Real-Time Action Scheduling in Pervasive Computing

Wenwei Xue
Nokia Research Center, Beijing, China
E-mail: wayne.xue@nokia.com

Qiong Luo and Lionel M. Ni
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong, China
E-mail: {luo, ni}@cse.ust.hk

Keywords: pervasive computing, real-time action scheduling, query processing, device eligibility

Received: November 23, 2010

Pervasive computing applications, such as video surveillance and robot control, involve diversified operations on physical devices. We call a sequence of operations on a device an action and study how to schedule real-time actions on the devices in pervasive computing. We identify a number of novel characteristics of this pervasive action scheduling problem and develop a dynamic, heuristic algorithm for the problem. The algorithm performs priority-based action scheduling whenever some device becomes free and does not reply on any system-defined scheduling interval. We have implemented our proposed action scheduling algorithm in a pervasive query processing system named Aorta and evaluated its performance using actions in a pervasive lab monitoring application. Our simulation results demonstrate the algorithm ensures small dropping rate of actions and has tiny computation cost.

Povzetek: Opisan je izvirni sistem Aorta, kjer je akcija opisana kot povpraševalni operator.

1 Introduction

In pervasive computing [27], many types of devices are embedded in the physical world and execute real-time actions for the applications [22][24]. Example devices in pervasive computing include sensor nodes, network cameras, programmable robots, and handheld devices such as PDAs and phones. Here we define an *action* as a predefined sequence of operations to be executed on a device that is encapsulated in a user-defined function [1][8]. Due to the real-time requirement of action executions, action scheduling on the devices among multiple applications is a crucial problem in pervasive computing. For instance, in a lab surveillance scenario a number of concurrent applications may require the cameras and robots deployed in the lab to take photos or perform tasks at different locations from time to time. A photo taken or a task performed will become obsolete and useless if it cannot be scheduled and executed timely on some device of the corresponding type. In this paper, we address this action scheduling problem in the framework of a pervasive query processing system *Aorta* that we have developed [29][30].

The problem of job scheduling on parallel machines [16] and its variants [3][5][11][13][17][18][19][20][23][25][26] have been widely investigated in the literature. The action scheduling problem we study can be regarded as a new variant of this classic problem in the pervasive computing scenario. This is because an action execution for an application is often not fixed to a specific device but can be performed on any device that satisfies certain condition. Take the lab surveillance scenario as example again. Whenever a sensor node installed in the lab detects abnormally high *noise* readings lasting a period, which are likely caused by the loud conversation of people around the node, a robot is automatically controlled to move to the location of the node and issue a warning to ensure the quiet working environment in the lab. Every robot in the lab is a candidate device for this action execution and it is sufficient to operate one but not all of them to perform the task.

Our action scheduling problem inherits several characteristics of the classic parallel machine scheduling problem, including unrelated devices, device eligibility restrictions and deadlines of non-preemptive action executions [16]. In addition, our problem has a unique characteristic that is the interaction between actions and devices [29] in pervasive computing. More specifically, the actions often change the physical status of the devices that execute them. Such change in turn affects the future executions of succeeding actions on the devices. The physical status of a device is represented in Aorta as the current values of a set of *status attributes* defined in the virtual table for the type of device [30]. Example status attributes in different virtual device tables are *voltage*, *freeRAM* for *sensors* and *phones*; *pan*, *tilt*, *zoom* for *cameras*; *loc*, *angle* for *robots*.

```
CREATE AQ noise_rejection AS
    SELECT      warn(r.id, s.loc, "messages/warning.txt")
    FROM        sensors s, robots r
    WHERE       600 < (SELECT   winavg(ss.noise, 5, 5, minute)
                       FROM     sensors ss
                       WHERE    every(10, second) AND ss.id = s.id)
    DEADLINE  30 seconds
```

Figure 1: The *noise_rejection* query for lab surveillance.

Figure 2: Devices in the pervasive lab.

The scheduling model we face in our problem is dynamic rather than the static model adopted in the classic problem. The action executions to be scheduled on the devices are dynamically arriving at the system over time. In contrast, the classic problem takes a static set of jobs as input and assumes all kinds of job information, e.g., the start or processing times of the jobs, are known a priori before the scheduling [16].

We summarize all these characteristics of our action scheduling problem and propose a dynamic, heuristic algorithm to solve the problem. Whenever a device becomes free, the algorithm selects an action request queued in the system that has the highest priority to be serviced on the device. We define an *action request* as the request for an action execution from an application with instantiated values for the input parameters of the action. The priority of an action request on a device is computed using the response time of the request on the device, the deadline and the candidate device number of the request, as well as the current eligibility and reliability of the device. We have implemented the algorithm in our Aorta prototype and seamlessly integrated it with other mechanisms in the system [30].

The effectiveness of declarative queries to task networks of devices has been illustrated by lots of recent work in both database [12][31] and networking [10][21] communities. Following this programming paradigm, Aorta uses SQL-based continuous queries having actions embedded [29] to express the processing logics of pervasive computing applications. We call these queries *action-embedded queries*. With this abstraction for applications, the process of action scheduling in Aorta is encapsulated into the adaptive group optimization of multiple concurrent queries running in the system. Although in this paper we present and evaluate our action scheduling algorithm based on these system implementation details of Aorta, the algorithm is generic and is indifferent to the particular application interface.

We have designed the syntax and semantics of action-embedded queries to accord with the requirements of action scheduling. An optional DEADLINE clause is provided in Aorta's query interface for applications to tell the system the *deadline* of an action request from a query, which is defined as the interval between the time when the request is issued and the time when the request is serviced on a device (i.e., the action execution has been finished). Moreover, when the WHERE clause of an action-embedded query is evaluated as *true* and a set of candidate devices is determined for the action request, the request will be scheduled and serviced only once on a selected device among these candidates [29]. As an example, Figure 1 shows a *noise_rejection* query in Aorta that abstracts the robot patrol application for lab surveillance we have described previously.

We have built a case study application on our Aorta prototype to monitor the pervasive research lab in our department. The lab is equipped with desktops having removable hard disks, notebooks, and various types of devices including Crossbow motes [4], AXIS 2130 network cameras [2], ER1 personal robots [6], PDAs and phones (Figure 2). This pervasive lab monitoring application is used as an illustrative example throughout the paper as well as in our performance evaluation of the proposed action scheduling algorithm.

The remainder of this paper is organized as follows. We describe the Aorta system model for action scheduling in Section 2. We identify the characteristics of our action scheduling problem in Section 3 and present a dynamic, heuristic algorithm for the problem in Section 4. In Section 5, we perform simulation studies to evaluate the effectiveness of our proposed scheduling algorithm using actions in the pervasive lab monitoring application.

We discuss related work in Section 6 and conclude the paper in Section 7.

2 System model for action scheduling

In this section, we describe the model we implement in the Aorta system to effectively support the action scheduling on devices.

2.1 Actions and action operators

Aorta only supports actions that operate a single device. We focus our study on single- device actions due to three main reasons: (i) they are prevalent in real-world applications [1][8][22][24], (ii) they are more practical and manageable in implementation, and (iii) in combination with action or query nesting, they can be used to compose many multi-device actions that have simple communication logics between devices [29].

