)$$

$$Minimize(\max_i(C_i) - \min_i(A_i)) \quad (5)$$

5.2. The Proposed Scheduling Algorithm

5.2.1 Overview

Algorithms 1, 2, and 3 describe our proposed hardware task scheduling algorithm, namely **ECfEE** (**E**nergy-**E**fficient

and **C**omputation-**E**fficient algorithm with **f**requency adaptation). An overview of the algorithm is given as follows. The algorithm is divided into two parts. In the first part, it starts by choosing a partitioning strategy with the goal of executing a maximum number of hardware tasks ($SelectPS-T()$, Algorithm 2). Then it finds a working frequency for all the tasks selected to be executed in the chosen partitioning strategy, with the goal of minimizing energy consumption ($SelectWorkingFrequency()$, Algorithm 3). Using a working frequency as described in Algorithm 3 is the key step of **ECfEE** for reducing energy consumption, because the tasks will execute at a slower speed. Moreover, as we will show later, it will also shorten the total completion time of the set of tasks.

In the second part, it mainly deals with the case when a task finishes execution. In this case, if not all the tasks have finished execution, the scheduler needs to either find a new task to execute on the unoccupied RM or schedule an NOP task on the RM; otherwise, the reconfigurable device will be reconfigured to execute another set of new tasks.

5.2.2 Details of The Proposed Scheduling Algorithm

By referring to Figure 4, our algorithm works on a hardware task queue, T_{queue}, whose size is changed dynamically, i.e., hardware tasks are continuously being injected into the queue. We assume that the queue is implemented using some data structure such as a linked list, and hence the algorithm takes as input the pointer pointing to the head of the queue, T_{queue_head}. Our algorithm considers the first q_{range} hardware tasks when it carries out the scheduling procedure. To prevent some tasks from starving, we introduce another parameter, s_i, to each task T_i. s_i is set to zero initially, and it is incremented by one, if $T_i \in T_{queue}[1..q_{range}]$ was not scheduled during a round of choosing candidate tasks for scheduling, i.e., Steps 1–6 of Algorithm 2.

In Step 3 of Algorithm 1, F denotes the set of frequencies that *all* the RMs in the reconfigurable device may be clocked at. All RMs sharing the same working frequency is a consideration for practical implementation, as recommended by [14]. Moreover, the fact that the available frequencies are stepped by 5 is also a practical consideration. The primary reason is to reduce the memory space required for storing the configuration files of the reconfigurable device. In $SelectWorkingFrequency()$ of Algorithm 3, we choose a working frequency for the scheduled tasks such that all the tasks can finish execution all together, with the minimal average difference of execution time. The reason for doing so is that it can reduce the amount of executing NOP tasks in Steps 14–25 of Algorithm 1. Since the execution of NOP tasks (instead of normal hardware tasks), is considered a waste of resources, by reducing the amount of executing NOP tasks, it can better use the resources of RMs, and hence better use the energy of the system.

In Steps 14–25 of Algorithm 1, after a task has finished execution, we do not choose any task that can fill the RM. There are two reasons for that. Firstly, we want to avoid the loop of scheduling NOP task when there is no task suitable for execution under the current partitioning strategy but we need to wait for the executing tasks to finish. Secondly, after all the tasks under the current partitioning strategy finish their execution, the system can consider another partitioning strategy so as to schedule the maximum possible number of tasks for execution.

The dominating computation of the proposed scheduling algorithm **ECfEE** is in the $SelectPS-T()$ function, where for each partitioning strategy, q_{range} tasks have to be looked up so as to choose a suitable partitioning strategy. Thus, the complexity of **ECfEE** is $O(p \cdot q_{range})$ where p is the maximum number of partitions in the dynamic area.