For every action in Aorta, we require the identity of the device that the action is executed on to be not fixed in the function code block. In contrast, the device should be explicitly or implicitly identified by the instantiated parameter values for the action at run time. As a typical example, the first input parameter of the *warn* action in Figure 1 determines the robot on which a specific execution of the action will be executed. The necessity of this restriction stems from the "black box" nature of actions. Being a UDF, an action is registered to Aorta as a compiled code block and it is impossible for the system to modify its implementation details. Consequently, if the identity of the device to execute an action is fixed in the code block, there is little room for action scheduling on the parallel devices. In this case, our problem degenerates to a single-machine scheduling problem [16] on individual devices.

Aorta makes an action embedded in a query a first-class operator in the evaluation plan of the query. An action operator contains the following information about an action: (i) the name, (ii) the specifications of input parameters, (iii) the pointer to the function code block to be invoked. Furthermore, all queries having an action on the same type of device share a single action operator among their query plans. Every query plan is connected to the shared action operator via a common input queue of the operator. The action operator maintains corresponding information about the action embedded in each query so that it can use the correct information to schedule a specific execution of an action for a query.

An action operator is created when the first query embedded with an action on the type of device is registered to Aorta. Subsequent queries having actions on the same type of device are connected to the operator by an update of the information maintained in the operator. These shared action operators give the Aorta query optimizer a global view of the current action workload on individual types of devices in the system. Rather than being optimized separately without coordination, multiple queries are grouped and the action executions for them are adaptively scheduled as a whole.

2.2 Scheduling model

Figure 3 depicts the scheduling model for every action operator in Aorta's query processing framework. In the figure, d_j ($1 \le j \le m$) denotes all devices of a type involved in the Aorta system that the operator is in charge of action scheduling on, e.g., the set of programmable robots. q_i ($1 \le i \le n$) denotes the plan of Query i and R_i the streaming action requests issued from the query over time. R denotes the whole stream of action requests that arrive at the input queue of the action operator and $R = \bigcup_{i=1}^{n} R_i$.

Being the main component of the operator, the scheduler implements the dynamic and heuristic action scheduling algorithm we have developed. a_i ($1 \le i \le n$) in the operator denotes the stored specification information about the action that is embedded in Query i.

3 Characteristics of action scheduling

The action scheduling problem we study has a unique set of characteristics that is tightly related to the application scenario of pervasive computing. We identify all characteristics of the problem one by one as follows.

(1) Action-device interaction. There is a special kind of interaction between actions and devices in pervasive computing: an action execution on a device may change the physical status of the device; in turn, the physical status of a device may affect the cost of an action execution on the device. This interaction is generic to several cost metrics for actions in pervasive computing, such as the response time, the power consumption and the price of service. It makes our scheduling of actions more complex than traditional job scheduling, because the costs of an action execution on candidate devices are different and dynamically changing.

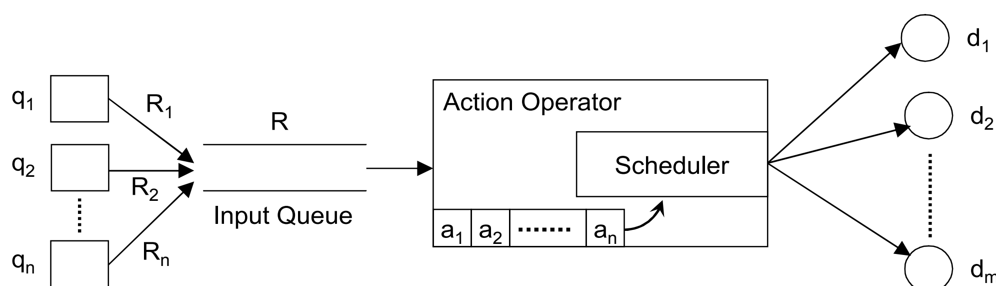


Figure 3: Scheduling model for an action operator in Aorta.

```
CREATE AQ snapshot AS
    SELECT     photo(c.ip, s.loc, "photos/admin")
    FROM       sensors s, cameras c
    WHERE      s.accel_x > 500 AND coverage(c.loc, s.loc)
```

Figure 4: The *snapshot* query in the pervasive lab monitoring application.

To illustrate the interaction, two actions in the pervasive lab monitoring application are listed in Examples 1 and 2. The device physical status related to the action is the location of a robot or the head position of a camera, respectively.

Example 1. Consider the *warn*(*id*, *location*, *text_file*) action Figure 1 on programmable robots [6]. The action operates a robot with *id* to rotate towards and go straight to a target *location*, and play a warning message whose content is specified in *text_file* when arriving at the location. An execution of the action changes the location of the robot. The response time of the execution is proportional to the distance between the target location and the current location of the robot.■

Example 2. Consider the *photo*(*ip*, *location*, *directory*) action in Fig on PTZ network cameras [2]. The action operates a camera with *ip* to move its head to a position pointing at *location* and take a photo. The action then stores the photo that the camera takes to *directory*. An execution of the action changes the position of the camera head (i.e., the pan, tilt, zoom values). The response time of the action execution depends on the current head position of the camera.■

(2) Dynamic request arrival in a global queue. Action requests from multiple queries continually and dynamically arrive at the single input queue of an action operator over time. There is no local request queue for a device. The system has no prior knowledge about the arrival time, deadline or candidate devices of each request.

(3) Deadlines of requests. Action executions are of little use for pervasive computing applications if they cannot be finished in a timely manner. An unscheduled action request should be dropped when the system detects that the request cannot be serviced within its deadline.

(4) Independent and non-preemptive requests. There is no message communication between any two action requests as they represent separate executions of actions. Since the execution flows of actions are encapsulated in code blocks and are unknown to the system, an action execution on a device cannot be interrupted and multiple executions cannot be interleaved on a device. As a result, no communication is required in the scheduling to transplant a partially-serviced action request from one device to another.

(5) Unrelated devices. The cost of an action request on a device is generally not related to those costs of the request on the other devices. In other words, the devices in our action scheduling model are unrelated [16]. Each device services the action requests scheduled on it individually.

(6) Device eligibility. The candidate devices for an action request often include a subset of all devices of the type in the system. For instance, for the *snapshot* query in Figure 4, the set of candidate devices for a request is determined by the function *coverage*(*loc*1, *loc*2) in the query condition. The function returns *true* if and only if the view range of the camera with location *loc*1 covers the location *loc*2. This example also indicates the fact that the set of candidate devices for multiple action requests of the same query may be different. We say that a device is *eligible* for an action request if it is a candidate for servicing the request.

The last four characteristics of our problem can be mapped to the following characteristics of the classic parallel machine scheduling problem [16] in order: (i) deadlines of jobs, (ii) non-preemptive jobs, (iii) unrelated machines, and (iv) machine eligibility restrictions for jobs. Moreover, the first characteristic of our problem is similar to the sequence-dependent setup time of jobs in the classic problem [16]. The major difference is that we are facing sequence-dependent response time (in general, cost) of action requests rather than setup time. To the best of our knowledge, there is no existing scheduling algorithm for the classic problem or for any variant of it whose design has taken all the characteristics (i)-(iv) and the sequence-dependent setup time into consideration.

The second and third characteristics of our problem require us to adopt a dynamic model [11][17] for action scheduling rather than the static model in the classic problem [3][16]. In our previous work, we have developed two non-real time algorithms for action scheduling in Aorta without considering the request deadlines [29] [30]. The algorithms are based on a static scheduling model that divides the system time into a sequence of equal-length scheduling intervals and schedules action requests arriving in each interval individually. However, in our subsequent real testbed evaluation of Aorta, we found that these prior algorithms have a major drawback when applying to the scheduling of actions with deadlines, that is, their performance in practice largely depends on the length of the system-defined scheduling interval. If the interval is long, requests that arrive earlier in an interval suffers from a large delay and their deadlines are more likely to be missed, whereas very few requests can be scheduled together in each interval and the static group scheduling becomes less effective if the interval is short.