Algorithm 1 ECfEE

Schedule(T_{queue_head})

1: $q_{range} \leftarrow \alpha \cdot N_{MAX}$ /* α is a predefined constant */
2: $s_{threshold} \leftarrow \beta \cdot N_{MAX}$ /* β is a predefined constant */
3: $F \leftarrow \{f_{MIN}, f_{MIN}+5, f_{MIN}+10, ..., f_{MAX}\}$
4: $P \leftarrow GeneratePartitioningStrategies(N_{MAX})$
5: $p_{current} \leftarrow null$ /* current partitioning strategy used */
6: $T_{scheduled} \leftarrow null$ /* set of scheduled tasks */
7: $T_{not\_scheduled} \leftarrow null$ /* considered but not scheduled */
8: $f_{working} \leftarrow null$ /* selected working frequency */
9: **while** (TRUE) **do**
10: $(p_{current}, T_{scheduled}, T_{not\_scheduled}) \leftarrow SelectPS-T(T_{queue\_head}, q_{range}, s_{threshold}, P, T_{not\_scheduled})$
11: $f_{working} \leftarrow SelectWorkingFrequency(T_{scheduled}, F)$
12: Reconfigure the dynamic area for execution of tasks.
13: **while** (not all the tasks have finished execution) **do**
14: **if** a task has finished execution in RM_j **then**
15: **if** there is some other task still executing **then**
16: **if** $\exists$ task $T_i \in T_{queue}[1..q_{range}]$ and T_i will finish execution no later than any current executing task **then**
17: Schedule T_i to execute in RM_j.
18: **if** $T_i \in T_{not\_scheduled}$ **then**
19: $T_{not\_scheduled} \leftarrow T_{not\_scheduled} - T_i$
20: **end if**
21: **else**
22: Schedule NOP task to execute in RM_j.
23: **end if**
24: **end if**
25: **end if**
26: **end while**
27: **end while**

6. Simulation Results

6.1. Simulation Setting

Because currently there is no benchmark package for a reconfigurable system, similar to [5, 11], the performance of the proposed scheduling algorithm is studied through simulation. The simulation program is written in C. We con-

Algorithm 2 ECfEE—Selection of Partitioning Strategy and Tasks for Scheduling

SelectPS-T($T_{queue\_head}, q_{range}, s_{threshold}, P, T_{not\_scheduled}$)

1: **if** $\exists\ s_i \geq s_{threshold}$ and $T_i \in T_{not\_scheduled}$ **then**
2: $s_j \leftarrow \max(s_i | T_i \in T_{not\_scheduled})$
3: Select $p_{current} \in P$ which allows the task having s_j and the maximum number of other tasks $\in T_{queue}[1..q_{range}]$ to execute in the RMs.
4: **else**
5: Select $p_{current} \in P$ which can accommodate the maximum number of tasks within $T_{queue}[1..q_{range}]$.
6: **end if**
7: Denote the set of tasks which contributes to the selection of $p_{current}$ as $T_{scheduled}$.
8: $T_{not\_scheduled} \leftarrow T_{not\_scheduled} \bigcup (T_{queue}[1..q_{range}] - T_{scheduled})$
9: $T_{queue} \leftarrow T_{queue} - T_{scheduled}$
10: $s_j \leftarrow s_j + 1$, where $T_j \in T_{not\_scheduled}$

Algorithm 3 ECfEE—Selection of Working Frequency for the Tasks Selected for Scheduling

SelectWorkingFrequency($T_{scheduled}, F$)

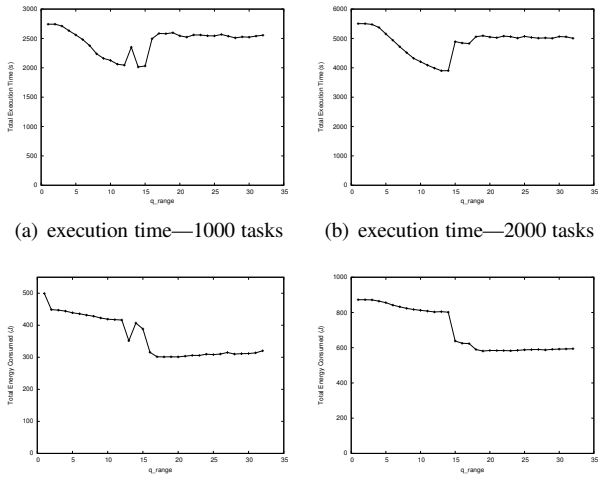
1: $f_{scheduled}^{MAX} \leftarrow \min(f_i^{MAX} | T_i \in T_{scheduled})$
2: For each $f \in F$ and $f_{MIN} \leq f \leq f_{scheduled}^{MAX}$, calculate:
- $G_{average}^{time} = \frac{1}{|T_{scheduled}|} \cdot \sum g^{time}(\mathbf{v_i}, f)$, where $T_i \in T_{scheduled}$
- $\sum (G_{average}^{time} - g^{time}(\mathbf{v_i}, f))^2$, where $T_i \in T_{scheduled}$