With a dynamic scheduling model, it has been proved that if we do not assume any prior knowledge about the arrival times of the continuously-arriving jobs, an opti-

mal algorithm does not theoretically exist for a job scheduling problem [5]. In comparison, a static model makes the design of an optimal scheduling possible as all kinds of information about the jobs in a scheduling interval is available before the scheduling starts. Nevertheless, such static scheduling problem is NP-hard and too computationally expensive to be feasible in our real-time scenario [30]. Even a sub-optimal, non-heuristic solution for the problem, such as the Simulated Annealing (SA) algorithm we have studied before [30], requires large computation cost given a small input size.

As a summary, the unique set of characteristics of our action scheduling problem in pervasive computing makes scheduling algorithms in the literature based on a static model inapplicable to the problem, due to their significant running time or unconcern for a few characteristics. These negative observations, as well as the effectiveness of our prior static heuristic algorithms for non-real time action scheduling [30], motivate us to propose a new dynamic, heuristic algorithm for the real-time action scheduling in Aorta.

4 Heuristic scheduling algorithm

We present the detailed design of our heuristic algorithm for real-time action scheduling in this section. Algorithm 1 formulates the input and output of the problem and depicts the flow of our proposed algorithm. The algorithm is called by an action operator immediately after the operator is generated. We developed the algorithm based on the *List Scheduling* (LS) discipline in scheduling theory [16] due to the tiny algorithmic running time incurred by the discipline. Whenever a device becomes free, the LS discipline schedules a request in the queue that the device is eligible for on the device using a heuristic.

Algorithm 1 starts by initializing the status information about all devices involved in the scheduling (Lines 1-3). Such information is dynamically maintained during the execution flow of the algorithm (Lines 10, 19). The algorithm then enters an endless scheduling loop (Line 4) and performs action scheduling on the devices round by round. The loop stops only when the system is terminated and the action operator is destroyed.

In each round of the scheduling loop, Algorithm 1 first examines all requests in the input queue of the action operator and removes those requests whose deadlines have been missed at this time (Lines 5-6). Next, for each device that is currently free, the algorithm computes the priority (*PRI* value) of every request in the queue that the device is eligible for using Function *computePriority* (Line 14). Function *estimateCost*(r_i, d_j, M) estimates the current cost of request r_i on device d_j based on a cost model M (Line 12). The algorithm then selects the request-device pair (r_s, d_s) having the highest priority (Lines 15-16) and schedules request r_s on device d_s in this round (Line 18). Note that we regard a request-device pair has a higher priority if the *PRI* value of this pair is smaller.

Algorithm 1: Dynamic and Heuristic Action Scheduling

Input: An action operator P on a type of device and the set of all m devices of the type $D = (d_1, d_2, \ldots, d_m)$.
The streaming action requests $R = (r_1, r_2, \ldots, r_n, \ldots)$ appear in the input queue of P.
Each $r_i \in R$ has a deadline DL_i and a set of candidate devices $D_i \in D$.

Output: A schedule of R on D. Each $r_i \in R$ is either assigned to and serviced by a device $d \in D_i$,
or is dropped due to the missing of its deadline.

1: **for** each device $d_j \in D$ ($1 \le j \le m$) **do**
2: d_j.*nextFreeTime* = *\$now*; /* *nextFreeTime* indicates the next time when d_j will become free */
3: poll and store the current physical status of d_j; /* *\$now* denotes the current system time */
4: **while** *true* **do**
5: **for** each action request r_i in R **do**
6: **if** (*\$now* >= DL_i) **then** remove r_i from R; /* r_i is dropped due to the missing of its deadline */
7: $r_s = null$; $d_s = null$; $PRI_s = +inf$; /* request r_s is to be scheduled on device d_s in this round */
8: **for** each device $d_j \in D$ **do**
9: **if** d_j.*nextFreeTime* > *\$now* **then** continue; /* d_j is currently busy */
10: d_j.*nextFreeTime* = *\$now*; update the new physical status of d_j;
11: **for** each action request r_i in R with $d_j \in D_i$ **do**
12: $cost_{ij}$ = *estimateCost*(r_i, d_j, M); /* M is the cost model used to estimate the cost of r_i on d_j */
13: **if** (*\$now* + $cost_{ij}$ > DL_i) **then** continue; /* deadline of r_i cannot be caught on d_j at this time */
14: PRI_{ij} = *computePriority*(r_i, d_j, $cost_{ij}$);
15: **if** $PRI_{ij} < PRI_s$ **then**
16: $r_s = r_i$; $d_s = d_j$; $PRI_s = PRI_{ij}$;
17: **if** PRI_s != $+inf$ **then**
18: remove r_s from R and service it on d_s;
19: d_s.*nextFreeTime* = *\$now* + the cost of servicing r_s on d_s;
20: **if** *\$now* < *min*{d_j.*nextFreeTime*} ($1 \le j \le m$) **then** sleep until *min*{d_j.*nextFreeTime*};
21: **else** sleep until the arrival of a new action request;

If it happens that no free device is eligible for the requests in the queue, Algorithm 1 is paused to avoid the extensive computation overhead of vain loops. The execution of the algorithm will be resumed by the query optimizer later when a new request arrives in the queue (Line 21). On the other hand, if the algorithm detects that all devices are currently busy, it pauses the execution of itself until the nearest time in future when at least one of these devices become free (Line 20).

In the following of this section, we present in more detail a number of important issues in our algorithm design. We describe how the algorithm deals with each characteristic of the action scheduling problem in Section 4.1. We introduce the model for cost estimation of action executions in Section 4.2. In Section 4.3, we describe how the priority of a request on a device is heuristically computed. The assignment of default deadlines to non-real time requests is discussed in Section 4.4.

4.1 Dealing with action scheduling characteristics

We have considered all six characteristics of the action scheduling problem in the design of our heuristic algorithm. The algorithm incorporates a corresponding approach to deal with each characteristic. For the action-device interaction, Algorithm 1 updates real-time physical status of a free device (Line 10) before computing the priorities of the requests on the device in each round of scheduling. The physical status update involves the process of sending a request message to the device and parsing real-time values of the status attributes from the response message of the device [30].

The LS discipline adopted by Algorithm 1 efficiently enables the dynamic model that our action scheduling problem requires. No matter what the arrival pattern and rate of action requests are, the algorithm performs a round of scheduling when and only when a device is free and there are requests in the queue that the free device is eligible for.

As a real-time scheduling algorithm, Algorithm 1 repeatedly examines the deadlines of the action requests in the queue and drops a request immediately when it detects that the deadline of the request cannot be caught. Moreover, the deadline of a request is considered as a parameter in the priority computation of the request in Function *computePriority* (see Section 4.3).

Before a device finishes a previous action request, Algorithm 1 will not schedule a succeeding request on the device. This ensures that the action requests are serviced on the devices in an independent and non-preemptive manner. The unrelated environment and eligibility restrictions of devices are handled in the algorithm by examining only the action requests that a free device is eligible for and re-computing the cost of each request on a free device in every round of scheduling.

4.2 Cost model for actions

To determine whether an action request can be serviced timely, Algorithm 1 requires a cost model to estimate the response time, and probably the cost values under other metrics, of the request on a device (Line 12). For this purpose, we have previously developed a cost model for actions using response time as the cost metric [29].

Given the physical status of a device, the cost model is able to accurately estimate the response time of any request on the device. The core component of the model is a set of action profiles, each of which specifies the composition of an action in Aorta in terms of the sequential and/or parallel execution of a number of atomic operations. These atomic operations are specific to the type of device the action is executed on and are pre-defined in our Aorta system. Their costs are obtained from empirical measurements in our study. The cost of an action on a device is then estimated using the action profile, the estimated costs of the atomic operations on the type of device, and the physical status of the device [29]. We have implemented the cost model in Aorta and validated its correctness using actions on real devices including cameras and robots. Unless otherwise specified, when we say the “cost” of an action request in the following of this paper, we mean the response time of the request on a device.

Although we use response time as the single cost metric in our study, our proposed action scheduling algorithm is general and is indifferent to the specific metrics used in the cost model. The cost values in Algorithm 1 (Lines 12-14, 19) can be evaluated using other cost metrics for actions, e.g. power consumption, or the combination of multiple metrics. The change of cost metrics will not affect the applicability of the algorithm at all. The only requirement of the algorithm is that a cost model with response time as one of its metrics must be available. In addition, the model may be flexibly designed to selectively involve a few other metrics and compute a more generic request cost value to be used in the algorithm.

Because we focus on action scheduling in this paper, we omit the computation formulas in our cost model and refer interested readers to our previous work for the details [29].

4.3 Priority computation

As a main sub-procedure of the algorithm, Function *computePriority* uses Equation (1) to compute the PRI_{ij} value of a request-device pair (r_i, d_j) as the multiplication of three values: the *basic priority* of the pair, the *current eligibility degree* and the *current reliability degree* of device d_j. These three values are denoted as B_{ij}, E_j and R_j and are computed using Equations (2), (5) and (6), respectively.

$$PRI_{ij} = B_{ij} * E_j * R_j \tag{1}$$

$$B_{ij} = C_{ij} + W_1 * DL_i + W_2 * CDN_i \tag{2}$$

$$W_1 = \sum_{k=1}^{nj} C_{kj} \Big/ \sum_{k=1}^{nj} DL_k \tag{3}$$

$$W_2 = \sum_{k=1}^{nj} C_{kj} \Big/ \sum_{k=1}^{nj} CDN_k \quad (4)$$

$$E_j = 1 + n_j / n \quad (5)$$

$$R_j = 1 + FP_j \quad (6)$$

Equation (2) computes the basic priority B_{ij} of request r_i on device d_j as the weighted sum of three parameters: (i) the current cost C_{ij} of servicing r_i on d_j, (ii) the deadline DL_i of r_i, and (iii) the number of candidate devices CDN_i of r_i. The intuition is that the smaller the cost of a request on a free device, or the more urgent the request, or the less flexibility to schedule the request on the other devices, the higher priority should be given to schedule the request on this device. As a result, our scheduling heuristic is a weighted combination of three simple heuristics under LS discipline in existing work on job scheduling: whenever a machine is free, select the job with the minimum processing time [3][18], or with the minimum deadline [18], or with the least number of candidate machines [16] to be first processed on the machine.

Among the three parameters in Equation (2), we choose cost as the base parameter for the weighted-sum computation. Our consideration is to use the most dynamic parameter as the base in order to make the computed B_{ij} value as specific to the request-device pair (r_i, d_j) as possible. Unlike the value of C_{ij}, the values of DL_i and CDN_i depend on r_i only but not d_j and do not change along with the physical status of d_j.

Rather than setting system-defined static values, we use an adaptive mechanism to adjust the two weights W_1 and W_2 used in Equation (2). When computing B_{ij} in a particular round of scheduling, the set of all unscheduled requests that device d_j is eligible for are identified. Denote this set of requests as R_j. The values of W_1 and W_2 in this round are then computed using Equations (3)-(4) as the sum of current costs on d_j of all requests in R_j divided by the sum of deadlines or candidate device numbers of these requests.

This adaptive setting of weight values roughly maps the values of DL_i and CDN_i into the same magnitude of C_{ij}. It avoids the problem that the large magnitude of one parameter will dominate those of the other two when they are added. Moreover, overall information about the three parameters of all requests in R_j is considered when the basic priority of each request is computed. We use a simple summation to model this overall information to keep the computation cost of Equation (2) negligible, so that the running time of Algorithm 1 will not be greatly affected by the frequent invocations of such computation. We have tried and tested a few alternative combinations of the three parameters in Equation (2), such as multiplication and priority-based ordering in a list. By extensive trial experiments, we found that they all induce worse performance on action scheduling than the summation.

The essence of the multiplication of B_{ij} by E_j and R_j in Equation (1) is to use real-time status of a device to adjust the priority of request scheduling on the device. In Equation (5), the current eligibility degree E_j of a device d_j is computed as one plus the size n_j of R_j divided by the number n of all requests in the queue. The purpose is to give a higher priority to request scheduling on a device that is eligible for a smaller percent of the total requests. A larger value of E_j implies that the device d_j is currently eligible for more requests and a lower priority will be given to request scheduling on the device.

The current reliability degree R_j of a device d_j computed by Equation (6) is introduced because action executions occasionally fail on unreliable physical devices due to temporary hardware malfunctions [30]. Consequently, it is more desirable to schedule requests on a device that induce fewer failures of action executions in history. In the equation, FP_j is the percentage of action executions on d_j that has failed. The value is kept track by the Aorta query optimizer since the system starts. We assume that there is a way for the optimizer to know whether an action execution on a device is failed or not, either through a system error returned by the invocation of the function code block or by a notification message from the application. Same as E_j, the larger the value of R_j is the more unreliable and unfavourable to schedule a request on the device d_j.

4.4 Default deadline assignment for action requests

Algorithm 1 assumes that every action request has a deadline. The deadline of a request is specified in the DEADLINE clause of the query that issues the request. If this optional clause is not provided for a query, which suggests all requests from the query are not real-time, Aorta uses a simple scheme to assign a system-default deadline to the requests.

In the scheme, we let the evaluation plan of a query that has no DEADLINE clause dynamically maintain the average request arrival rate S_a of the query. Suppose T is the interval between the current system time and the time when Aorta starts, and there are N_a action requests issued from the query within T. The current value of S_a is estimated as N_a / T. If a new request is issued from the query at this time, its deadline is assigned to be the current system time plus $1 / S_a$. The motivation of this scheme is a previous request from a query should have been serviced before the next request from the same query appears.

5 Performance evaluation

We have evaluated the performance of our real-time action scheduling algorithm using two actions in the pervasive lab monitoring application: the *warn* action on robots and the *photo* action on cameras, whose operations have been described in Section 3. We call our algorithm DPH (Dynamic, Priority-based Heuristic action scheduling) in the experiments.

5.1 Experimental setup

Parameter	Description
N_d	Number of devices
λ	Mean arrival rate of the requests (unit: requests/second)
w	Weight parameter indicates the deadline range of the requests
f	Maximum failure rate of action executions on the devices
S_d	Dropping rate
T_a	Average service time of a scheduled request
T_s	Average scheduling time of a request

Table 1: Symbols for an action scheduling workload.