3: Denote the f which gives the smallest value of $\sum (G_{average}^{time} - g^{time}(\mathbf{v_i}, f))^2$ in the previous step as $f_{working}$.

struct T_{queue} by randomly generating a set of tasks of the following parameters: 1.) $N_{MAX} = 8$ and $W_{MIN} = 1$, and task width $w_i \in [1, 2, 4, 8]$; 2.) $F_{MIN} = 20$ and $F_{MAX} = 50$ such that $f_i^{MAX} \in [20..50]$ and $f_{working} \in [20, 25, 30, 35, 40, 45, 50]$; 3.) $t_i \in [500, 5000]$ ms; and 4.) $E_i \in [100, 1200]$ mJ, which is proportional to w_i.

6.2. Tradeoff between Execution Time and Energy Consumption

In our simulations, we set $\beta = 2$, which means that $s_{threshold} = 16$. Figures 7(a) and 7(c) show, respectively, the total execution time and total energy consumed in executing 1000 randomly generated tasks. As can be seen from the figures, for q_{range} smaller than 15, the total execution time and total energy consumed decrease with q_{range}. The reason for this is that by using a larger q_{range}, the scheduler can choose a better partitioning strategy to schedule more tasks for better utilization of the RMs. This also means that the tasks are likely to execute at a frequency lower than f_i^{MAX} and hence energy is saved. On the other hand, for q_{range} larger than 15, an increase in q_{range} causes the total execution time to increase and the total energy consumed to decrease. The cause of this is due to the fact that by further increasing q_{range}, the scheduler can further schedule more tasks in a partitioning strategy. However, by considering more tasks, the $f_{scheduled}^{MAX}$ chosen ($SelectWorkingFrequency()$, Algorithm 3) will be of a lower value. As a result, the $f_{working}$ chosen will also be of a lower value, causing the tasks take more time to finish, but save more energy. Thus, from Figures 7(a) and 7(c), we can see that $q_{range} = 15$ is a good tradeoff between the total execution time and total energy consumed. The same value of $q_{range} = 15$ is observed in the execution of 2000 tasks, as shown in Figures 7(b) and 7(d).



(a) execution time—1000 tasks (b) execution time—2000 tasks

(c) energy consumed—1000 tasks (d) energy consumed—2000 tasks

Figure 7. Total execution time and energy consumption against different values of q_{range} in ECfEE.

6.3. Comparison with An Existing Algorithm

The problem of placing reconfigurable modules onto the dynamic area of the 1D-slotted model as shown in Figure 2 is similar to the 1D bin-packing problem. Best-Fit and First-Fit are two well-known online algorithms for the 1D bin-packing problem, and they have been considered for hardware task placement [1]. In our study, we have adapted a variant of Best-Fit to compare the performance with our proposed scheduling algorithm. We name this algorithm **BF** and the algorithm is outlined in Algorithm 4. On the other hand, it should be noted that **BF** is effectively **ECfEE** with $q_{range} = 1$. To choose a suitable partitioning strategy, **BF** only considers the head task of the task queue for each partitioning strategy. Thus, the complexity of **BF** is $O(p)$ where p is the maximum number of partitions in the dynamic area.