5.1.1 Simulation platform

The pervasive lab only has a small number of real devices. To enable large-scale and controllable performance studies of our algorithm, we have developed home-grown robot and camera simulators to simulate the ER1 robots [6] and the AXIS 2130 cameras [2] in the lab on which the *warn* or *photo* action is executed.

We tuned the simulators through extensive tests using the real devices, and made an operation executed on a simulated robot or camera has very similar effects to that on a real device such as the cost of the operation and the change of physical status the operation results in. All experiments we present were conducted using the two simulators.

5.1.2 Workload generation

We generated synthetic scheduling workloads of *warn* or *photo* action requests and used these workloads as the input traces for our simulation studies. Each workload contains totally 1000 requests for an action that arrives dynamically over time to be scheduled on a number of simulated devices. The arrival of the requests follows a Poisson process [11][17] with a mean arrival rate λ. Each of the devices was assigned with a failure rate to indicate the probability that a request scheduled on the device will fail. The failure rate of a device was uniformly picked from the range $[0, f]$. $f \in [0, 1)$ is a parameter we set to limit the maximum failure rate of action executions on the devices.

In a scheduling workload we generated, the cost of an action request on a candidate device was randomly and uniformly picked from the cost range of the action. In the pervasive lab monitoring application, the cost range in seconds of a *warn* action is [8.32, 43.73], and that of a *photo* action is [0.41, 8.87]. The deadline of a request was uniformly picked from the range [*avg_cost*, *w* * *avg_cost*]. *avg_cost* denotes the cost of the action execution on a device in the average case and its value is 26.03 sec for *warn* or 4.64 sec for *photo*. The weight *w* is a parameter we set to examine how the tightness of request deadlines affects the performance of various heuristic scheduling algorithms we compared in the experiments.

The distribution of candidate device numbers of the requests in a workload was one of two kinds: (i) the candidate device number of each request was uniformly and independently picked from the range $[1, N_d]$, or (ii) the candidate device numbers of all requests were picked from $[1, N_d]$ and follow a Zipfian distribution as a whole. N_d is the number of devices involved in the scheduling. For brevity, we call the workloads with these two kinds of distributions the *Uniform* and *Zipfian* workloads in the experiments. To be more specific, in a Zipfian workload many requests have only one candidate device, fewer requests have two, even fewer have three and so on. As a result, a large portion of the requests in a Zipfian workload are skewed on a small subset of the devices.

5.1.3 Performance metrics

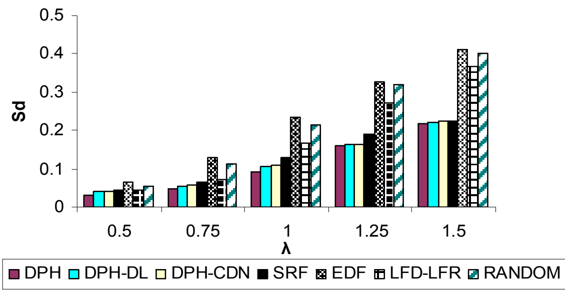
The performance metrics we studied for an action scheduling workload include: (i) the dropping rate S_d, (ii) the average service time of a scheduled request T_a, and (iii) the average scheduling time of a request T_s. The *dropping rate* is defined as the percentage of action requests in the workload that are dropped in the scheduling due to deadline missing. The *average service time of a scheduled request* is defined as the average response time of the scheduled requests in the workload. The *average scheduling time of a request* is defined as the average computation cost that the algorithm spends on scheduling one request in the workload.

A good algorithm for our real-time action scheduling problem must first achieve a small dropping rate of the action requests over time. In other words, the dropping rate should be considered as the primary performance metric for our problem. Under a stable dropping rate, it is desirable that on average the scheduled requests are serviced as fast as possible so that applications requiring the action executions are responded rapidly. Furthermore, the running time of the algorithm must be negligible to ensure the scheduling process will not add considerable delay to the requests waiting in the queue.

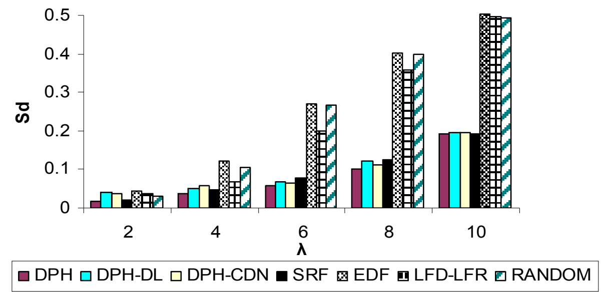
Table 1 summarizes the symbols we used in the experiments to denote the parameters and metrics of an action scheduling workload.

5.2 Comparison of various scheduling heuristics

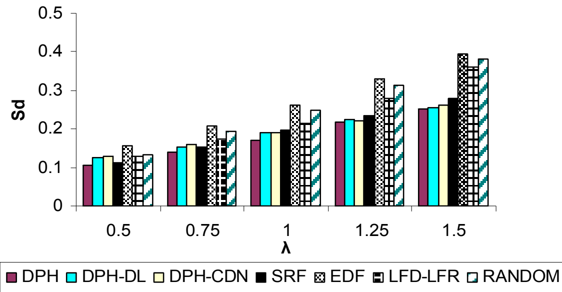
The core of our DPH algorithm is the scheduling heuristic that computes the priority of a request on a device as the weighted sum of three parameters: the cost on the device, the deadline and the candidate device number of the request. In this section, we validate the choice of using cost as the base parameter in the weighted-sum computation of our heuristic and demonstrate the performance benefit of our heuristic over existing dynamic scheduling heuristics in the literature.



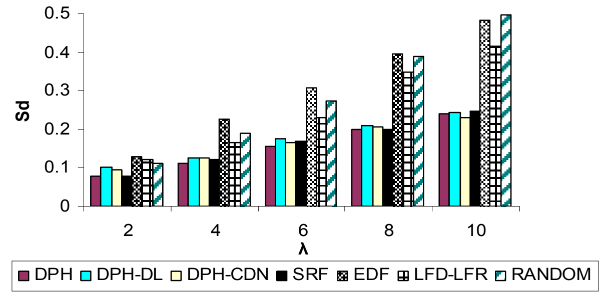
(a) *warn* + Uniform



(b) *photo* + Uniform

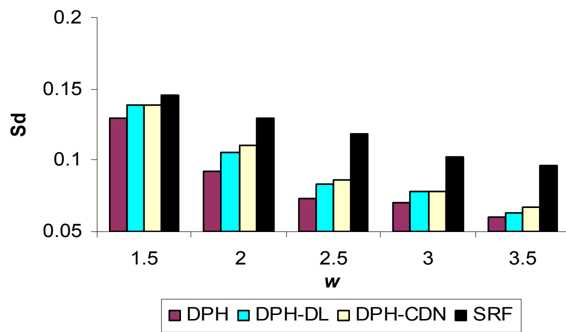


(c) *warn* + Zipfian

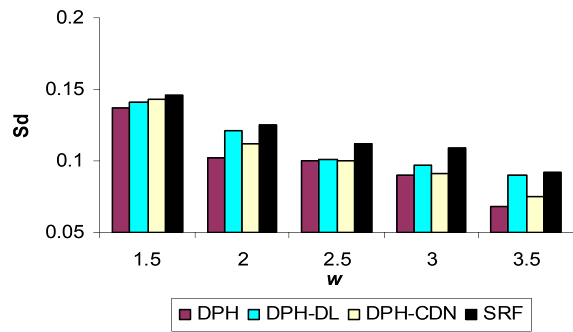


(d) *photo* + Zipfian

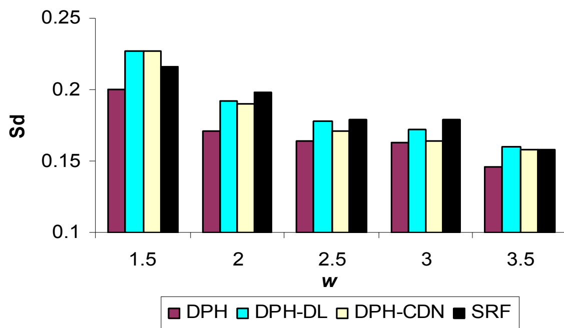
Figure 5: Dropping rates of seven scheduling heuristics with different request arrival rates.



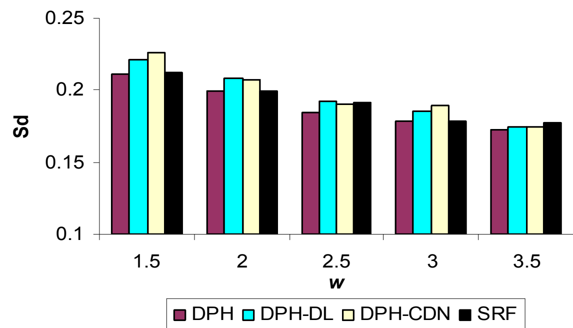
(a) *warn* + Uniform



(b) *photo* + Uniform



(c) *warn* + Zipfian



(d) *photo* + Zipfian

Figure 6: Dropping rates of four scheduling heuristics with different *w* parameters.

We compared our heuristic with three heuristics widely used for classic job scheduling: (i) shortest request first [3][18], (ii) earliest deadline first [18], and (iii) least flexible device with least flexible request first [16]. In the experiments, we replaced our heuristic with one of these heuristics in DPH while keeping other sub-modules of the algorithm unmodified. The consequent algorithms are denoted as SRF, EDF and LFD-LFR, correspondingly. We picked the three heuristics for performance comparison with ours because they have been illustrated to be effective for different variants of the parallel machine scheduling problem under a number of performance metrics [16]. Moreover, they can be easily used with the LS discipline to enable a dynamic scheduling model as our heuristic.

Whenever a device becomes free in a round of scheduling, SRF, EDF and LFD-LFR select a request in the queue to be serviced on the device if and only if the request has the smallest cost on the device, the earliest deadline, or the least number of candidate devices. If multiple devices are free at the same time, SRF first schedules the request-device pair with the smallest cost, EDF first schedules the request with the earliest deadline on any free candidate device, and LFD-LFR first schedules a request on the device that is eligible for the least number of requests in the queue.

To verify the correct design of our heuristic, two variants of it using deadline or candidate device number as the base parameter in the weighted-sum computation were also included in the performance comparison. We denote DPH with these two variant heuristics as DPH-DL and DPH-CDN. As the baseline of performance analysis, we included a RANDOM algorithm in the comparison. Under the LS discipline, the algorithm randomly schedules each request on one of its candidate devices.

We first examined the dropping rates achieved by different scheduling heuristics. Figure 5 shows the metric values of the seven algorithms with varied request arrival rate. For each action scheduling workload in this experiment, the values of N_d and w were fixed to be 20 and 2.0, and the failure rate of every device was set to be zero (i.e., $f = 0.0$). Each value in the figure is the average of ten independent runs of the experiment. This is the same for all figures and tables shown in the following. We used different λ values for the *warn* and *photo* workloads in the figure due to the different cost ranges of the two actions. The values in the figure were subtly picked to demonstrate the performance differences between the algorithms from the situation that the devices in a workload were slightly underloaded to the situation that they were heavily overloaded by the requests in the workload.

In Figure 5 we see that, all algorithms dropped a larger percentage of requests when the requests were arriving at a faster rate. With the same λ value, the algorithms dropped more requests under Zipfian workloads than Uniform workloads. This was because the requests in a Zipfian workload were skewed on few devices and these skewed requests had a higher probability of being dropped due to the overloading of their limited candidate devices. The figure shows that DPH consistently dropped fewer requests than DPH-DL and DPH-CDN under all kinds of workloads, although the performance differences between them were not very large. This result verifies we have chosen the correct base parameter for our heuristic. In the meanwhile, our heuristic has a nice property that its performance is indifferent to the base parameter chosen.

DPH had a significant better performance than EDF, LFD-LFR and RANDOM in Figure 5. The performance difference between DPH and these algorithms got large when the request arrival rate increased. Among the three, the performance of EDF was as bad as RANDOM and LFD-LFR always performed better than both of them. The result indicates that EDF is not suitable for our action scheduling problem, because the dynamic costs and candidate device numbers of action requests have become the dominating factors to the performance of an algorithm for our problem.

The performance of SRF in the figure was much closer to DPH compared to LFD-LFR. It was even better than DPH-DL and DPH-CDN in several cases. This was because cost is the most dynamic parameter in our problem and the cost of an action request on a device will change dynamically in the scheduling process with the physical status updates of the device. As a result, an unscheduled request that currently has a larger cost on a device has the opportunity to get its cost on the device decreased after the physical status of the device is changed. This effect makes the minimization of the current request cost on a free device more beneficial in our problem than in classic job scheduling.

DPH still outperformed SRF noticeably as shown in Figure 5, especially when the devices had a moderate load in the middle part of each sub-figure. The performance benefit of DPH over SRF was larger under Uniform workloads than Zipfian workloads, and was larger under *warn* workloads than *photo* workloads. This indicates that when requests are skewed or the cost range of action is small, the performance of two reasonable scheduling heuristics gets closer and the opportunity for performance improvement is lowered. More specifically, DPH dropped 3%-30%, 3%-24%, 8%-14% and 1%-10% fewer requests than SRF in Figure 5(a)-(b), respectively.

Figure 6 shows the dropping rates of DPH, DPH-DL, DPH-CDN and SRF when the w parameter for request deadline generation varied. We still fixed $N_d = 20$ and $f = 0.0$ in this experiment, and set $\lambda = 1.0$ for the *warn* workloads and $\lambda = 8.0$ for the *photo* workloads. No matter

how w is varied, EDF, LFD-LFR and RANDOM performed much worse than the other four algorithms the same as in Figure 5. As a result, we do not include the results for these algorithms in Figure 6 in order to make the performance differences between the other four algorithms more apparent.

As expected, more requests were dropped by every algorithm when the deadlines of requests got smaller. DPH consistently outperformed the other three no matter how the value of w was changed. Figure 6(a)-(b) indicates under Uniform workloads the performance difference between DPH and SRF increased when the request deadlines were large. For example, DPH dropped 11%, 30% and 38% fewer requests than SRF when the value of w was set to be 1.5, 2.0 and 2.5 under the *warn* Uniform workloads. In comparison, the performance difference between DPH and SRF remained nearly constant under Zipfian workloads, as shown in Figure 6(c)-(d).