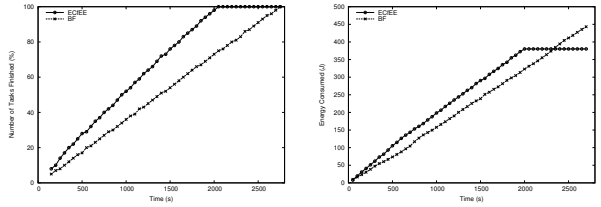
In Figure 8, we compare our proposed scheduling algorithm **ECfEE** with **BF**. For the comparison, 1000 tasks are

Algorithm 4 BF

Schedule(T_{queue_head})

1: $F \leftarrow \{f_{MIN}, f_{MIN}+5, f_{MIN}+10, ..., f_{MAX}\}$
2: $P = GeneratePartitioningStrategies(N_{MAX})$
3: **while** (TRUE) **do**
4: Select $p_{current} \in P$ where the partitioning strategy $p_{current}$ allows the head task of T_{queue} to execute and contains the maximum number of RMs.
5: With reference to the f_i^{MAX} of the head task, select the largest possible $f_{working} \in F$.
6: Schedule the head task to execute and remove it from T_{queue}.
7: For any free RM in current partitioning strategy, if the head task of T_{queue} can execute on it, schedule the head task to execute and remove the task from T_{queue}. This step is repeated until no suitable task is found.
8: Run Steps 4–7 when a task finishes execution.
9: **end while**

executed. From the figures, we can see that, when compared to **BF**, **ECfEE** can significantly reduce the energy consumption and execution time. Specifically, **ECfEE** can reduce execution time and energy consumption by 26% and 14%, respectively. From these results, it is interesting to see that by exploiting adaptive working frequency of hardware tasks, we can not only reduce the total energy consumption, but also reduce the total execution time of the tasks.



(a) task completion (b) energy consumed

Figure 8. Performance comparison between ECfEE and BF.

7. Prototype Design of the Square Sensor Node

7.1. Hardware Platform Design

Figure 9 shows the block diagram of a prototype design of the square sensor node shown in Figure 3. The prototype consists of two FPGAs—the master FPGA is for implementing a MicroBlaze based μClinux system [6] while the slave FPGA is for carrying out some hardware tasks (note that a leaf node consists of only the master FPGA with a camera attached to it). A camera is attached to the slave FPGA so that the square sensor node can also take the job of environmental monitoring. Moreover, some SDRAM memory is attached to the slave FPGA for storing image data so that the image can be processed quickly. In the system, there is an RFM DR3100-1 433.92 MHz/115.2 kbps transceiver module [10], which enables the square sensor node to talk to other nodes, on which there is also a transceiver module. On the other hand, the camera used in our study is a COMedia C3038 camera module [3], which can capture a color image of up to 356×292 pixels.

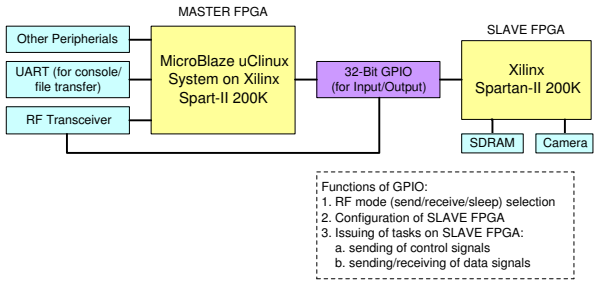


Figure 9. The block diagram of a square sensor node.

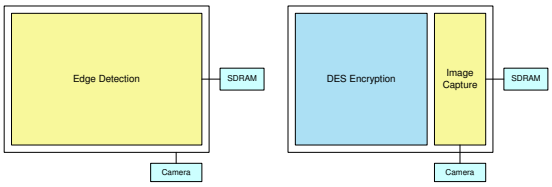
7.2. The Application

Our target application in the wireless sensor network is edge detection, a well-known image processing technique. The edge detection process is a bottleneck in the application scenario, and therefore the process is worth being carried out in an FPGA so as to increase the system performance. The square sensor node itself will constantly capture some images to perform edge detection. In addition, it will also receive images from leaf nodes to perform edge detection. The resulted images after edge detection will then be sent to upper layers and finally to the base station. To enhance data security, the resulted images after edge detection can be encrypted using the DES encryption algorithm before they are sent out. Thus, the following hardware tasks are defined for the square sensor node: 1.) capture an image and save it to the SDRAM; 2.) perform edge detection; and 3.) perform DES encryption.

The partitioning of the slave FPGA for executing the above three hardware tasks is shown in Figure 10. Despite that the chosen FPGA device can only allow us to execute a quite limited number of hardware tasks at the same time, which does not allow the full utilization of the proposed scheduler, we would like to demonstrate the potential of using a such scheduler.