Workload \ Algorithm		DPH	DPH-DL	DPH-CDN	SRF	EDF	LFD-LFR	RANDOM
warn	Uniform	19.77	20.12	20.21	18.17	23.12	23.16	22.94
	Zipfian	17.92	21.11	21.47	17.44	22.83	22.54	22.68
photo	Uniform	2.58	2.66	2.58	2.27	4.2	4.07	4.19
	Zipfian	2.64	2.95	2.95	2.24	4.05	3.98	4.24

Table 2: T_a achieved by different scheduling heuristics (unit: second)

Workload \ Algorithm		DPH	SRF	EDF	LFD-LFR	RANDOM
warn	Uniform	0.291	0.279	0.246	0.268	0.252
	Zipfian	0.266	0.257	0.237	0.252	0.241
photo	Uniform	0.113	0.110	0.075	0.080	0.076
	Zipfian	0.100	0.101	0.076	0.082	0.077

Table 3: T_s achieved by different scheduling heuristics (unit: second)

We next investigated the average service time and average scheduling time achieved by different scheduling heuristics. The results are listed in Tables II and III. The parameter setting we used in this experiment was N_d = 20, w = 2.0, f = 0.0, λ = 1.0 for the *warn* workloads and λ = 8.0 for the *photo* workloads.

In Table 2, SRF achieved the smallest T_a among the seven heuristics under all kinds of workloads. This was because SRF always picked the request currently in the queue that had the smallest cost on a free device to be serviced in each round of scheduling. Considering the unknown arrival pattern of action requests and the dynamically changing costs of them on devices, we expected SRF would have the best possible performance on T_a in practice. Our DPH achieved the second best T_a among the heuristics compared. EDF and LFD-LFR performed as bad as RANDOM in this experiment due to the inconsideration of the request cost in these heuristics. The result further proved that the performance benefit of DPH over its variants DPH-DL, DPH-CDN and the existing EDF, LFD-LFR heuristics for our action scheduling problem.

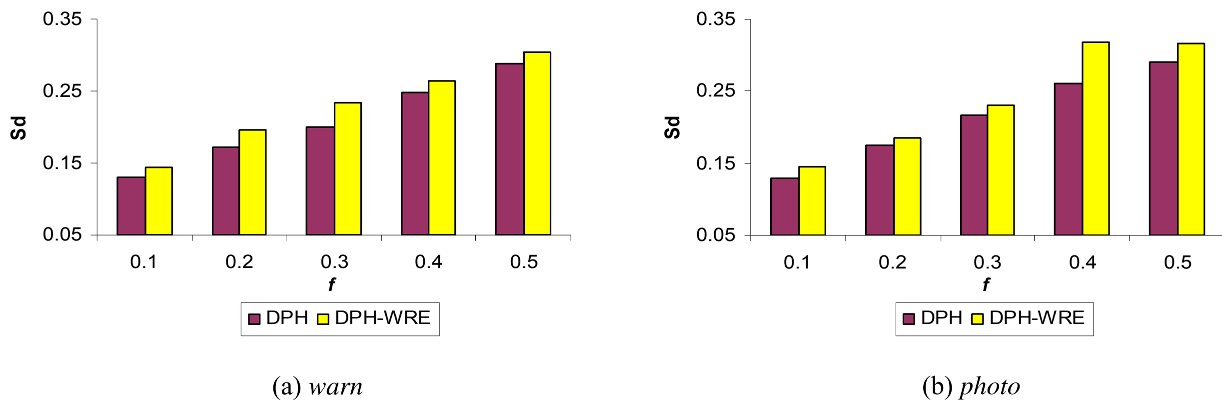
The difference between DPH and SRF in Table 2 was consistently smaller than 0.5 seconds indifferent to what kinds of workloads we used. This difference is insignificant to the *warn* action but non-negligible to the *photo* action, considering the different cost ranges of the two actions. The result implies that DPH can perform as well as SRF in T_a when the average cost of the action to be scheduled is in the magnitude of tens of seconds or larger.

Table 3 shows that the average scheduling time of DPH was about the same as that of SRF under all kinds of workloads. On average DPH only required 1-12 milliseconds more computation cost than SRF in each round of scheduling. Since SRF is one of the existing scheduling heuristic requiring the simplest processing logic, the result indicates that the computation cost of DPH is negligible. The average scheduling time of DPH-DL and DPH-CDN are not shown in the table because they were almost the same as that of DPH in the experiment. The small scheduling time of EDF, LFD-LFR and RANDOM in the table was due to the fact that these algorithms had a much larger dropping rate than DPH or SRF as shown in Figures 4-5, so that fewer requests are remained to be selected from in each round of scheduling.

We have run a number of additional experiments using different settings of N_d, w and λ values in the workloads. All these experiments came up with results that are consistent with those revealed by Figures 4-5 and Tables 2-3. We omit the details here.

In summary, our DPH algorithm significantly outperformed existing EDF and LFD-LFR heuristic algorithms for action scheduling on the dropping rate and the

average service time of a scheduled request. In comparison with SRF, DPH performed better in the dropping rate and performed equally well in the average service time when the action cost magnitude is not smaller than tens of seconds. When the cost magnitude is small, DPH performed a little worse than SRF in the average service time but still dropped noticeably less requests than SRF in the scheduling process. With a similar dropping rate, the scheduling time of DPH was as small as those required by existing heuristic algorithms.



(a) *warn* (b) *photo*

Figure 7: Dropping rates of DPH with and without reliability estimation on devices.

5.3 Effectiveness of reliability estimation

We validate the effectiveness of involving device reliability estimation in the priority computation of our heuristic in this section. We modify Equation (1) to exclude *Rj* from the multiplication and denote the consequent variant of our algorithm DPH-WRE (Without Reliability Estimation). For each workload in this experiment, the parameters other than f were fixed to be $N_d = 20$, $w = 2.0$, $\lambda = 1.0$ for the *warn* workloads and $\lambda = 8.0$ for the *photo* workloads.

We varied the value of the parameter f from 0.1 to 0.5 in the experiment and the dropping rates of DPH and DPH-WRE under Uniform workloads are shown in Figure 7. Because our results showed that the average service time and scheduling time were hardly affected by the existence of reliability estimation, we omit the results for these two metrics.

In Figure 7 we see that, compared to its counterpart without reliability estimation, DPH was able to reduce the number of dropping requests by 6%-19% under both the *warn* and *photo* workloads. We have tried Zipfian workloads and other parameter settings of the workloads. The experiments all had very similar results to those in Figure 7.

6 Related work

There have been a number of publications in pervasive computing that study using sensor readings to actuate the operations on different types of devices, such as sensor nodes [12], cameras [24], phones [22], vehicles [28] and lights [7]. The EnviroTrack system [1] tracks and invokes pre-defined computations in response to environmental objects like vehicles over networks of sensor nodes. Flinn et al. proposed a remote execution system in pervasive computing named Spectra [8]. Their idea of best execution plan selection for an application has a similar goal to our scheduling of an action request on the best candidate device.

User-defined functions have been widely supported in commercial DBMS products [9][14][15]. These functions run outside the core of the DBMS and their side effects to data (e.g., updates) are never considered during execution. In Aorta, we put actions as query operators and effectively schedule their executions on the devices. Our action scheduling algorithm considers the effects of action executions on the devices. The physical status change of a device incurred by an action execution is examined before the next execution is scheduled on the device.

Braun et al. [3] have compared the performance of eleven static heuristics for parallel machine scheduling under various kinds of task workloads. Their results show that in most cases the simple Min-min heuristic performs better than other complex heuristics such as Generic Algorithms, Simulated Annealing and A*. Most heuristics studied in the paper require a complete knowledge about a static task set before the scheduling is performed so they are inapplicable to our dynamic scenario.