7.3. Implementation Results

Using the TIFF format and a resolution of 356×292 pixels, a captured image is of size 310KBytes while the



(a) one partition (b) two partitions

Figure 10. Partitioning of the slave FPGA for executing different hardware tasks.

resulted edge detection image is of size 57KBytes. There is about 82% reduction in file transfer size when we transmit the resulted edge detection image instead of the original image, which can greatly reduce the bandwidth and energy consumption.

When executing the above-mentioned tasks using the slave FPGA (instead of using the 48 MHz MicroBalze microprocessor), on which the DES engine runs at 33 MHz while the edge detection algorithm runs at 20 MHz, the speedup factors for image capture, DES encryption and edge detection are 60, 40 and 22, respectively. The speedup factor is defined as follows:

$$Speedup = \frac{execution\_time(MicroBlaze)}{execution\_time(FPGA)} \quad (6)$$

We have also compared the execution of the above three hardware tasks in an execution environment with and without multitasking support. Without multitasking support, the three tasks are executed alternately in the FPGA. With multitasking support, DES encryption and image capture are performed concurrently in the FPGA. It is found that, with multitasking support, the total execution time of the three tasks is reduced by about 15%.

8. Conclusions

In this paper, we have presented the design of a hardware task scheduler in an embedded reconfigurable computing platform for batch processing of image data in a wireless sensor network. We have also presented a prototype design of a sensor node which employs reconfigurable computing for batch processing of image data. Implementation results show performance gain in adopting such system design.

References

[1] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, Jan. 2000.

[2] S. Choi, J.-W. Jang, S. Mohanty, and V. K. Prasanna, "Domain-Specific Modeling for Rapid System-Wide Energy Estimation of Reconfigurable Architectures," *Proc. Engineering of Reconfigurable Systems and Algorithms (ERSA'02)*, June 2002.

[3] COMedia C3038 Camera Module Datasheet, `http://home.pacific.net.hk/comedia/c3038.pdf`, 2006.

[4] J. Khan and R. Vemuri, "An Efficient Battery-Aware Task Scheduling Methodlogy for Portable RC Platforms," *Proc. Field Programmable Logic and Applications (FPL'04)*, pp. 669–678, Sept. 2004.

[5] B. Krishnamoorthy, J. G. Wu, and T. Srikanthan, "Hardware Partitioning Algorithm for Reconfigurable Operating System in Embedded Systems," *Proc. Sixth Real-Time Linux Workshop*, pp. 117–123, Nov. 2004.

[6] MB μClinux, `http://www.itee.uq.edu.au/~jwilliams/mblaze-uclinux/`, 2006.

[7] J. M. Rabaey, "Silicon Platforms for the Next Generation Wireless Systems - What Role Does Reconfigurable Hardware Play?" *Proc. Field-Programmable Logic and Applications 2000*, pp. 277–285, Sept. 2000.

[8] V. Raghunathan, S. Ganeriwal, and M. Srivastava, "Energy Efficient Wireless Packet Scheduling and Fair Queuing," *ACM Trans. Embedded Computing Systems*, vol. 3, no. 1, pp. 3–23, Feb. 2004.

[9] N. K. Ratha and A. K. Jain, "Computer Vision Algorithms on Reconfigurable Logic Arrays," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 1, pp. 29–43, Jan. 1999.

[10] RF Monolithics Inc., `http://www.rfm.com/`, 2006.

[11] C. Steiger, H. Walder, and M. Platzner, "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1393–1407, Nov. 2004.

[12] H. Walder, "Operating System Design for Partially Reconfigurable Logic Devices," PhD Thesis, Swiss Federal Institute of Technology (ETH), Apr. 2005.

[13] A. Woo and D. E. Culler, "A Transmission Control Scheme for Media Access in Sensor Networks," *Proc. MOBICOM 2001*, pp. 221–235, July 2001.

[14] Xilinx Inc., "Xilinx Application Note XAPP290: Two Flows for Partial Reconfiguration: Module-Based or Difference-Based," v1.2 edition, Sept. 2004.