Moreover, the Min-min heuristic the authors identified to have a generally good scheduling performance is essentially the same as the SRF algorithm that we have studied in the experiments. Our results have illustrated that our proposed algorithm noticeably outperforms SRF on the dropping rate of dynamically- arriving action requests.

Previous work on multiprocessor scheduling in real-time systems [13][18] takes static sets of tasks as the scheduling input and aims at enhancing the schedulability of these task sets. Their proposed scheduling algorithms are based on a branch-and-bound tree search with backtracks and a weighted-sum heuristic integrating the deadline and the earliest start time of a task. These algorithms are too running time-intensive to be adaptable to our scenario and no approach is applied in them to adaptively fix the weight value used in the heuristics as our algorithm. The resource requirements of tasks other than CPU processing time or the parallelization of a task on multiple processors have also been considered in this scheduling work. These issues are not related to our action scheduling problem.

Many load balancing approaches have been proposed in distributed systems for the non-real time scheduling of jobs on an interconnected network of computers [11][19][20][25][26]. Each job first arrives at the local queue of a computer and then transmitted to be processed by another computer if the original one has been overloaded. In comparison, in our action scheduling problem all devices of a type share a global queue of action requests and there is no communication among devices in the scheduling process. The scheduling model in this distributed computing scenario is dynamic as ours. However, the computers are all regarded as identical rather than unrelated and there is no job deadline involved in the problem.

Reliability-driven job scheduling in parallel and distributed systems has attracted many research efforts in the literature [17][23]. A reliability model that assumes the independent failures of jobs on the computers following a Poisson probability distribution is proposed to drive the scheduling process combined with the job costs. In comparison, in our heuristic action scheduling algorithm we estimate the reliability of an action request on a device in a much simpler way based on computing the average failure rate of requests on the device in history.

7 Conclusion

We have presented the design of the new dynamic and heuristic algorithm for real-time action scheduling in our current prototype of Aorta, which is a query processing system for pervasive computing. We make actions as query operators in Aorta and share a single action operator among the plans of multiple concurrent queries having actions on the same type of device. Each action operator adaptively performs the process of action scheduling on all devices of the type using the scheduling algorithm we propose.

We identify all characteristics of the action scheduling problem we study and apply corresponding approaches to handle each characteristic in our algorithm. The heuristic we develop for action scheduling is based on the priority computation for each unscheduled action request that a free device is eligible for and selecting the request with the highest priority to be serviced by the device at this time. The priority of a request on a device incorporates many parameters involved in the scheduling: the cost of the request on the device, the deadline and the candidate device number of the request, the current eligibility and reliability degree of the device.

We have performed simulation studies to compare our priority-based heuristic with three heuristics for dynamic scheduling in the literature. The results demonstrated the performance benefit of our heuristic over these existing heuristics. We have also conducted experiments to validate our choice of using cost as the base parameter in the weighted-sum computation of our heuristic as well as the effectiveness of the reliability estimation on devices in the heuristic.

References

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th Inter-national Conference on Distributed Computing Systems*, 2004, pp. 582-589.

[2] Axis Communications. http://www.axis.com.

[3] T.D. Braun, H.J. Siegel, N. Beck, L.L. Bölöni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen, and R.F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing* 61(6) (2001) 810-837.

[4] Crossbow Inc. http://www.xbow.com.

[5] M.L. Dertouzos and A.K.L. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering* 15(2) (1989) 1497-1506.

[6] Evolution Robotics. www.evolution.com.

[7] C. Feng, L. Yang, J.W. Rozenblit, and P. Beudert. Design of a wireless sensor network based automatic light controller in theater arts. *In Proceedings of the 14th International Conference and Workshops on the Engineering of Computer-Based System*, 2007, pp. 161-170.

[8] J. Flinn, S.Y. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. *In Proceedings of the 22nd International Conference on Distributed Computing Systems*, 2002, pp. 217-226.

[9] IBM DB2. www.ibm.com/db2.

[10] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. *In Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, 2000, pp. 56-67.

[11] H. Lin and C. Raghavendra. A dynamic load-balancing policy with a central job dispatcher (LBC). *IEEE Transactions on Software Engineering* 18(2) (1992) 148-158.

[12] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems* 30(1) (2005) 122-173.

[13] G. Manimaran and C.R. Murthy. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems* 9(3) (1998) 312-319.

[14] Microsoft SQL Server. www.microsoft.com/sql/.

[15] Oracle Database. http://www.oracle.com/database/index.html.

[16] M. Pinedo. *Scheduling Theory, Algorithms, and Systems*. 2nd Edition, Prentice Hall, 2002.

[17] X. Qin and H. Jiang. A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs on heterogeneous clusters. *Journal of Parallel and Distributed Computing* 65(8) (2005) 885-900.

[18] K. Ramamritham, J. Stankovic, and P.F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 1(2) (1990) 184-194.

[19] K.W. Ross and D.D. Yao. Optimal load balancing and scheduling in a distributed computer system, *Journal of the Association for Computing Machinery* 38(3) (1991) 676-689.

[20] R. Shah, B. Veeravalli, and M. Misra. On the design of adaptive and decentralized load balancing algorithms with load estimation for computational grid environments. *IEEE Transactions on Parallel and Distributed Systems* 18(12) (2007) 1675-1686.

[21] C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor information networking architecture and applications. *IEEE Personal Communications* 8(4) (2001) 52-59.

[22] F. Siegemund and C. Flörkemeier. Interaction in pervasive computing settings using bluetooth-enabled active tags and passive RFID technology together with mobile phones. *In Proceedings of the 1st International Conference on Pervasive Computing and Communications*, 2003, pp. 378-387.

[23] S. Srinivasan and N.K. Jha. Safety and reliability driven task allocation in distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 10(3) (1999) 238-251.

[24] N.M. Su, H. Park, E. Bostrom, J. Burke, M.B. Srivastava, and D. Estrin. Augmenting film and video footage with sensor data. *In Proceedings of the 2nd International Conference on Pervasive Computing and Communications*, 2004, pp. 3-12.

[25] R. Subrata, A.Y. Zomaya, and B. Landfeldt. Game-theoretic approach for load balancing in computational grids. *IEEE Transactions on Parallel and Distributed Systems* 19(1) (2008) 66-76.

[26] A.N. Tantawi and D. Towsley. Optimal static load balancing in distributed computer systems. *Journal of the Association for Computing Machinery* 32(2) (1985) 445-465.

[27] M. Weiser. The computer for the 21st century, *Scientific American* 265(3) (1991) 94-100.

[28] Z. Wu, Q. Wu, H. Cheng, G. Pan, M. Zhao, and J. Sun. ScudWare: A semantic and adaptive middleware platform for smart vehicle space. *IEEE Transactions on Intelligent Transportation Systems* 8(1) (2007) 121-132.

[29] W. Xue and Q. Luo. Action-oriented query processing for pervasive computing. *In Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, 2005, pp. 305-316.

[30] W. Xue, Q. Luo, and L.M. Ni. Systems support for pervasive query processing. *In Proceedings of the 25th International Conference on Distributed Computing Systems*, 2005, pp. 135-144.

[31] Y. Yao and J. Gehrke. Query processing for sensor networks. *In Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*, 2003.